Configure a SDN Router to Process Packets with Different Priorities

Xuyang Cao

University of California San Diego, San Diego

December, 2019

Abstract

In a LAN there is usually one router serving as the gateway to the internet. Ubiquitous computing and new communication models have enabled more and more networking devices to transfer data concurrently within a LAN. Among these devices, some may be critical and may require an allocation of constant high bandwidth for them. When competition exists, some less important but data-hungry devices may deplete the bandwidth and hence negatively affect more critical ones. A regular router usually evenly distributes the bandwidth under a competition and hence may fail to guarantee QoS (Quality of Service) for certain devices. In this report I aim to build a router using SDN (software-defined networking) technologies under Mininet's simulated environment that will be able to process packets from different IP addresses with different priorities and hence ensure a certain proportion of bandwidth is achieved for certain devices. Also, I hope this technical report can provide a summary of how to build a general SDN router using the described tech stack so you can further customize it based on the actual needs.

1 Introduction and Technologies

In this part, what technologies are used and what are their roles in this project are briefly discussed, coupled with some recap of general network-layer knowledge. In this way, I hope readers could have a general if not complete understanding of tools and terms mentioned later and feel more comfortable reading instead of constantly being interrupted by them and jumping into searching, as there are various niche products and network-related tools that not everyone has used before.

- Mininet: an open-source network emulator tool originally developed by Stanford university. It allows people to design topology, create SDN, generate the corresponding realistic virtual network and test. Mininet alleviates the harassment of establishing the physical topology. Once the work is done on Mininet, it can be migrated to some actual hardware and put into production[1].
- Control plane and data plane in network layer: in computer networking, the network layer can be further divided into these two planes. Data plane is where data, or packets, are processed and forwarded to where they should go, so basically all switching devices doing such forwarding tasks belong to the data plane. On the other hand, the control plane is where instructions of how to forward packets are generated and sent to those switching devices so the internet can be operated in a way we expect. Control plane can be a piece of software, and our goal in this project is to write such software in the control plane and link it to the router in our topology.
- Open vSwitch: the type of virtual switch available in Mininet. As a data-plane component, it can be customized and controlled by control-plane components to have desired switching and forwarding behaviors.
- OpenFlow: a communication protocol between the control plane and the data plane. It allows the controller to send instructions to and install flow tables in the switching devices and decide how specific packets should be forwarded.
- POX: a Python library available in Mininet. It allows developers to write their own controllers for switching devices in the customized topology.

- Wireshark: a packet analyzer. It allows users to examine the details of packets passing through a specific interface.
- Iperf and Netperf: tools measuring the throughput/bandwidth between two end hosts (by either TCP or UDP).

2 Network Topology and Acceptance Test Plans

To be able to test and verify the effectiveness of our eventual work, we need to first design a static network topology. Given the topology, we can then build our SDN router, put it into use, and set corresponding acceptance test plans based on the structure of the designed network.

Since we want the soft router to be able to handle packets generated from various devices in a LAN, route them to the internet, and, in certain circumstance, to prioritize the forwarding of packets from certain hosts and ensure their high throughput, we can create the following topology:

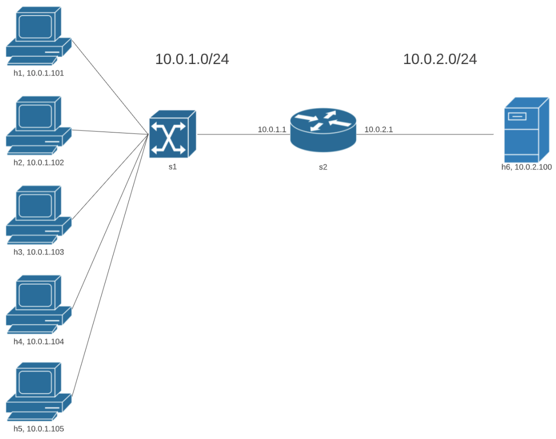


Figure 1: network topology

As the figure indicates, in our project we will create two networks, namely 10.0.1.0/24, 10.0.2.0/24, which are connected by the router s_2 in the middle. On the left of the router, 10.0.1.0/24 will be the home network with 5 hosts h_1, h_2, h_3, h_4, h_5 resembling various computing devices. Since they are in the same LAN, a switch $s1$ is connected to all of them and then connected to the gateway 10.0.1.1. On the right of the router, 10.0.2.0/24 will resemble the internet. We put an actual content server h_6 there so it can communicate and exchange data with our five devices at home. By this means, we can measure the throughput/bandwidth between any pair $(h_x, h6)$, where $x \in \{1, 2, 3, 4, 5\}$. In this way, under different scenarios these measured throughput values can serve as metrics to reflect the correctness as well as the effectiveness of our SDN.

Among those 5 hosts h_1, h_2, h_3, h_4, h_5, we let h_5, or the IP address 10.0.1.105, be the high-priority devices. Therefore, according to the aforementioned competition scenario, when those normal devices (h_1 to h_4) are not competing for resources, each of them should be guaranteed a sufficient amount of bandwidth so that they will be able to achieve various common operations in a stable and healthy manner (e.g. buffering a movie, backing up data, system update, IP phone call between home and work). However, for the high-priority device h_5, since it can represent a handful of data-hungry devices (like HD security monitors), during off-peak time it should be able to utilize nearly all available bandwidth, if necessary. Also, even when the all hosts are busying sending data, with the help of this SDN router h_5 should still tightly control a considerable amount of bandwidth allowing, for instance, several numbers of HD streaming.

Therefore, we can list out several specifications where our acceptance testing will soon be based on:

1. The switch s_1 should allow devices in the home LAN to communicate with each other (i.e. forwarding frames based on the MAC addresses and broadcasting).

2. The router s_2 should allow devices in the home LAN to communicate with the content server h_6 in the internet (i.e. forwarding IP packets).

3. When only one "normal" device in home LAN is (constantly and intensively) using the internet (i.e. exchanging data with h_6), it should have a sufficient amount of bandwidth x, and we set that $300Mbps < x$, given the total available bandwidth is more than $1Gbps$.

4. When several "normal" devices in home LAN are transmitting data with the internet concurrently and competing for the bandwidth resource, they should equally divide the available resources.

5. When only the high-priority device is (constantly and intensively) using the internet, it can occupy (nearly) all the bandwidth and achieve very high throughput.

6. When "normal" devices and the high-priority device h_5 are attempting to transmit unbounded amount of data concurrently, h_5 should be able to occupy most of the bandwidth, while the others can share the remaining small amount of bandwidth.

3 Softerization of Control Plane

As we have discussed in the first section, to enable packet switching/forwarding in a network component, we need to install some software in it so the software can serve as the "brain" and give the hardware correct instructions of what to do when it receive some random packet. In Mininet as well as most SDN-friendly switching devices in the current market, OpenFlow is one of the major protocols used for communications between the control plane and the data plane. Here is the general workflow[3]:

* The control plane (the switching device) and the data plane (the software) doing handshakes periodically to ensure the connection is still on.

1. When a packet arrives at the switching device at some interface, the device will first try to query all the flow entries installed in its flow table. If this packet matches criteria in any specific entry, the packet will then be processed based on the action written in that entry.

2. If no flow entries can match the packet, the packet will be forwarded to the control plane. The software in the control plane will examine this packet, make decisions of what to do with it, encapsulate both the instruction and the packet into a larger-sized OpenFlow packet, and send this OpenFlow packet back to the data plane.

3. The data plane receives the OpenFlow packet, process the original packet based on the instructions, and optionally install a flow entry if it is told to do so (usually it does so for the next time the same packet will no longer need to be sent to the control plane and efficiency is promoted).

Our goal is to write such software so the switch s_1 and the router s_2 will be instructed to meet the specifications described in section 2. In OpenFlow and Mininet, this type of software is called a controller. There are several handy and powerful libraries in popular programming languages that will empower developers to write their own controllers. For example, we have POX and Ryu in Python, Beacon and Floodlight in Java, Trema in Ruby. We use POX in this project, and for any procedure described below, you could always refer to some specific lines in the source codes come with this report.

Here is the high-level procedures we implement and adapt from online public repositories[2] for our POX controller[4] in the control plane:

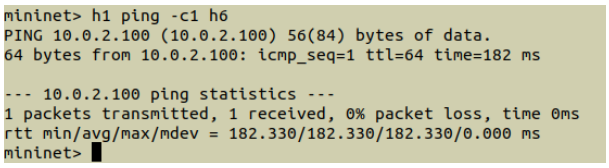
Algorithm 1: Controller for both switch s_1 and router s_2

```
Data: switch's mac address table: switch_mac_to_port; router's routing table:
      static_routing_table; packet received from the data plane: packet_in
Result: OpenFlow packet is sent back to the data plane and instructs the hardware to forward
        packet_in to the correct port
1  switch_mac_to_port ← an empty map;
2  static_routing_table ← map(host IP to array(router interface, host MAC address));
3  Procedure switch(packet)
4      packet_src_MAC ← source MAC address of packet;
5      packet_dst_MAC ← destination MAC address of packet;
6      port_in ← the incoming port of packet;
7      add packet_src_MAC and port_in as a key value pair to switch_mac_to_port;
8      if packet_dst_MAC in switch_mac_to_port then
9          instruct the data plane to forward packet to switch_mac_to_port.packet_dst_MAC;
10         current_flow_entry ← flow entry(criteria1: source MAC address ==
             packet_src_MAC, criteria2: destination MAC address ==
             packet_dst_MAC, action1: forward the packet to port switch_mac_to_port.packet_dst_MAC);
11         instruct the data plane to install current_flow_entry;
12     else
13         instruct the data plane to broadcast packet to all ports except the incoming one;
14     end
15 Procedure end
16 Procedure router(packet)
17     packet_type ← Ethernet type of packet;
18     if packet_type == ARP request targeting 10.0.1.1 or 10.0.2.1 then
19         assemble an ARP reply ARP_reply;
20         source MAC address of ARP_reply ← a fake MAC address like 40:10:40:10:40:10;
21         destination MAC address of ARP_reply ← destination MAC address of packet;
22         instruct the data plane to send ARP_reply to the incoming port of packet;
23     else if packet_type == IP packet then
24         source MAC address of packet ← router's MAC address 40:10:40:10:40:10;
25         query static_routing_table using destination IP address of packet as the key, and we get
             that IP's MAC address final_MAC_addr and egress port port_out;
26         destination MAC address of packet ← final_MAC_addr;
27         instruct the data plane to forward modified packet to port_out;
28         packet_src_IP ← source IP address of packet;
29         packet_dst_IP ← destination IP address of packet;
30         declare variable queue_number;
31         if packet_src_IP == 10.0.1.105 then
32             queue_number ← 1;
33         else
34             queue_number ← 2;
35         end
36         current_flow_entry ← flow entry(criteria1: source IP address ==
             packet_src_IP, criteria2: destination IP address ==
             packet_dst_IP, action1: source MAC address ←
             40:10:40:10:40:10, action2: destination MAC address ←
             final_MAC_addr, action3: enqueue the packet to queue queue_number and forward it to port_out);
37         instruct the data plane to install current_flow_entry;
38     end
39 Procedure end
40 Create a listener listening to the data plane for any incoming packet packet_in;
41 if packet_in is from s_1 then
42     switch (packet);
43 else if packet_in is from s_2 then
44     router (packet);
45 end
```

For a switch, its main job is to handle the frame switching in a LAN. Every time when a packet enters the switch's function, it will write the sender's information into the MAC address table and therefore know which port is the egress port when someone else tries to send something to this sender. In this way, the controller not only can correctly switch frames but also distinguish itself from a hub device, which always inefficiently broadcasts. Then, this series of operations in the control plane is installed as a flow entry to the switch in the data plane so for the next time the same packet will be directly processed and the efficiency is further improved.

As a network-layer component, the router will have more tasks in its to-do lists. For any of the hosts in the home LAN to send data to the internet, it must first send the data packet to its default gateway, which is this router. However, since the host and router are in the same network, they need to know each other's MAC address in order to communicate. Therefore, the router function in our controller program should be able to handle ARP requests from its connected end devices. We use a fake but handy MAC address 40:10:40:10:40:10 for the router's all interfaces so when s_2 receive any ARP request targeting itself, it will generate and send back the ARP reply[9]. In addition, when an IP packet arrives, instead of just forwarding it, the router should update the header of this packet: since the source and destination MAC addresses denote the start and end of only one hop jump, the new source MAC address should be the old destination, and the new destination MAC address should be the destination IP's MAC address. To this point, the packet can be forwarded with no error. However, since we want to prioritize the packet forwarding for host h_5, we can distinguish the senders by putting the incoming packets into different queues of the router[6], and do the prioritization job in the data plane. When a packet arrive at the queue, it will be automatically processed. However, we can customize the queue in Open vSwitch, and the details are included in the next section. Like how to control the switch, this series of actions for packet routing is installed as a flow entry to the router for the future convenience[3].

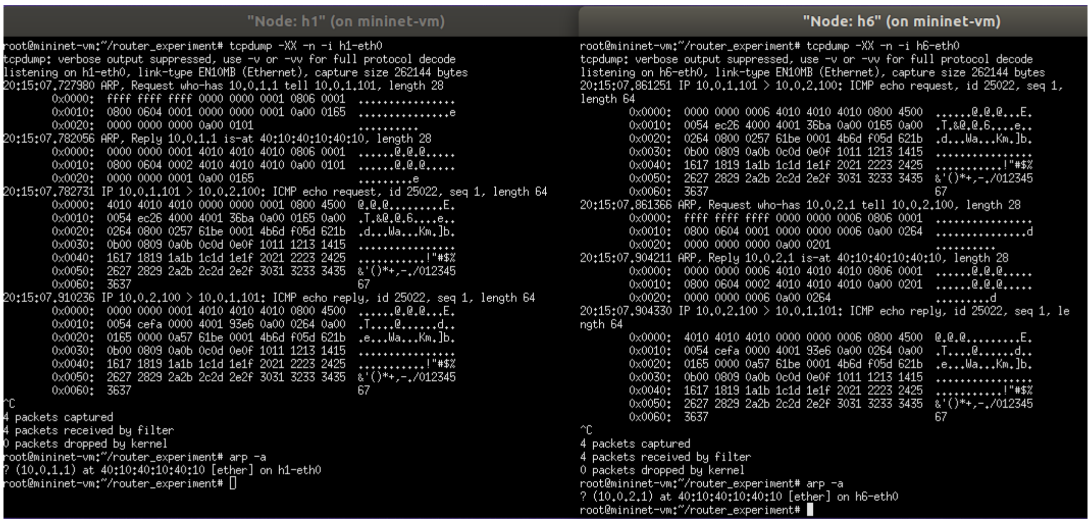
We can verify the switch and router works as we expect by letting h_1 ping h_6. As the following figures indicate, all packets are processed in the way we want:



```
mininet> h1 ping -c1 h6
PING 10.0.2.100 (10.0.2.100) 56(84) bytes of data.
64 bytes from 10.0.2.100: icmp_seq=1 ttl=64 time=182 ms

--- 10.0.2.100 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 182.330/182.330/182.330/0.000 ms
mininet> █
```

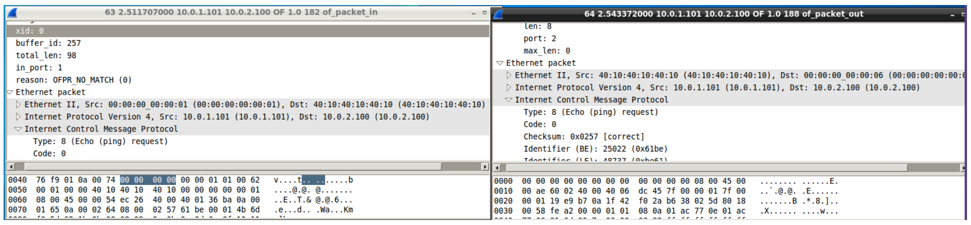
Figure 2: ping is successful



```
root@mininet-vm:~/router_experiment# tcpdump -XX -n -i h1-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h1-eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
20:15:07.727980 ARP, Request who-has 10.0.1.1 tell 10.0.1.101, length 28
        0x0000:  ffff ffff ffff 0000 0000 0001 0806 0001  ................
        0x0010:  0800 0604 0001 0000 0000 0001 0a00 0165  ...............e
        0x0020:  0000 0000 0000 0a00 0101                 ..........
20:15:07.782056 ARP, Reply 10.0.1.1 is-at 40:10:40:10:40:10, length 28
        0x0000:  0000 0000 0001 4010 4010 4010 0806 0001  ......@.@.@.....
        0x0010:  0800 0604 0002 4010 4010 4010 0a00 0101  ......@.@.@.....
        0x0020:  0000 0000 0001 0a00 0165                 .........e
20:15:07.782731 IP 10.0.1.101 > 10.0.2.100: ICMP echo request, id 25022, seq 1, length 64
        0x0000:  4010 4010 4010 0000 0000 0001 0800 4500  @.@.@.........E.
        0x0010:  0054 ec26 4000 4001 36ba 0a00 0165 0a00  .T.&@.@.6....e..
        0x0020:  0264 0800 0257 61be 0001 4b6d f05d 621b  .d...Wa...Km.]b.
        0x0030:  0b00 0809 0a0b 0c0d 0e0f 1011 1213 1415  ................
        0x0040:  1617 1819 1a1b 1c1d 1e1f 2021 2223 2425  ...........!"#$%
        0x0050:  2627 2829 2a2b 2c2d 2e2f 3031 3233 3435  &'()*+,-./012345
        0x0060:  3637                                     67
20:15:07.910236 IP 10.0.2.100 > 10.0.1.101: ICMP echo reply, id 25022, seq 1, length 64
        0x0000:  0000 0000 0001 4010 4010 4010 0800 4500  ......@.@.@...E.
        0x0010:  0054 cefa 0000 4001 93e6 0a00 0264 0a00  .T....@.......d..
        0x0020:  0165 0000 0a57 61be 0001 4b6d f05d 621b  .e...Wa...Km.]b.
        0x0030:  0b00 0809 0a0b 0c0d 0e0f 1011 1213 1415  ................
        0x0040:  1617 1819 1a1b 1c1d 1e1f 2021 2223 2425  ...........!"#$%
        0x0050:  2627 2829 2a2b 2c2d 2e2f 3031 3233 3435  &'()*+,-./012345
        0x0060:  3637                                     67
^C
4 packets captured
4 packets received by filter
0 packets dropped by kernel
root@mininet-vm:~/router_experiment# arp -a
? (10.0.1.1) at 40:10:40:10:40:10 [ether] on h1-eth0
root@mininet-vm:~/router_experiment# []
```

```
root@mininet-vm:~/router_experiment# tcpdump -XX -n -i h6-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h6-eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
20:15:07.861251 IP 10.0.1.101 > 10.0.2.100: ICMP echo request, id 25022, seq 1,
length 64
        0x0000:  0000 0000 0006 4010 4010 4010 0800 4500  ......@.@.@...E.
        0x0010:  0054 ec26 4000 4001 36ba 0a00 0165 0a00  .T.&@.@.6....e..
        0x0020:  0264 0800 0257 61be 0001 4b6d f05d 621b  .d...Wa...Km.]b.
        0x0030:  0b00 0809 0a0b 0c0d 0e0f 1011 1213 1415  ................
        0x0040:  1617 1819 1a1b 1c1d 1e1f 2021 2223 2425  ...........!"#$%
        0x0050:  2627 2829 2a2b 2c2d 2e2f 3031 3233 3435  &'()*+,-./012345
        0x0060:  3637                                     67
20:15:07.861366 ARP, Request who-has 10.0.2.1 tell 10.0.2.100, length 28
        0x0000:  ffff ffff ffff 0000 0000 0006 0806 0001  ................
        0x0010:  0800 0604 0001 0000 0000 0006 0a00 0264  ...............d
        0x0020:  0000 0000 0000 0a00 0201                 ..........
20:15:07.904211 ARP, Reply 10.0.2.1 is-at 40:10:40:10:40:10, length 28
        0x0000:  0000 0000 0006 4010 4010 4010 0806 0001  ......@.@.@.....
        0x0010:  0800 0604 0002 4010 4010 4010 0a00 0201  ......@.@.@.....
        0x0020:  0000 0000 0006 0a00 0264                 .........d
20:15:07.904330 IP 10.0.2.100 > 10.0.1.101: ICMP echo reply, id 25022, seq 1, le
ngth 64
        0x0000:  4010 4010 4010 0000 0000 0006 0800 4500  @.@.@.........E.
        0x0010:  0054 cefa 0000 4001 93e6 0a00 0264 0a00  .T....@.......d..
        0x0020:  0165 0000 0a57 61be 0001 4b6d f05d 621b  .e...Wa...Km.]b.
        0x0030:  0b00 0809 0a0b 0c0d 0e0f 1011 1213 1415  ................
        0x0040:  1617 1819 1a1b 1c1d 1e1f 2021 2223 2425  ...........!"#$%
        0x0050:  2627 2829 2a2b 2c2d 2e2f 3031 3233 3435  &'()*+,-./012345
        0x0060:  3637                                     67
^C
4 packets captured
4 packets received by filter
0 packets dropped by kernel
root@mininet-vm:~/router_experiment# arp -a
? (10.0.2.1) at 40:10:40:10:40:10 [ether] on h6-eth0
root@mininet-vm:~/router_experiment# █
```

Figure 3: packets involved in h_1 and h_6 and their ARP caches



```
63 2.511707000 10.0.1.101 10.0.2.100 OF 1.0 182 of_packet_in
  xid: 0
  buffer_id: 257
  total_len: 98
  in_port: 1
  reason: OFPR_NO_MATCH (0)
▽ Ethernet packet
  ▷ Ethernet II, Src: 00:00:00_00:00:01 (00:00:00:00:00:01), Dst: 40:10:40:10:40:10 (40:10:40:10:40:10)
  ▷ Internet Protocol Version 4, Src: 10.0.1.101 (10.0.1.101), Dst: 10.0.2.100 (10.0.2.100)
  ▽ Internet Control Message Protocol
      Type: 8 (Echo (ping) request)
      Code: 0

0040  76 f9 01 0a 00 74 00 00  00 00 00 00 01 01 00 62   v....t.. .......b
0050  00 01 00 00 40 10 40 10  40 10 00 00 00 00 00 01   ....@.@. @.......
0060  08 00 45 00 00 54 ec 26  40 00 40 01 36 ba 0a 00   ..E..T.& @.@.6...
0070  01 65 0a 00 02 64 08 00  02 57 61 be 00 01 4b 6d   .e...d.. .Wa...Km
```

```
64 2.543372000 10.0.1.101 10.0.2.100 OF 1.0 188 of_packet_out
      len: 8
      port: 2
      max_len: 0
▽ Ethernet packet
  ▷ Ethernet II, Src: 40:10:40:10:40:10 (40:10:40:10:40:10), Dst: 00:00:00_00:00:06 (00:00:00:00:00:0
  ▷ Internet Protocol Version 4, Src: 10.0.1.101 (10.0.1.101), Dst: 10.0.2.100 (10.0.2.100)
  ▽ Internet Control Message Protocol
      Type: 8 (Echo (ping) request)
      Code: 0
      Checksum: 0x0257 [correct]
      Identifier (BE): 25022 (0x61be)

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E.
0010  00 ae 60 02 40 00 40 06  dc 45 7f 00 00 01 7f 00   ..`.@.@. .E......
0020  00 01 19 e9 b7 0a 1f 42  f0 2a b6 38 02 5d 80 18   .......B .*.8.]..
0030  00 58 fe a2 00 00 01 01  08 0a 01 ac 77 0e 01 ac   .X...... ....w...
```

Figure 4: controller modified the packet arrived at s_2

Sniffing packets on h_1 and h_6 when the ping starts (Figure 3), we see the ARP request from hosts are successfully handled by the router and its "fake" MAC address is remembered in those hosts' ARP caches. After knowing the gateway's MAC address, the IP packet, namely that single ping, then can pass the router. Using Wireshark to examine the controller (Figure 4), we see the MAC addresses are also successfully updated. Up to this point, we have the working switch and router, and specifications 1 and 2 (in section 2) should be met.

4 Configuration of Data Plane and Priority Queue

As we have discussed in the above section, Network layer consists of the control plane and the data plane (Figure 5). Upon receiving instructions from the controller, the packets are then pushed into the default/specified queue. In the data plane, we can add/remove and configure queues we want, though hopefully the configuration somehow follows the intention of the control plane.

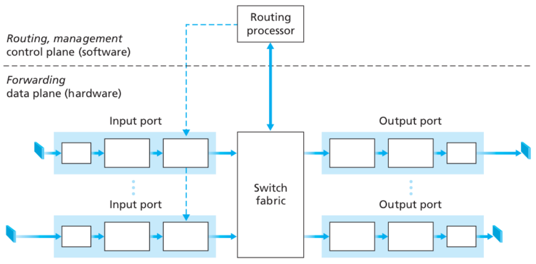
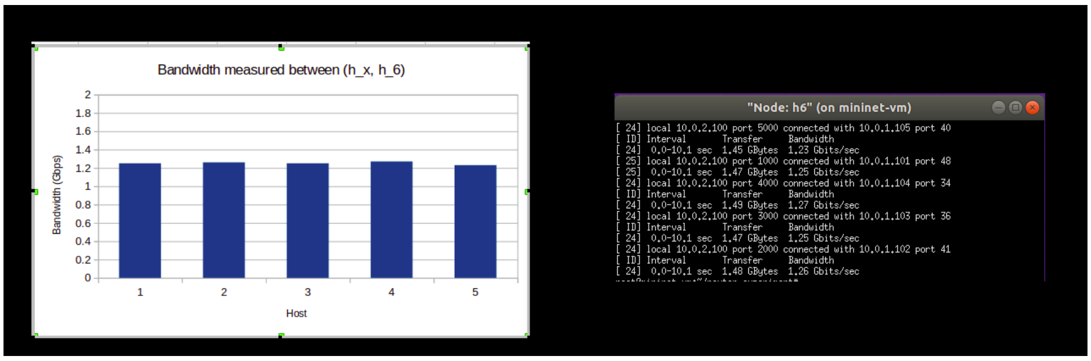


Figure 5: structure of network layer

In section 3, the controller has instructed the router to enqueue any packet from or to 10.0.1.105 to queue 2, and others will go to queue 1. In this manner, the control plane helps the data plane differentiate normal and high-priority packets, and now we will jump into customizing queues 1 and 2 so they will have different performances and the tech specifications in our project can be met.

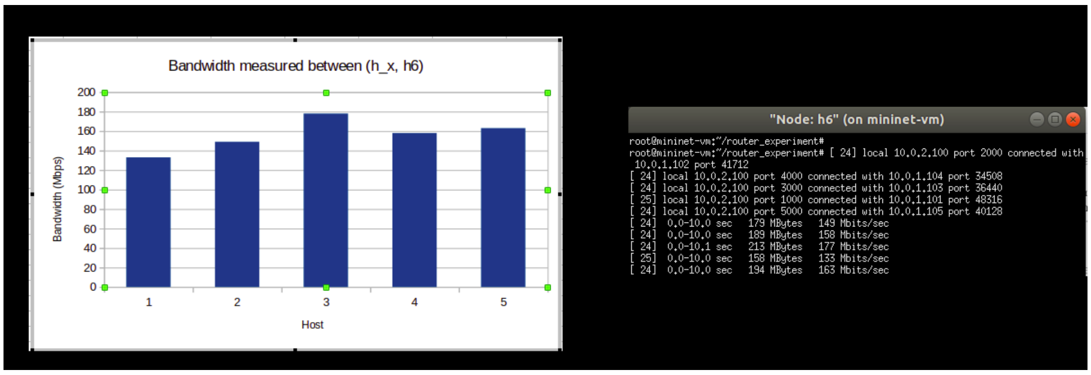
Firstly, we set up a bandwidth limit $2Gbps$ at the entry point of the home LAN. Also, to measure the bandwidth, we use iPerf, which will try to transfer as many data as possible between two ends for a certain length of time and probe the limit. When no configuration of queues has been set up, we measure the bandwidth between pairs $(h_{x\in\{1,2,3,4,5\}}, h_6)$ in various scenarios.

When only 1 host is busy transmitting data with h_6, we have the result shown in Figure 6: every host can utilize most of the available bandwidth (the maximum bandwidth is $2Gbps$ and iPerf uses TCP in this test, which has the well-known additive-increase/multiplicative-decrease (AIMD) feedback control so the performance should be below the ideal). When 5 hosts start data bursting at approximately the same time, we have the result shown in Figure 7. We see that the bandwidth are shared among 5 hosts fairly evenly, and no host gets any "privilege" at the moment. When 3 hosts are idle and only 2 (h_4 and h_5) are competing for the resources, we have the bandwidth of the two hosts shown in Figure 8. Theoretically the two hosts, when no prioritization is presented, should equally divide the bandwidth. However, we see that for 3 out of 4 tests, one host has a significantly higher bandwidth over the other one. Since the transport-layer protocol in these tests are TCP and stream-oriented, the sender tends to easily reach the current bandwidth limit and slow down the transmission. We start the two pairs $((h_4, h_6)$ and $(h_5, h_6))$ of TCP streaming not perfectly at the same time. Therefore, one TCP connection will first take control of must of the resources, and thus for the other one the point where it will experience any congestion is much less than half of the total bandwidth. Then, due to the property of TCP, it will most likely stay in that small range and bounce up and down in a zigzag manner. Because the other connection is not striving for more resources since TCP's congestion control, the dominant one keeps occupying most of the bandwidth. This TCP imbalance becomes less obvious when more hosts are involved in the competition: after any relatively dominant TCP connection halves its Congestion Window and freeing up that amount of bandwidth, a large number of competitors will easily collect the freed resources as more "probings" happen. Using another network measurement tool Netperf which support UDP testing[7], we have the result shown in Figure 9. The bandwidth is divided more more evenly.


Bandwidth measured between (h_x, h_6)
Bandwidth (Gbps)
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
5
Host
"Node: h6" (on mininet-vm)

```
[ 24] local 10.0.2.100 port 5000 connected with 10.0.1.105 port 40
[ ID] Interval       Transfer     Bandwidth
[ 24]  0.0-10.1 sec  1.45 GBytes  1.23 Gbits/sec
[ 25] local 10.0.2.100 port 1000 connected with 10.0.1.101 port 48
[ 25]  0.0-10.1 sec  1.47 GBytes  1.25 Gbits/sec
[ 24] local 10.0.2.100 port 4000 connected with 10.0.1.104 port 34
[ ID] Interval       Transfer     Bandwidth
[ 24]  0.0-10.1 sec  1.49 GBytes  1.27 Gbits/sec
[ 24] local 10.0.2.100 port 3000 connected with 10.0.1.103 port 36
[ ID] Interval       Transfer     Bandwidth
[ 24]  0.0-10.1 sec  1.47 GBytes  1.25 Gbits/sec
[ 24] local 10.0.2.100 port 2000 connected with 10.0.1.102 port 41
[ ID] Interval       Transfer     Bandwidth
[ 24]  0.0-10.1 sec  1.48 GBytes  1.26 Gbits/sec
```

Figure 6: 1 host transmitting data to h_6

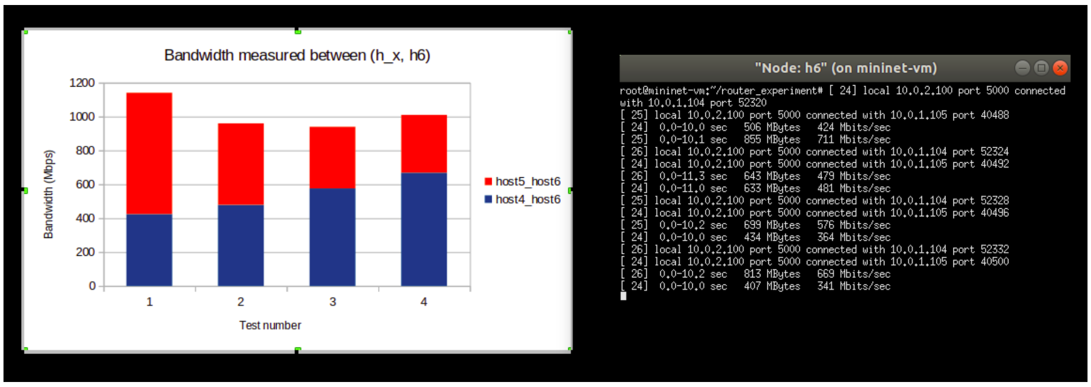

Bandwidth measured between (h_x, h6)
Bandwidth (Mbps)
200
180
160
140
120
100
80
60
40
20
0
1
2
3
4
5
Host
"Node: h6" (on mininet-vm)

```
root@mininet-vm:~/router_experiment#
root@mininet-vm:~/router_experiment# [ 24] local 10.0.2.100 port 2000 connected with
 10.0.1.102 port 41712
[ 24] local 10.0.2.100 port 4000 connected with 10.0.1.104 port 34508
[ 24] local 10.0.2.100 port 3000 connected with 10.0.1.103 port 36440
[ 25] local 10.0.2.100 port 1000 connected with 10.0.1.101 port 48316
[ 24] local 10.0.2.100 port 5000 connected with 10.0.1.105 port 40128
[ 24]  0.0-10.0 sec   179 MBytes   149 Mbits/sec
[ 24]  0.0-10.0 sec   189 MBytes   158 Mbits/sec
[ 24]  0.0-10.1 sec   213 MBytes   177 Mbits/sec
[ 25]  0.0-10.0 sec   158 MBytes   133 Mbits/sec
[ 24]  0.0-10.0 sec   194 MBytes   163 Mbits/sec
```

Figure 7: 5 hosts transmitting data concurrently to h_6

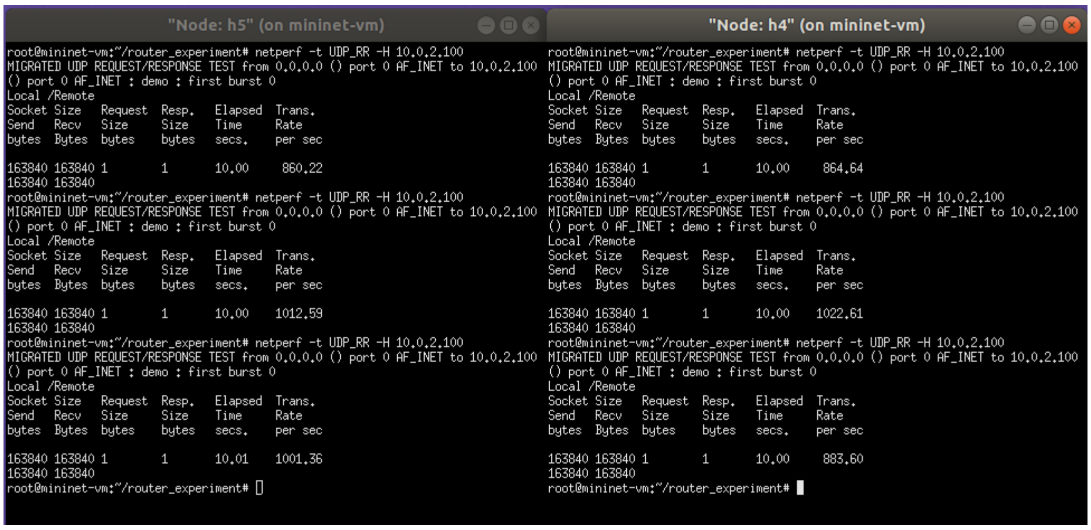
Now, to prioritize packets from h_5, we will set a few rules and configure on router s_2's queue 1 and queue 2. Open vSwitch supports Quality of Service very well[5]. We can shape our router's queues by modifying their max/min rate of packet processing as well as a variable called "priority"[8]. If needed, a high-priority queue will receive all remaining bandwidth (computing resources to process the packets) before that bandwidth can be allocated to any lower-priority queue. In this way, we can not only guarantee our high-priority packets a minimum amount of computing power (for packet processing) inside the router by setting a min rate, but also allows it to use as much available bandwidth as possible (in the queue of the router), which further accelerates the packet processing. Therefore, according to the specifications in section 2, we have set the following rules in the data plane:

1. Queue 1's minimum rate of bandwidth $= 2Gbps$, so the packets will be processed at least as fast as the they can be transmitted on the path (we give a bandwidth limit $2Gbps$ at the entry point



```
root@mininet-vm:~/router_experiment# [ 24] local 10.0.2.100 port 5000 connected
with 10.0.1.104 port 52320
[ 25] local 10.0.2.100 port 5000 connected with 10.0.1.105 port 40488
[ 24]  0.0-10.0 sec   506 MBytes   424 Mbits/sec
[ 25]  0.0-10.1 sec   855 MBytes   711 Mbits/sec
[ 26] local 10.0.2.100 port 5000 connected with 10.0.1.104 port 52324
[ 24] local 10.0.2.100 port 5000 connected with 10.0.1.105 port 40492
[ 26]  0.0-11.3 sec   643 MBytes   479 Mbits/sec
[ 24]  0.0-11.0 sec   633 MBytes   481 Mbits/sec
[ 25] local 10.0.2.100 port 5000 connected with 10.0.1.104 port 52328
[ 24] local 10.0.2.100 port 5000 connected with 10.0.1.105 port 40496
[ 25]  0.0-10.2 sec   699 MBytes   576 Mbits/sec
[ 24]  0.0-10.0 sec   434 MBytes   364 Mbits/sec
[ 26] local 10.0.2.100 port 5000 connected with 10.0.1.104 port 52332
[ 24] local 10.0.2.100 port 5000 connected with 10.0.1.105 port 40500
[ 26]  0.0-10.2 sec   813 MBytes   669 Mbits/sec
[ 24]  0.0-10.0 sec   407 MBytes   341 Mbits/sec
```

Figure 8: 2 hosts transmitting data concurrently to h_6



```
root@mininet-vm:~/router_experiment# netperf -t UDP_RR -H 10.0.2.100
MIGRATED UDP REQUEST/RESPONSE TEST from 0.0.0.0 () port 0 AF_INET to 10.0.2.100
() port 0 AF_INET : demo : first burst 0
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec

163840 163840 1        1       10.00     860.22
163840 163840
root@mininet-vm:~/router_experiment# netperf -t UDP_RR -H 10.0.2.100
MIGRATED UDP REQUEST/RESPONSE TEST from 0.0.0.0 () port 0 AF_INET to 10.0.2.100
() port 0 AF_INET : demo : first burst 0
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec

163840 163840 1        1       10.00    1012.59
163840 163840
root@mininet-vm:~/router_experiment# netperf -t UDP_RR -H 10.0.2.100
MIGRATED UDP REQUEST/RESPONSE TEST from 0.0.0.0 () port 0 AF_INET to 10.0.2.100
() port 0 AF_INET : demo : first burst 0
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec

163840 163840 1        1       10.01    1001.36
163840 163840
root@mininet-vm:~/router_experiment# 
```

```
root@mininet-vm:~/router_experiment# netperf -t UDP_RR -H 10.0.2.100
MIGRATED UDP REQUEST/RESPONSE TEST from 0.0.0.0 () port 0 AF_INET to 10.0.2.100
() port 0 AF_INET : demo : first burst 0
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec

163840 163840 1        1       10.00     864.64
163840 163840
root@mininet-vm:~/router_experiment# netperf -t UDP_RR -H 10.0.2.100
MIGRATED UDP REQUEST/RESPONSE TEST from 0.0.0.0 () port 0 AF_INET to 10.0.2.100
() port 0 AF_INET : demo : first burst 0
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec

163840 163840 1        1       10.00    1022.61
163840 163840
root@mininet-vm:~/router_experiment# netperf -t UDP_RR -H 10.0.2.100
MIGRATED UDP REQUEST/RESPONSE TEST from 0.0.0.0 () port 0 AF_INET to 10.0.2.100
() port 0 AF_INET : demo : first burst 0
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec

163840 163840 1        1       10.00     883.60
163840 163840
root@mininet-vm:~/router_experiment# 
```

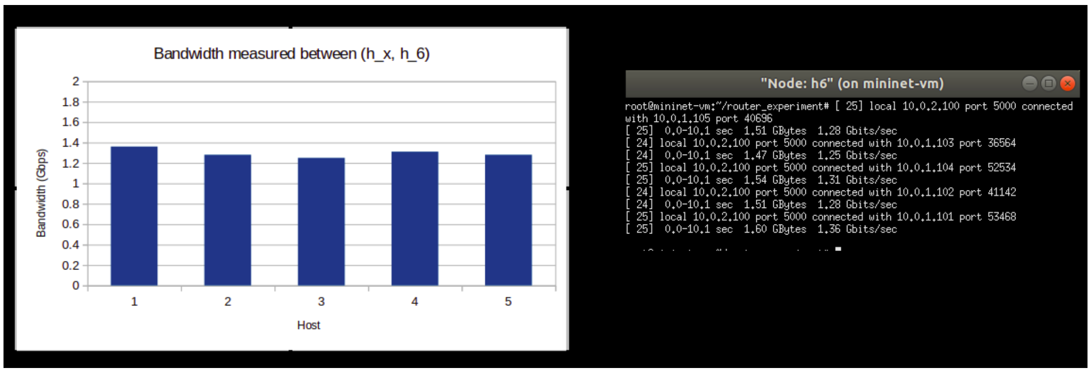
Figure 9: 2 hosts transmitting UDP data concurrently to h_6

of the home LAN).

2. Queue 2's maximum rate of bandwidth $= 2Gbps$, so the packets will not be processed faster than what the link can handle (This will not reduce the maximum bandwidth for hosts h_1, h_2, h_3, h_4. However, if the router's total bandwidth is much more than $2Gbps$, this makes queue 2 less aggressive in terms of taking away unnecessary resources from queue 1).

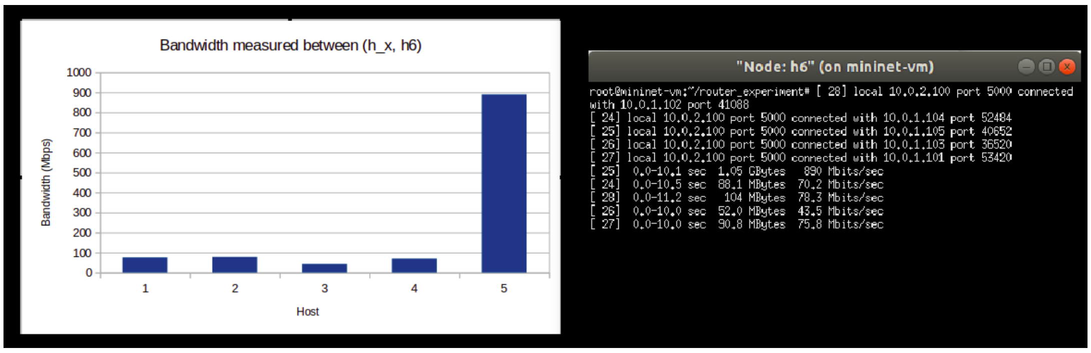
3. Priority of queue 1 $>$ priority of queue 2. When both two queues are assigned some packets, this ensures any extra bandwidth is allocated to queue 1 not queue 2.

Having configured the data plane according to these rules, we do the same tests as when no configuration has been set up. Each host in the home LAN does iPerf connection with content server h_6 sequentially in different time, the result is shown in Figure 10. As what we expect, when only one host is transmitting data, it should be able to use virtually all available bandwidth. When 5 hosts busily transmit data at the same time, as shown in Figure 11, the high-priority host h_5 takes control of most bandwidth, which make senses since with a much more "powerful" and higher-priority queue, packets from h_5 flow therefore much faster. In this way, h_5's TCP connection can easily dominates the others, and even though as mentioned in its mirror test there are many competitors constantly probing freed resources, due to this prioritization and guaranteed high speed of packet processing h_5's TCP connection can easily recapture its discarded resources caused by congestion control from other challengers, if any. Essentially, the major benefit of such priority queue here is to "strengthen" the "TCP dominance". Similarly, shown in Figure 12 when h_5 is competing with only one other host h_4, h_5's privilege becomes even more obvious as there are less attempts of probing to compete for any remaining bandwidth. To this point, we can confirm that this set up for prioritizing our router works and our all specifications have been met in TCP cases.



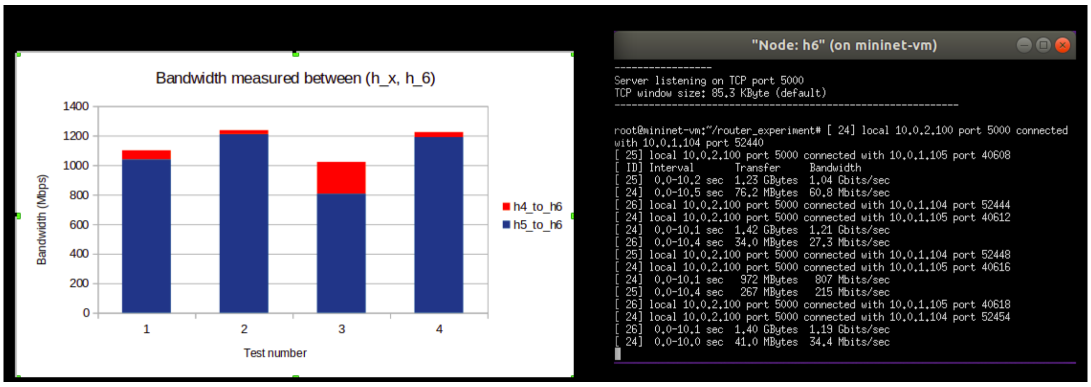
```
"Node: h6" (on mininet-vm)
root@mininet-vm:~/router_experiment# [ 25] local 10.0.2.100 port 5000 connected
with 10.0.1.105 port 40696
[ 25]  0.0-10.1 sec  1.51 GBytes  1.28 Gbits/sec
[ 24] local 10.0.2.100 port 5000 connected with 10.0.1.103 port 36564
[ 24]  0.0-10.1 sec  1.47 GBytes  1.25 Gbits/sec
[ 25] local 10.0.2.100 port 5000 connected with 10.0.1.104 port 52534
[ 25]  0.0-10.1 sec  1.54 GBytes  1.31 Gbits/sec
[ 24] local 10.0.2.100 port 5000 connected with 10.0.1.102 port 41142
[ 24]  0.0-10.1 sec  1.51 GBytes  1.28 Gbits/sec
[ 25] local 10.0.2.100 port 5000 connected with 10.0.1.101 port 53468
[ 25]  0.0-10.1 sec  1.60 GBytes  1.36 Gbits/sec
```

Figure 10: 1 host transmitting data to h_6 after configuration



```
"Node: h6" (on mininet-vm)
root@mininet-vm:~/router_experiment# [ 28] local 10.0.2.100 port 5000 connected
with 10.0.1.102 port 41088
[ 24] local 10.0.2.100 port 5000 connected with 10.0.1.104 port 52484
[ 25] local 10.0.2.100 port 5000 connected with 10.0.1.105 port 40652
[ 26] local 10.0.2.100 port 5000 connected with 10.0.1.103 port 36520
[ 27] local 10.0.2.100 port 5000 connected with 10.0.1.101 port 53420
[ 25]  0.0-10.1 sec  1.05 GBytes   890 Mbits/sec
[ 24]  0.0-10.5 sec  88.1 MBytes  70.2 Mbits/sec
[ 28]  0.0-11.2 sec   104 MBytes  78.3 Mbits/sec
[ 26]  0.0-10.0 sec  52.0 MBytes  43.5 Mbits/sec
[ 27]  0.0-10.0 sec  90.8 MBytes  75.8 Mbits/sec
```

Figure 11: 5 hosts transmitting data concurrently to h_6 after configuration



```
"Node: h6" (on mininet-vm)
----------------
Server listening on TCP port 5000
TCP window size: 85.3 KByte (default)
------------------------------------------------------------

root@mininet-vm:~/router_experiment# [ 24] local 10.0.2.100 port 5000 connected
with 10.0.1.104 port 52440
[ 25] local 10.0.2.100 port 5000 connected with 10.0.1.105 port 40608
[ ID] Interval       Transfer     Bandwidth
[ 25]  0.0-10.2 sec  1.23 GBytes  1.04 Gbits/sec
[ 24]  0.0-10.5 sec  76.2 MBytes  60.8 Mbits/sec
[ 26] local 10.0.2.100 port 5000 connected with 10.0.1.104 port 52444
[ 24] local 10.0.2.100 port 5000 connected with 10.0.1.105 port 40612
[ 24]  0.0-10.1 sec  1.42 GBytes  1.21 Gbits/sec
[ 26]  0.0-10.4 sec  34.0 MBytes  27.3 Mbits/sec
[ 25] local 10.0.2.100 port 5000 connected with 10.0.1.104 port 52448
[ 24] local 10.0.2.100 port 5000 connected with 10.0.1.105 port 40616
[ 24]  0.0-10.1 sec   972 MBytes   807 Mbits/sec
[ 25]  0.0-10.4 sec   267 MBytes   215 Mbits/sec
[ 26] local 10.0.2.100 port 5000 connected with 10.0.1.105 port 40618
[ 24] local 10.0.2.100 port 5000 connected with 10.0.1.104 port 52454
[ 26]  0.0-10.1 sec  1.40 GBytes  1.19 Gbits/sec
[ 24]  0.0-10.0 sec  41.0 MBytes  34.4 Mbits/sec
```

Figure 12: 2 hosts transmitting data concurrently to h_6 after configuration

However, when we try to redo this series of tests using UDP, the pattern restricts itself to roughly:

$$\text{bandwidth of a host} = \frac{\text{total available bandwidth}}{\#\text{ of competing hosts}}$$

.

The prioritization scheme takes the advantage of congestion control of TCP. When we switch to UDP connection, rather than slowing down, senders keep pushing as much data as they can to the routers. As a result, when the altruism of TCP does not exist any more, h_5 loses its privileges. However, in today's internet, most of our traffic belongs to TCP and congestion control is very common. Thus, this router should work correctly and be able to prioritize packet delivery of specified hosts, as most of the services we use today fully (e.g. file transfer, data backup, system update, video streaming) or at least partially (e.g. live streaming, IP phone call) rely on TCP.

5 Conclusion

In this project, we built and tested a SDN-based router that can forward packets and prioritize the packet forwarding for some specific IP address on Mininet. We started with introducing necessary tools and the concepts of control plane as well as data plane. Then, we designed our network topology and created our testing plans. After identifying the tech specifications and detailed goals, we began the creation of our soft router as well as the auxiliary switch by first programming our POX controller in the control plane. Having done that, we verified that both switching devices are functioning well. Then, we configured our router on the data plane so two queues in it can process packets from h_5 and from other hosts in different manners, which therefore will utilize congestion control of TCP to expand the bandwidth of high-priority sender with others inadvertently slowing down their transmissions. We showed results of tests and verified this prioritization scheme. We found out this scheme will has no effect if connections are UDP-based. However, as the usage of TCP is much higher, this SDN design and such prioritization should work in a real-life scenario.

References

[1] Mininet: An Instant Virtual Network on your Laptop (or other PC)
`http://mininet.org/`

[2] Qiang: qiangzheng211/-RouterExercise
`https://github.com/qiangzheng211/-RouterExercise`

[3] Mininet: mininet/openflow-tutorial
`https://github.com/mininet/openflow-tutorial/wiki`

[4] Writers in POX community: POX Wiki
`https://openflow.stanford.edu/display/ONL/POX+Wiki#POXWiki-POXAPIs`

[5] Writers in Open vSwitch: Quality of Service (QoS) Rate Limiting
`http://docs.openvswitch.org/en/latest/howto/qos/#one-physical-network`

[6] Dr. Ke: Lab 5: set traffic to different output queues (QoS issue)
`http://csie.nqu.edu.tw/smallko/sdn/mySDN_Lab5.pdf`

[7] Bloger "wsgzao": A summary about the effectiveness of iPerf and Netperf
`https://wsgzao.github.io/post/netperf/`

[8] Open vSwitch Manual
`http://www.openvswitch.org/support/dist-docs/ovs-vswitchd.conf.db.5.txt`

[9] Wikipedia: EtherType
`https://en.wikipedia.org/wiki/EtherType`