Cloud Mini Project 3

Task I : Submit the custom topology python code

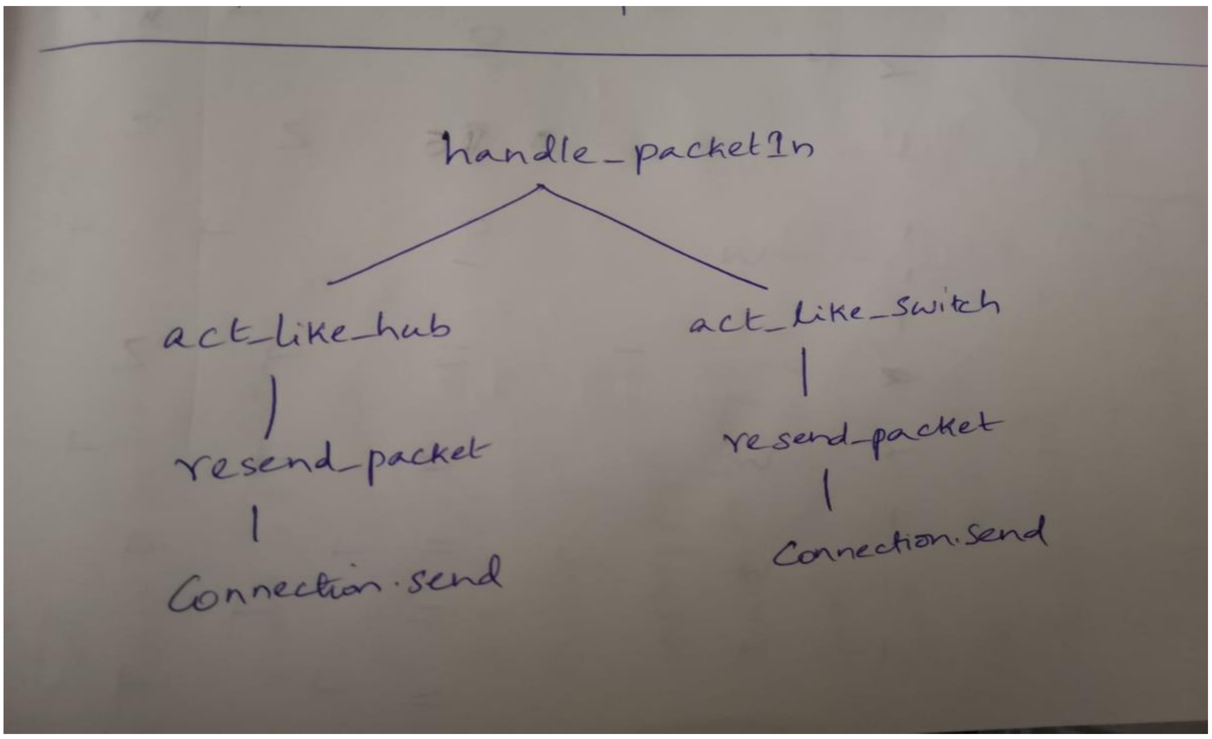
Code is in the tar.gz file

Task II: Study the “of_tutorial” Controller

No code to modify in task 2. Just using the git code provided by Prof.

Q1. Draw the function call graph of this controller. For example, once a packet comes to the controller, which function is the first to be called, which one is the second, and so forth?

Ans: When a packet is received by the controller, the _handle_PacketIn method is called, which in turn calls either act_like_hub or act_like_switch depending on which behavior is desired. Both of these methods may call resend_packet, which then calls connection.send to send the packet back out to the network.



Q2. Have h1 ping h2, and h1 ping h8 for 100 times (e.g., h1 ping -c100 p2). How long does it take (on average) to ping for each case? What is the difference, and why?

Ans:

I executed the below commands as per the question and here are my results

- ➔ h1 ping h2
- --- 10.0.0.2 ping statistics ---
- 402 packets transmitted, 402 received, 0% packet loss, time 401640ms

- rtt min/avg/max/mdev = 1.691/2.432/9.998/0.633 ms

➔ h1 ping -c 100 h8

```
PING 10.0.0.8 (10.0.0.8) 56(84) bytes of data.

--- 10.0.0.8 ping statistics ---

100 packets transmitted, 100 received, 0% packet loss, time 99157ms

rtt min/avg/max/mdev = 6.836/9.070/18.008/1.283 ms
```

Analysis:

Based on the output I received, here are my observations.

1. The number of packets transmitted and received is different: in the first ping from h1 to h2, 402 packets were transmitted and received, whereas in the second ping from h1 to h8, only 100 packets were transmitted and received.
2. The RTT values are different: in the ping from h1 to h2, the minimum RTT was 1.691 ms, the average RTT was 2.432 ms, the maximum RTT was 9.998 ms, and the standard deviation (mdev) was 0.633 ms. In the ping from h1 to h8, the minimum RTT was 6.836 ms, the average RTT was 9.070 ms, the maximum RTT was 18.008 ms, and the standard deviation (mdev) was 1.283 ms.

The main reason for these differences is likely due to the distance and network conditions between h1 and the two destinations (h2 and h8). The lower RTT values and higher number of packets transmitted and received in the ping from h1 to h2 suggests that the two hosts are likely located closer together and/or connected by a faster and more reliable network link, whereas the higher RTT values and lower number of packets transmitted and received in the ping from h1 to h8 suggests that h1 and h8 are further apart and/or connected by a slower and less reliable network link.

Q.3 Run "iperf h1 h2" and "iperf h1 h8". What is "iperf" used for? What is the throughput for each case? What is the difference, and why?

Ans:

iperf is a tool used to measure network performance by generating TCP and UDP data streams and measuring the throughput, packet loss, and latency of the network. It can be used to test the maximum achievable bandwidth between two network endpoints, and to diagnose network congestion issues. iperf can also be used for testing the quality of network links and detecting the presence of firewalls, routers, or other devices that may be blocking or filtering network traffic.

Below are my results when I ran the commands in my VM:

iperf h1 h2

```
*** Iperf: testing TCP bandwidth between h1 and h2

*** Results: ['7.04 Mbits/sec', '8.25 Mbits/sec']

iperf h1 h8

*** Iperf: testing TCP bandwidth between h1 and h8

*** Results: ['1.90 Mbits/sec', '2.21 Mbits/sec']
```

The throughput for h1 to h2 is 7.04 Mbits/sec and 8.25 Mbits/sec, while the throughput for h1 to h8 is 1.90 Mbits/sec and 2.21 Mbits/sec.
The difference in throughput between the two cases is due to the difference in the network topology and the distance between the hosts. Since h2 is directly connected to h1 in the same network, the throughput between them is higher than the throughput between h1 and h8, which requires the data to traverse multiple switches and links in the network. This introduces more latency and can cause congestion and packet loss, which can reduce the throughput.

Q.4 Which of the switches observe traffic? Please describe your way for observing such traffic on switches (hint: adding some "print" functions in the “of_tutorial” controller).

In this topology, all switches are observing traffic since they are all involved in connecting the hosts together. However, the switches that are most likely to observe the most traffic are the ones that are closest to the centre of the topology, i.e., s5 and s6, because they connect the other switches together. Since this is a simple hub-like switch that floods incoming packets out to all ports except the incoming port. Therefore, all switches in the topology observes traffic, as all incoming packets would be flooded out to all ports.

Task III: MAC learning controller

Code is in the tar.gz file

Q.1 Please describe how the above code works, such as how the "MAC to Port" map is established. You could use a ‘ping’ example to describe the establishment process (e.g., h1 ping h2).

Ans:

After adding the act_like_switch function to the given _of_tutorial.py code, it will become a code which is a simple OpenFlow controller that can act either as a hub or as a learning switch. When a new OpenFlow switch connects to the controller, a new instance of the Tutorial class is created.
The Tutorial class has three methods: __init__(), _handle_PacketIn(), and act_like_hub(), and act_like_switch(). The __init__() method initializes the instance of the class, and sets up an event listener for packet-in messages from the switch. The _handle_PacketIn() method handles packet-in messages from the switch, and the act_like_hub() and act_like_switch() methods implement the hub-like behavior and the learning switch behavior, respectively.

The mac_to_port dictionary is used to keep track of which MAC address is associated with which switch port. When a packet is received by the switch and sent to the controller due to a table miss, the controller will first learn the port associated with the source MAC of the packet by storing the source MAC address and the input port number in the mac_to_port dictionary. Then, if the destination MAC address of the packet is in the mac_to_port dictionary, the controller will send the packet out the port associated with the destination MAC. If the destination MAC is not known, the controller will flood the packet out all ports except the input port.

For example, lets consider h1—s1—h2

Two hosts connected to S1.

When h1 sends a ping request to h2, the packet will arrive at the switch s1. Since s1 has never seen h1 before, it sends a packet-in message to the controller, which is handled by the _handle_PacketIn() method. The act_like_switch() method is called, and since the mac_to_port dictionary is empty, it stores the source MAC address (h1) and the input port (1) in the mac_to_port dictionary. Then, since the destination MAC address (h2) is not known, it floods the packet out all ports except the input port (1).
When the packet arrives at h2, h2 responds with a ping reply. This time, s1 has already learned the port associated with h1, so it sends the packet only out the port connected to h2, which is port 2. The mac_to_port dictionary is updated with the destination MAC address (h1) and the output port (1).

From this point on, all traffic between h1 and h2 will be forwarded by the switch without involving the controller, because the switch has already learned which MAC address is associated with which port.

Similarly if I Ping h1 to h2, it works as below:

- ➔ h1 sends an ICMP echo request (ping) to h2. The Ethernet frame will have the source MAC address of h1 and the destination MAC address of h2.
- ➔ The switch s1 receives the Ethernet frame and does not have an entry in its "MAC to Port" map for the source MAC address of h1. Thus, it floods the Ethernet frame to all of its ports, including the port that connects to s2.
- ➔ The switch s2 receives the flooded Ethernet frame and adds an entry to its "MAC to Port" map that maps the source MAC address of h1 to the port that received the frame. Then, s2 looks up the destination MAC address of h2 in its "MAC to Port" map and finds that it is connected to the port that connects to h2.
- ➔ s2 forwards the Ethernet frame only to the port that connects to h2.
- ➔ h2 receives the ICMP echo request and responds with an ICMP echo reply. The Ethernet frame will have the source MAC address of h2 and the destination MAC address of h1.
- ➔ The switch s2 receives the Ethernet frame, looks up the destination MAC address of h1 in its "MAC to Port" map, and finds that it is connected to the port that received the original ICMP echo request.
- ➔ s2 forwards the Ethernet frame only to the port that connects to s1.
- ➔ The switch s1 receives the Ethernet frame, looks up the destination MAC address of h1 in its "MAC to Port" map, and finds that it is connected to the port that received the original ICMP echo request.

- ➔ s1 forwards the Ethernet frame only to the port that connects to h1.
- ➔ h1 receives the ICMP echo reply, and the ping completes.

Throughout this process, the "MAC to Port" maps are populated by the switches as they learn the locations of the hosts based on the MAC addresses in the Ethernet frames.

Q.2 (Please disable your output functions, i.e., print, before doing this experiment) Have h1 ping h2, and h1 ping h8 for 100 times (e.g., h1 ping -c100 p2). How long did it take (on average) to ping for each case? Any difference from Task II (the hub case)?

Ans:

H1 ping h2

```
10.0.0.2 ping statistics ---

55 packets transmitted, 55 received, 0% packet loss, time 54090ms

rtt min/avg/max/mdev = 2.107/3.269/10.235/1.117 ms
```

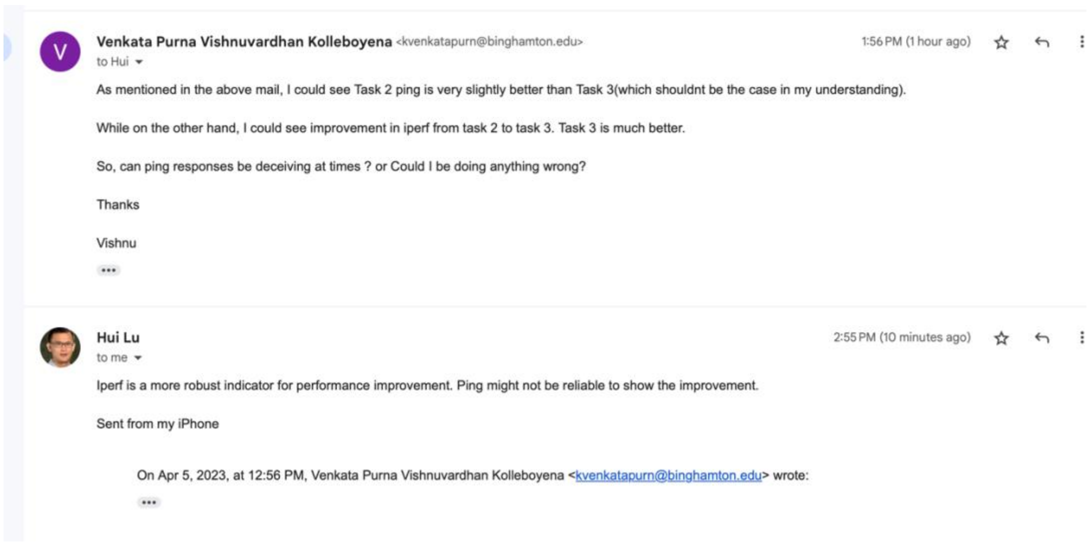
H1 ping -c 100 h8

```
--- 10.0.0.8 ping statistics ---

100 packets transmitted, 100 received, 0% packet loss, time 99168ms

rtt min/avg/max/mdev = 8.652/11.245/24.253/2.038 ms
```

Ping response for h1 ping h8 & h1 ping -c 100 h8 is not very stable. Sometimes it is better than Task 2. Sometimes it’s worse than Task 2.

Compared to Task 2, not much of difference can be seen but we cannot rely on ping alone for concluding performance improvement. In Q3, **iperf has seen a significant improvement** from Task 2 to Task 3, which **is robust indicator of improvement.**

I got confirmation for the same regarding this from our Prof as well.


Venkata Purna Vishnuvardhan Kolleboyena <kvenkatapurn@binghamton.edu>
1:56 PM (1 hour ago)
to Hui

As mentioned in the above mail, I could see Task 2 ping is very slightly better than Task 3(which shouldnt be the case in my understanding).

While on the other hand, I could see improvement in iperf from task 2 to task 3. Task 3 is much better.

So, can ping responses be deceiving at times ? or Could I be doing anything wrong?

Thanks

Vishnu

Hui Lu
2:55 PM (10 minutes ago)
to me

Iperf is a more robust indicator for performance improvement. Ping might not be reliable to show the improvement.

Sent from my iPhone

On Apr 5, 2023, at 12:56 PM, Venkata Purna Vishnuvardhan Kolleboyena <kvenkatapurn@binghamton.edu> wrote:

Q.3 Run "iperf h1 h2" and "iperf h1 h8". What is the throughput for each case? What is the difference from Task II?

Ans:

I could see improvement of throughput from Task 2 to Task 3.

Task 3 throughput and bandwidth is much better.

Task 2:

```
iperf h1 h2
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['7.04 Mbits/sec', '8.25 Mbits/sec']
iperf h1 h8
*** Iperf: testing TCP bandwidth between h1 and h8
*** Results: ['1.90 Mbits/sec', '2.21 Mbits/sec']
```

Task 3:

```
iperf h1 h2
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['13.4 Mbits/sec', '15.3 Mbits/sec']

iperf h1 h8
*** Iperf: testing TCP bandwidth between h1 and h8
*** Results: ['2.64 Mbits/sec', '3.28 Mbits/sec']
```

Task IV: MAC learning controller with OpenFlow rules

Code is in the tar.gz file

Q.1 Have h1 ping h2, and h1 ping h8 for 100 times (e.g., h1 ping -c100 p2). How long does it take (on average) to ping for each case? Any difference from Task III (the MAC case without inserting flow rules)?

Ans: Tremendous improvement can be seen from task 3 here.

When it was a hub(in Task2), ping time was worse. When it was a switch(in Task3), its mostly the same. But when we made switch work with open flow rules(Task4), the performance boost can be clearly visible.

H1 ping h2

65 packets transmitted, 65 received, 0% packet loss, time 65501ms

rtt min/avg/max/mdev = 0.053/0.109/2.458/0.294 ms

H1 ping -c 100 h8

100 packets transmitted, 100 received, 0% packet loss, time 101360ms

rtt min/avg/max/mdev = 0.068/0.167/7.258/0.712 ms

Q.2 Run "iperf h1 h2" and "iperf h1 h8". What is the throughput for each case? What is the difference from Task III?

Ans: After adding the open flow switch rules, the improvement is just outstanding. There is a huge jump in the performance.

Iperf h1 h2

*** Iperf: testing TCP bandwidth between h1 and h2

*** Results: ['23.2 Gbits/sec', '23.2 Gbits/sec']

Iperf h1 h8

*** Iperf: testing TCP bandwidth between h1 and h8

*** Results: ['18.6 Gbits/sec', '18.7 Gbits/sec']

When compared to task 3, we can see huge difference in the numbers in task 4.

Q.3 Please explain the above results — why the results become better or worse?

Performance : Task 4> Task 3> Task 2

It has become much better with each and every task.

The reason for that is,

In Task 2: We have made it act like an hub. As we know it very well, a hub is gonna flood all the hosts with packets, which will obviously result in poor performance.

In Task 3: We made the controller work like an switch. That's like replacing a hub with an switch, there is a boost in the performance, because, a switch is going to send the packets to the destination host alone and it doesn't flood the network. It remembers the MAC address of the host & when some packet comes again and asks to be sent to the same MAC, it does the job almost immediately without sending the request to controller. This saves us time. Hence , we can see improvement in Task 3 when compared to Task 2.

In Task 4: On top of the switch, we made the controller work with open flow rules, that means, it installs flow rules on switch to handle the future packets as well. Thus, the only packets that should arrive at the controller are those that the switch doesn't have a flow entry for. This is like making the switch even more smarter. OpenFlow separates the control plane from the data plane, which allows for centralized network management and simplified forwarding of traffic. This simplification leads to a reduction in processing time and improved throughput.That's exactly why, we can see a huge improvement in Task 4.

Q.4 Run pingall to verify connectivity and dump the output.

Ans:

```
pingall

*** Ping: testing ping reachability

h1 -> h2 h3 h4 h5 h6 h7 h8

h2 -> h1 h3 h4 h5 h6 h7 h8

h3 -> h1 h2 h4 h5 h6 h7 h8

h4 -> h1 h2 h3 h5 h6 h7 h8

h5 -> h1 h2 h3 h4 h6 h7 h8

h6 -> h1 h2 h3 h4 h5 h7 h8

h7 -> h1 h2 h3 h4 h5 h6 h8

h8 -> h1 h2 h3 h4 h5 h6 h7

*** Results: 0% dropped (56/56 received)
```

Q.5 Dump the output of the flow rules using "ovs-ofctl dump-flows" (in your container, not mininet). How many rules are there for each OpenFlow switch, and why? What does each flow entry mean (select one flow entry and explain)?

Ans: **Here is the output for each OpenFlow switch & at the end I explained using switch 7 dump.**

root@007e40094b99:~# ovs-ofctl dump-flows s1

```
 cookie=0x0, duration=82.302s, table=0, n_packets=0, n_bytes=0, dl_dst=1a:59:60:35:46:b1 actions=output:"s1-eth3"

 cookie=0x0, duration=82.255s, table=0, n_packets=0, n_bytes=0, dl_dst=5e:36:25:24:cd:2e actions=output:"s1-eth2"

 cookie=0x0, duration=82.160s, table=0, n_packets=0, n_bytes=0, dl_dst=d6:2a:6d:95:c2:ee actions=output:"s1-eth1"

 cookie=0x0, duration=82.053s, table=0, n_packets=0, n_bytes=0, dl_dst=4e:4e:a7:ea:5b:cb actions=output:"s1-eth3"

 cookie=0x0, duration=81.925s, table=0, n_packets=0, n_bytes=0, dl_dst=de:9e:ce:b3:22:c9 actions=output:"s1-eth3"

 cookie=0x0, duration=81.863s, table=0, n_packets=0, n_bytes=0, dl_dst=e6:ab:cc:81:60:d7 actions=output:"s1-eth3"

 cookie=0x0, duration=81.801s, table=0, n_packets=0, n_bytes=0, dl_dst=ea:05:cb:6e:d8:5c actions=output:"s1-eth3"

 cookie=0x0, duration=81.632s, table=0, n_packets=0, n_bytes=0, dl_dst=62:33:c8:94:78:a3 actions=output:"s1-eth3"
```

root@007e40094b99:~# ovs-ofctl dump-flows s2

```
 cookie=0x0, duration=85.865s, table=0, n_packets=0, n_bytes=0, dl_dst=1a:59:60:35:46:b1 actions=output:"s2-eth3"

 cookie=0x0, duration=85.811s, table=0, n_packets=0, n_bytes=0, dl_dst=5e:36:25:24:cd:2e actions=output:"s2-eth3"

 cookie=0x0, duration=85.713s, table=0, n_packets=0, n_bytes=0, dl_dst=d6:2a:6d:95:c2:ee actions=output:"s2-eth3"

 cookie=0x0, duration=85.614s, table=0, n_packets=0, n_bytes=0, dl_dst=4e:4e:a7:ea:5b:cb actions=output:"s2-eth3"

 cookie=0x0, duration=85.485s, table=0, n_packets=0, n_bytes=0, dl_dst=de:9e:ce:b3:22:c9 actions=output:"s2-eth3"

 cookie=0x0, duration=85.435s, table=0, n_packets=0, n_bytes=0, dl_dst=e6:ab:cc:81:60:d7 actions=output:"s2-eth1"

 cookie=0x0, duration=85.370s, table=0, n_packets=0, n_bytes=0, dl_dst=ea:05:cb:6e:d8:5c actions=output:"s2-eth2"
 cookie=0x0, duration=85.193s, table=0, n_packets=0, n_bytes=0, dl_dst=62:33:c8:94:78:a3 actions=output:"s2-eth3"
```

root@007e40094b99:~# ovs-ofctl dump-flows s3

```
 cookie=0x0, duration=92.082s, table=0, n_packets=0, n_bytes=0, dl_dst=1a:59:60:35:46:b1 actions=output:"s3-eth1"

 cookie=0x0, duration=92.008s, table=0, n_packets=0, n_bytes=0, dl_dst=5e:36:25:24:cd:2e actions=output:"s3-eth3"

 cookie=0x0, duration=91.911s, table=0, n_packets=0, n_bytes=0, dl_dst=d6:2a:6d:95:c2:ee actions=output:"s3-eth3"

 cookie=0x0, duration=91.830s, table=0, n_packets=0, n_bytes=0, dl_dst=4e:4e:a7:ea:5b:cb actions=output:"s3-eth2"

 cookie=0x0, duration=91.694s, table=0, n_packets=0, n_bytes=0, dl_dst=de:9e:ce:b3:22:c9 actions=output:"s3-eth3"

 cookie=0x0, duration=91.627s, table=0, n_packets=0, n_bytes=0, dl_dst=e6:ab:cc:81:60:d7 actions=output:"s3-eth3"

 cookie=0x0, duration=91.562s, table=0, n_packets=0, n_bytes=0, dl_dst=ea:05:cb:6e:d8:5c actions=output:"s3-eth3"

 cookie=0x0, duration=91.406s, table=0, n_packets=0, n_bytes=0, dl_dst=62:33:c8:94:78:a3 actions=output:"s3-eth3"
```

root@007e40094b99:~# ovs-ofctl dump-flows s4

```
 cookie=0x0, duration=96.118s, table=0, n_packets=0, n_bytes=0, dl_dst=1a:59:60:35:46:b1 actions=output:"s4-eth3"

 cookie=0x0, duration=96.054s, table=0, n_packets=0, n_bytes=0, dl_dst=5e:36:25:24:cd:2e actions=output:"s4-eth3"

 cookie=0x0, duration=95.957s, table=0, n_packets=0, n_bytes=0, dl_dst=d6:2a:6d:95:c2:ee actions=output:"s4-eth3"

 cookie=0x0, duration=95.867s, table=0, n_packets=0, n_bytes=0, dl_dst=4e:4e:a7:ea:5b:cb actions=output:"s4-eth3"

 cookie=0x0, duration=95.746s, table=0, n_packets=0, n_bytes=0, dl_dst=de:9e:ce:b3:22:c9 actions=output:"s4-eth1"

 cookie=0x0, duration=95.672s, table=0, n_packets=0, n_bytes=0, dl_dst=e6:ab:cc:81:60:d7 actions=output:"s4-eth3"

 cookie=0x0, duration=95.607s, table=0, n_packets=0, n_bytes=0, dl_dst=ea:05:cb:6e:d8:5c actions=output:"s4-eth3"
 cookie=0x0, duration=95.459s, table=0, n_packets=0, n_bytes=0, dl_dst=62:33:c8:94:78:a3 actions=output:"s4-eth2"
```

```
root@007e40094b99:~# ovs-ofctl dump-flows s5

 cookie=0x0, duration=98.841s, table=0, n_packets=0, n_bytes=0, dl_dst=1a:59:60:35:46:b1 actions=output:"s5-eth3"

 cookie=0x0, duration=98.788s, table=0, n_packets=0, n_bytes=0, dl_dst=5e:36:25:24:cd:2e actions=output:"s5-eth1"

 cookie=0x0, duration=98.692s, table=0, n_packets=0, n_bytes=0, dl_dst=d6:2a:6d:95:c2:ee actions=output:"s5-eth1"

 cookie=0x0, duration=98.591s, table=0, n_packets=0, n_bytes=0, dl_dst=4e:4e:a7:ea:5b:cb actions=output:"s5-eth3"

 cookie=0x0, duration=98.462s, table=0, n_packets=0, n_bytes=0, dl_dst=de:9e:ce:b3:22:c9 actions=output:"s5-eth3"

 cookie=0x0, duration=98.404s, table=0, n_packets=0, n_bytes=0, dl_dst=e6:ab:cc:81:60:d7 actions=output:"s5-eth2"

 cookie=0x0, duration=98.341s, table=0, n_packets=0, n_bytes=0, dl_dst=ea:05:cb:6e:d8:5c actions=output:"s5-eth2"

 cookie=0x0, duration=98.171s, table=0, n_packets=0, n_bytes=0, dl_dst=62:33:c8:94:78:a3 actions=output:"s5-eth3"
```

```
root@007e40094b99:~# ovs-ofctl dump-flows s6

 cookie=0x0, duration=105.607s, table=0, n_packets=0, n_bytes=0, dl_dst=1a:59:60:35:46:b1 actions=output:"s6-eth1"

 cookie=0x0, duration=105.541s, table=0, n_packets=0, n_bytes=0, dl_dst=5e:36:25:24:cd:2e actions=output:"s6-eth3"

 cookie=0x0, duration=105.443s, table=0, n_packets=0, n_bytes=0, dl_dst=d6:2a:6d:95:c2:ee actions=output:"s6-eth3"

 cookie=0x0, duration=105.355s, table=0, n_packets=0, n_bytes=0, dl_dst=4e:4e:a7:ea:5b:cb actions=output:"s6-eth1"

 cookie=0x0, duration=105.227s, table=0, n_packets=0, n_bytes=0, dl_dst=de:9e:ce:b3:22:c9 actions=output:"s6-eth2"

 cookie=0x0, duration=105.159s, table=0, n_packets=0, n_bytes=0, dl_dst=e6:ab:cc:81:60:d7 actions=output:"s6-eth3"

 cookie=0x0, duration=105.094s, table=0, n_packets=0, n_bytes=0, dl_dst=ea:05:cb:6e:d8:5c actions=output:"s6-eth3"
 cookie=0x0, duration=104.939s, table=0, n_packets=0, n_bytes=0, dl_dst=62:33:c8:94:78:a3 actions=output:"s6-eth2"
```

root@007e40094b99:~# ovs-ofctl dump-flows s7

```
 cookie=0x0, duration=109.550s, table=0, n_packets=0, n_bytes=0, dl_dst=1a:59:60:35:46:b1 actions=output:"s7-eth2"

 cookie=0x0, duration=109.490s, table=0, n_packets=0, n_bytes=0, dl_dst=5e:36:25:24:cd:2e actions=output:"s7-eth1"

 cookie=0x0, duration=109.393s, table=0, n_packets=0, n_bytes=0, dl_dst=d6:2a:6d:95:c2:ee actions=output:"s7-eth1"

 cookie=0x0, duration=109.298s, table=0, n_packets=0, n_bytes=0, dl_dst=4e:4e:a7:ea:5b:cb actions=output:"s7-eth2"

 cookie=0x0, duration=109.170s, table=0, n_packets=0, n_bytes=0, dl_dst=de:9e:ce:b3:22:c9 actions=output:"s7-eth2"

 cookie=0x0, duration=109.107s, table=0, n_packets=0, n_bytes=0, dl_dst=e6:ab:cc:81:60:d7 actions=output:"s7-eth1"

 cookie=0x0, duration=109.043s, table=0, n_packets=0, n_bytes=0, dl_dst=ea:05:cb:6e:d8:5c actions=output:"s7-eth1"

 cookie=0x0, duration=108.882s, table=0, n_packets=0, n_bytes=0, dl_dst=62:33:c8:94:78:a3 actions=output:"s7-eth2"

root@007e40094b99:~#
```

Analysis: Considering switch 7 dump

If we look at the above output for the command, there are 8 flow rules for Switch 7. Each flow entry consists of multiple fields separated by commas, including:

1. cookie: a unique identifier for the flow rule
2. duration: how long the flow rule has been active
3. table: which OpenFlow table the flow rule is in
4. n_packets: how many packets have matched this flow rule
5. n_bytes: how many bytes have been transferred for packets that match this flow rule
6. dl_dst: the destination MAC address for the flow rule
7. actions: what action(s) to take for packets that match this flow rule

For example, let's take the first flow entry as an example:

```
cookie=0x0, duration=109.550s, table=0, n_packets=0, n_bytes=0, dl_dst=1a:59:60:35:46:b1 actions=output:"s7-eth2"
```

This flow rule has been active for 109.550 seconds, is in table 0, and has not matched any packets yet (n_packets and n_bytes are both 0). It matches packets with a destination MAC address of 1a:59:60:35:46:b1 and specifies that those packets should be output to the s7-eth2 interface.

Task V: Layer-3 routing

The task will allow us to experience a bit of layer-3 routing. This is a simplified version of an IP router. Wecanuse"h1~h8 ifconfig" to find out IPaddressesofh1~h8arefrom10.0.0.1to10.0.0.8. Install IP-matching rules on switch 7, but let other switches stay as MAC learning switches (same as Section 3). The goal is that all hosts from h1 to h4 can ping h5 to h8, and switch 7 forwards packets via IP-matching flow rules (not MAC-matching rules). You can install IP-matching rules in a controller or using ovs-ofctl commands (hint: figure it out how to use this command).

Task V: Submit the controller code, or describe the method to set up the IP-matching rules.

Ans: Instead of modifying the controller code, I have explored few ovs-ofctl commands and I will describe the method to set up the IP-matching rules for this task.

- ➔ Once the topology is up and running, we can use below command in mininet to open console for switch s7.

  ```
  mnexec -a s7 bash
  ```

- ➔ Now we will get Inside the console for switch 7, and then we will use the following commands to install an IP-matching rule that forwards packets to the appropriate host based on the destination IP address:

```
ovs-ofctl add-flow s7 priority=100,dl_type=0x0800,nw_dst=10.0.0.1,actions=output:1

ovs-ofctl add-flow s7 priority=100,dl_type=0x0800,nw_dst=10.0.0.2,actions=output:2

ovs-ofctl add-flow s7 priority=100,dl_type=0x0800,nw_dst=10.0.0.3,actions=output:3

ovs-ofctl add-flow s7 priority=100,dl_type=0x0800,nw_dst=10.0.0.4,actions=output:4

ovs-ofctl add-flow s7 priority=100,dl_type=0x0800,nw_dst=10.0.0.5,actions=output:5

ovs-ofctl add-flow s7 priority=100,dl_type=0x0800,nw_dst=10.0.0.6,actions=output:6

 ovs-ofctl add-flow s7 priority=100,dl_type=0x0800,nw_dst=10.0.0.7,actions=output:7

 ovs-ofctl add-flow s7 priority=100,dl_type=0x0800,nw_dst=10.0.0.8,actions=output:8
```

- ➔ This will install eight IP-matching rules that forward packets to the appropriate hosts based on their IP addresses. Now we can test the connection using ping command just like how we are doing in task 2, task 3,task 4.
- ➔ By following the above steps, we should be able to install IP-matching rules on switch 7 and route traffic based on IP addresses.