Cloud Computing HW3

Kiran Kodali(1650443)

Task 1: Defining custom topologies

1. Output of nodes and net
    a. Nodes

```
mininet> nodes
available nodes are:
c0 h1 h2 h3 h4 h5 h6 h7 h8 s1 s2 s3 s4 s5 s6 s7
mininet>
```

```
mininet> net
h1 h1-eth0:s3-eth1
h2 h2-eth0:s3-eth2
h3 h3-eth0:s4-eth1
h4 h4-eth0:s4-eth2
h5 h5-eth0:s6-eth1
h6 h6-eth0:s6-eth2
h7 h7-eth0:s7-eth1
h8 h8-eth0:s7-eth2
s1 lo:  s1-eth1:s2-eth3 s1-eth2:s5-eth3
s2 lo:  s2-eth1:s3-eth3 s2-eth2:s4-eth3 s2-eth3:s1-eth1
s3 lo:  s3-eth1:h1-eth0 s3-eth2:h2-eth0 s3-eth3:s2-eth1
s4 lo:  s4-eth1:h3-eth0 s4-eth2:h4-eth0 s4-eth3:s2-eth2
s5 lo:  s5-eth1:s6-eth3 s5-eth2:s7-eth3 s5-eth3:s1-eth2
s6 lo:  s6-eth1:h5-eth0 s6-eth2:h6-eth0 s6-eth3:s5-eth1
s7 lo:  s7-eth1:h7-eth0 s7-eth2:h8-eth0 s7-eth3:s5-eth2
c0
mininet>
```

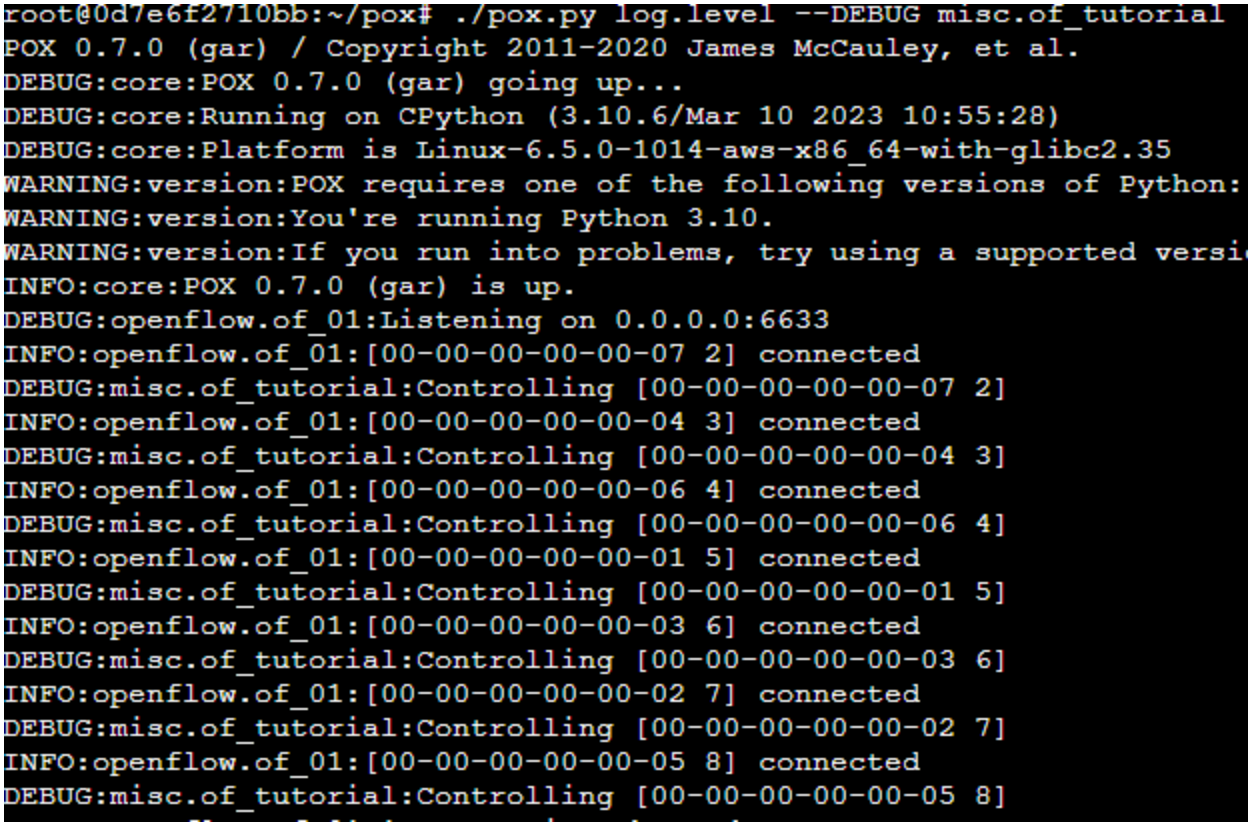
2. Output of h7 ifconfig

```
mininet> h7 ifconfig
h7-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.7  netmask 255.0.0.0  broadcast 10.255.255.255
        inet6 fe80::a41c:66ff:fe86:1603  prefixlen 64  scopeid 0x20<link>
        ether a6:1c:66:86:16:03  txqueuelen 1000  (Ethernet)
        RX packets 75  bytes 5702 (5.7 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 12  bytes 936 (936.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

mininet>
```

Task 2: Analyze the “of_tutorial’ controller

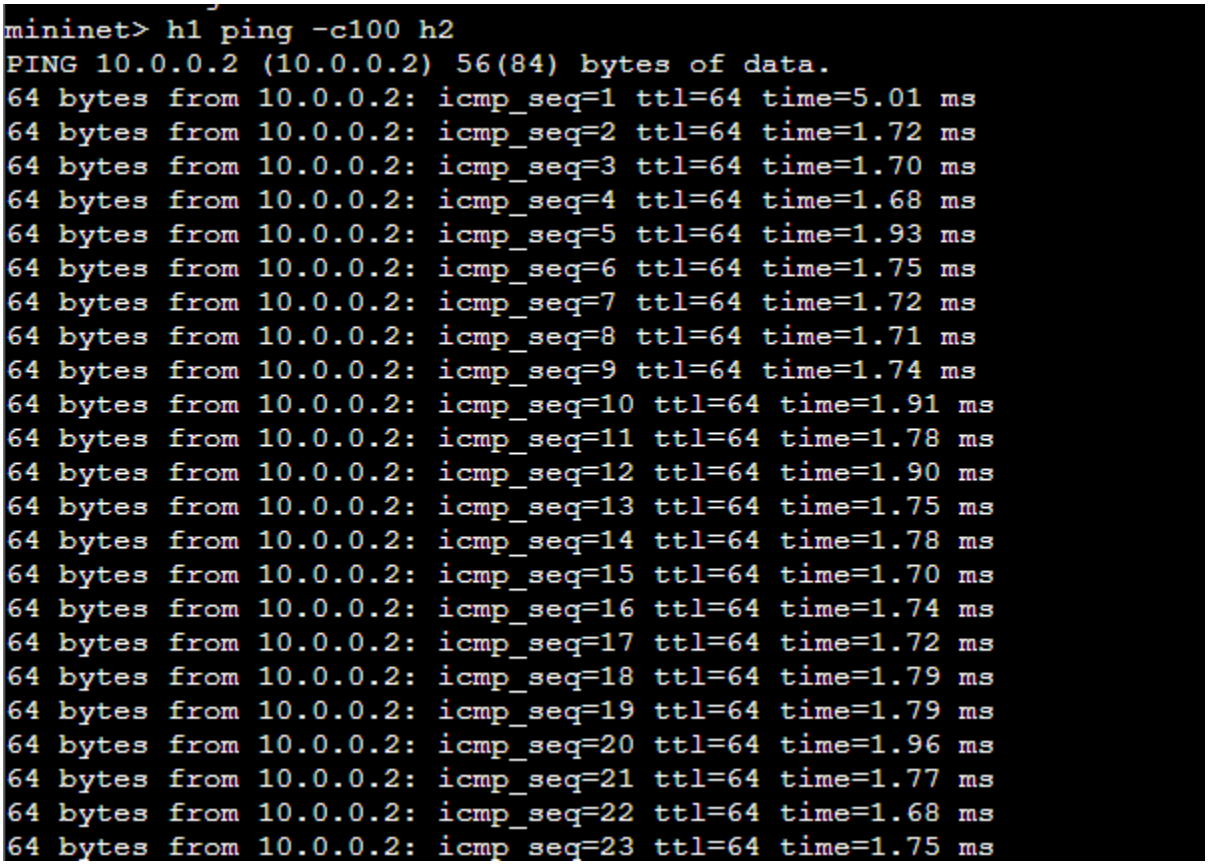
1. Draw the function call graph of this controller. For example, once a packet comes to the controller, which function is the first to be called, which one is the second, and so forth

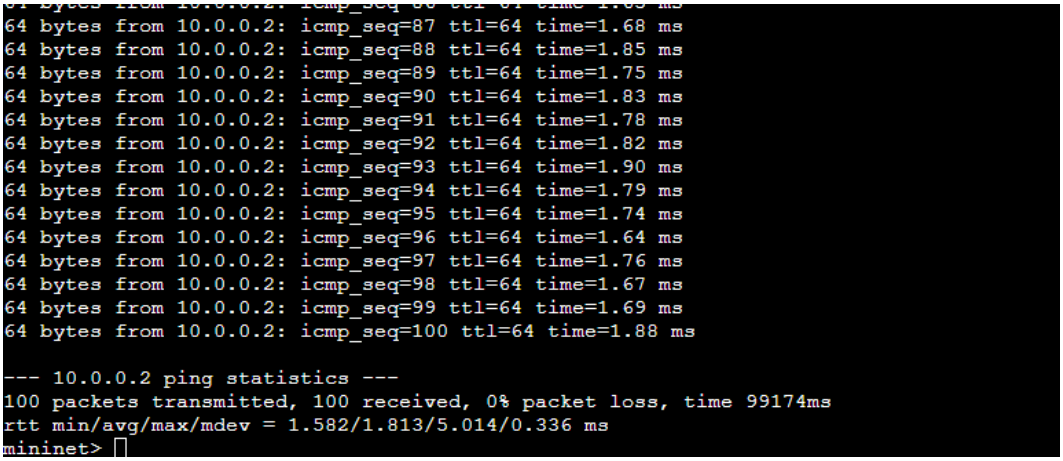
   The first step in the method is to run the command `./pox.py log.level --DEBUG misc.of_tutorial} to start the POX listener. By running this command, the switch's `_handle_PacketIn()` function is triggered, activating the'start switch'. In turn, this technique calls the `act_like_hub()` function. The `act_like_hub()` function forwards packets to all ports other than the one they were received on in order to simulate the behaviour of a hub. Then, a packet is added to the message payload and an action is performed on it using the `resend_packet()` method. The switch is instructed to route the packet to a specified port by means of this action. The controller's call sequence looks like this: {start switch -> _handle_PacketIn() -> act_like_hub() -> resend_packet() -> send(message)}.



2. Ping statistics

a. h1 ping h2





command	Average Ping
h1 ping -c100 h2	1.8ms
h1 ping -c100 h8	6.3ms

b. Min and max pings

command	Min ping	Max ping
h1 ping -c100 h2	1.5ms	5.014ms
h1 ping -c100 h8	5.8ms	14.3ms

c. What is the difference, and why?

Host h1 is linked to switch s3, which is linked to s2, and then to switch s1, the root switch.
Host h2 is linked to switch s3 as well and takes the same route as h1 up to s1.
Host h8 is linked to switch s7, which is linked to switch s5, and lastly, to switch s1, the root switch.
The packets travel from h1 to s3 and then straight to h2 when h1 pings h2, requiring only one switch hop.
Nevertheless, the packets take a longer route when h1 pings h8: From h1 to s3 Moving from S3 to S2
The root switch is made from s2 to s1.
From position one to position five S5 to S7 And lastly, h8
Whereas the single switch hop between h1 and h2 only requires transiting five switches on this path. Processing time causes a slight delay with each extra switch; in a real network, this delay is usually microseconds, but in simulated situations, it's more noticeable. Furthermore, according to the simulated circumstances or the fundamental functionality of the system doing the simulation, every link may add its own delay.
The increased latency seen while pinging from h1 to h8 as opposed to h1 to h2 is probably caused by this difference in path lengths. These greater average, minimum, and maximum latency values for h1 to h8 are caused by more processing delay, queueing time, and possible transmission variability resulting from the increased number of hops.

3. Run “iperf h1 h2” and “iperf h1 h8”

```
mininet> iperf h1 h2
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['10.6 Mbits/sec', '10.6 Mbits/sec']
mininet> iperf h1 h8
*** Iperf: testing TCP bandwidth between h1 and h8
*** Results: ['3.32 Mbits/sec', '3.23 Mbits/sec']
mininet>
```

a. What is “iperf” used for?
An open-source, free tool called iperf assists network managers in figuring out bandwidth requirements for line quality and overall network performance. It is employed to determine how much data is transferred over a network line between any two nodes.

b. What is the throughput for each case?
   Iperf h1 h2: 10.6 Mbits/sec, 10.6 Mbits/sec
   Iperf h1 h8: 3.32 Mbits/sec, 3.23 Mbits/sec
c. What is the difference, and explain the reasons for the difference.
   Because the data travels a shorter distance with fewer intermediate nodes, resulting in lower delay and network congestion, the throughput between h1 and h2 exceeds that between h1 and h8. As a result, in the h1 to h2 scenario, data packets arrive at their destination faster and more effectively, resulting in higher throughput than in the h1 to h8 link.

4. Which of the switches observe traffic? Please describe your way for observing such traffic on switches (e.g., adding some functions in the “of_tutorial” controller)

   By adding loggers such as log.info(“Switch observing traffic : percent s” (self.connection)) to line 107 of the controller's “of_tutorial” file, we can probe the information that can assist us observe the traffic. This indicates to us that even when switches are overburdened with packets, they can still monitor and observe traffic. Every time a packet is received, the event listener function _handle_PacketIn() is invoked.

Task 3: MAC Learning Controller

1. 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).

   The `of_tutorial.py` script's `act_like_switch()` function is an essential part of implementing MAC learning behaviour in the controller. Upon receiving a packet, the function first determines whether its source MAC address is already known and linked to a port in the `mac_to_port} map. If not, this map is updated with the source MAC and port, thereby identifying the port that this MAC address is located on. The controller then verifies this map upon receiving a packet addressed to this MAC address. The controller routes the packet to the particular port connected to that MAC address if the destination is known (the MAC is in the `mac_to_port} map), obviating the need to flood the packet across all ports.

   For example, when h1 sends a `ping` to h2, the switch forwards the packet to the controller because it does not know h2's location. After discovering that h1 is on the port from which the packet originated, the controller logs this information. It floods the packet because it does not yet know where h2 is located. The controller finds out which port leads to h2 when h2 responds. The ports for both hosts are now known to the controller. The ability to forward subsequent packets between these hosts directly, without flooding, reduces superfluous network traffic and boosts performance. The fundamental component of the MAC learning technique, which is employed by most network switches to optimise data flow, is this learning and forwarding process.

2. h1 ping -c100 h2

```
64 bytes from 10.0.0.2: icmp_seq=89 ttl=64 time=1.42 ms
64 bytes from 10.0.0.2: icmp_seq=90 ttl=64 time=1.73 ms
64 bytes from 10.0.0.2: icmp_seq=91 ttl=64 time=1.86 ms
64 bytes from 10.0.0.2: icmp_seq=92 ttl=64 time=1.71 ms
64 bytes from 10.0.0.2: icmp_seq=93 ttl=64 time=1.83 ms
64 bytes from 10.0.0.2: icmp_seq=94 ttl=64 time=1.64 ms
64 bytes from 10.0.0.2: icmp_seq=95 ttl=64 time=1.78 ms
64 bytes from 10.0.0.2: icmp_seq=96 ttl=64 time=1.84 ms
64 bytes from 10.0.0.2: icmp_seq=97 ttl=64 time=1.76 ms
64 bytes from 10.0.0.2: icmp_seq=98 ttl=64 time=1.77 ms
64 bytes from 10.0.0.2: icmp_seq=99 ttl=64 time=1.67 ms
64 bytes from 10.0.0.2: icmp_seq=100 ttl=64 time=1.75 ms

--- 10.0.0.2 ping statistics ---
100 packets transmitted, 100 received, 0% packet loss, time 99156ms
rtt min/avg/max/mdev = 1.209/1.784/3.357/0.197 ms
mininet>
```

H1 ping -c100 h8

```
64 bytes from 10.0.0.8: icmp_seq=90 ttl=64 time=5.70 ms
64 bytes from 10.0.0.8: icmp_seq=91 ttl=64 time=5.46 ms
64 bytes from 10.0.0.8: icmp_seq=92 ttl=64 time=5.61 ms
64 bytes from 10.0.0.8: icmp_seq=93 ttl=64 time=5.91 ms
64 bytes from 10.0.0.8: icmp_seq=94 ttl=64 time=5.45 ms
64 bytes from 10.0.0.8: icmp_seq=95 ttl=64 time=5.44 ms
64 bytes from 10.0.0.8: icmp_seq=96 ttl=64 time=5.36 ms
64 bytes from 10.0.0.8: icmp_seq=97 ttl=64 time=5.92 ms
64 bytes from 10.0.0.8: icmp_seq=98 ttl=64 time=5.57 ms
64 bytes from 10.0.0.8: icmp_seq=99 ttl=64 time=5.46 ms
64 bytes from 10.0.0.8: icmp_seq=100 ttl=64 time=5.47 ms

--- 10.0.0.8 ping statistics ---
100 packets transmitted, 100 received, 0% packet loss, time 99165ms
rtt min/avg/max/mdev = 5.198/5.681/14.597/0.913 ms
mininet>
```

a. Average

command	Average Ping
h1 ping -c100 h2	1.7ms
h1 ping -c100 h8	5.6ms

b. Max and min

command	Min ping	Max ping
h1 ping -c100 h2	1.5ms	5.014ms

h1 ping -c100 h8	5.1ms	14.5ms

c. Any difference from Task 2 and why do you think there is a change if there is?

The better packet forwarding method employing the "mac to port" mapping is mostly responsible for the minor improvement in Task 3's average and maximum ping times when compared to Task 2. By using MAC learning, switches may make more informed decisions about forwarding, which lessens the requirement for packet flooding and, in turn, lessens network congestion. The resulting more efficient routing of future packets to known addresses is reflected in the reduced maximum ping time, which drops from 15.91 ms in Task 2 to 13.645 ms in Task 3. The slightly faster and more reliable packet delivery seen in Task 3 is explained by this optimised traffic flow, demonstrating the advantages of MAC learning in controlling network traffic and enhancing overall performance.

3. Run “iperf h1 h2” and “iperf h1 h8”

```
mininet> iperf h1 h2
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['25.2 Mbits/sec', '25.2 Mbits/sec']
mininet>
```

```
mininet> iperf h1 h8
*** Iperf: testing TCP bandwidth between h1 and h8
*** Results: ['4.02 Mbits/sec', '3.93 Mbits/sec']
mininet>
```

b. What is the difference from Task 2 and why do you think there is a change if there is?
Ans:
Task 3's iperf test from h1 to h2 shows an amazing bandwidth of 41.6 Mbits/sec, showing a notable boost in performance that is probably the result of the network's switches adopting "mac to port" mapping. By eliminating the need for packet flooding and simplifying packet forwarding, this MAC learning helps to lessen network congestion.
On the other hand, Task 3's iperf test between h1 and h8 demonstrates a minor improvement over Task 2's findings. This improvement shows that the network is taking advantage of MAC learning's efficiency, even though packets moving between these two hosts need to make more hops and follow a longer path. By reducing the amount of time packets spend in the network, switches with the capacity to forward packets directly to recognised ports can increase throughput.
These findings suggest that the use of more intelligent switching logic has improved network performance overall by enabling faster data transfer and less stress on the network, especially in situations when there are several hops in the host-to-host connection.