1. The Python API and Topologies (50P)

Please explain – line by line – what happens in the following code:

This code creates and tests a simple network.

1: Define a function, the name is runExp which create a network and run simple performance test

2: Create a class instance customTopo, give the parameter n=2.

3: Call Mininet to create a network the parameter is topo which is a variable contain information of how to create the network.

4: Start this network.

5: Pass this created network to Command-line interface (CLI) which is useful for debugging the network, as it allows to view the network topology (with the net command), test connectivity (with the pingall command), and send commands to individual hosts.

6: Stop this network.

7: No code.

8: Create a class called customTopo which inherited from class Topo.

9: Initiate this class, with parameter self which point this class itself, and parameter n, and parameter **kwargs which allows you to pass keyworded variable length of arguments to a function.

10: Initiate Topo class which is inherited from class Topo.

11: Function addHost from Topo can create host. Here host 1 and host 2 are created.

12: Switch s1 is created.

13: A loop from 0 to n-1.

14: Add link s1 to h1.

15: Add link s1 to h2

16: No code.

17: If this file is the executed then 18, and 19 will be executed, if this file is imported by other files 18 and 19will not be executed.

18: SetLogLevel('info') tell mininet to print useful information.

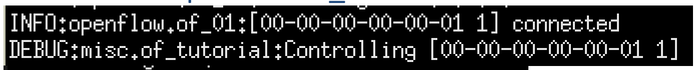
19: Run the function runExp() began from 1.

2. The POX controller (100P)

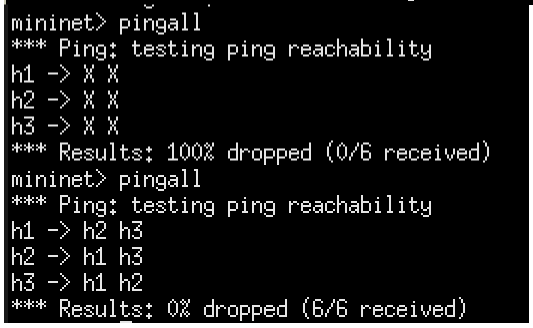
b. (10P) What do you observe after connecting the controller?

After connecting the controller hosts can communicate to each other, but before that, the hosts cannot communicate.

It shows the openflow-of_01 connected:



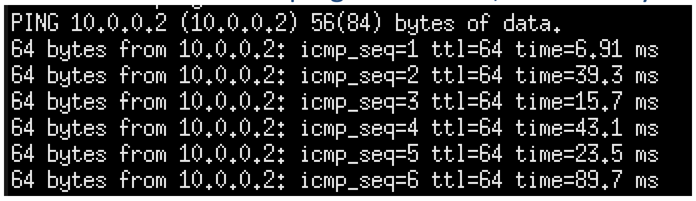
```
INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
DEBUG:misc.of_tutorial:Controlling [00-00-00-00-00-01 1]
```



```
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X
h2 -> X X
h3 -> X X
*** Results: 100% dropped (0/6 received)
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (6/6 received)
```

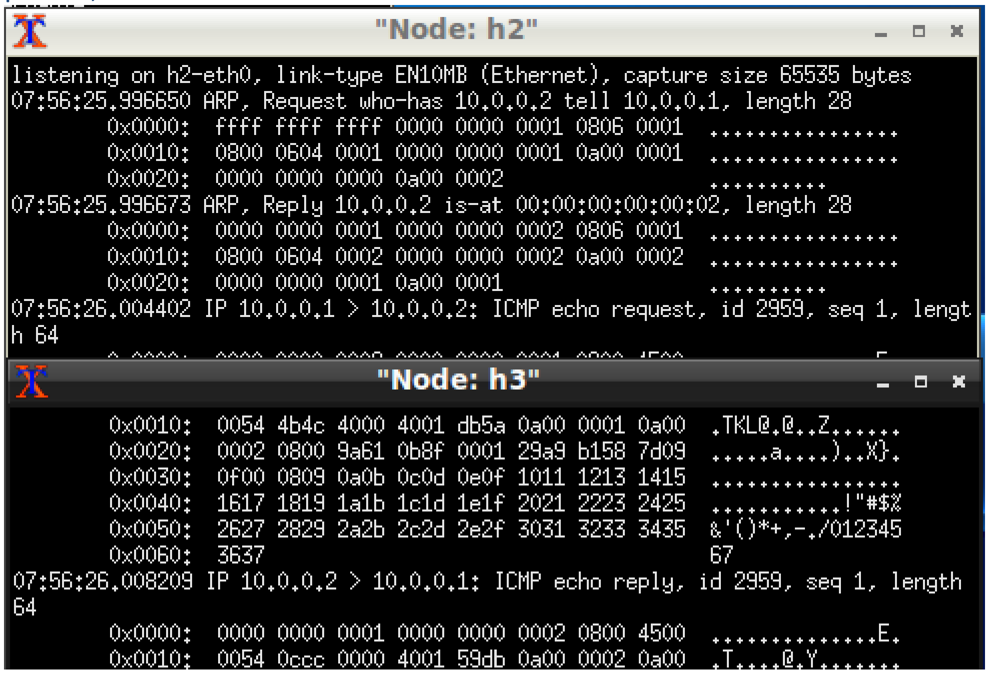
c. (20P) Open one xterm window for each host and run tcpdump on hosts 1 and 2:

Whatever which host ping which host, the latency is unstable:



```
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=6.91 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=39.3 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=15.7 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=43.1 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=23.5 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=89.7 ms
```

Even we use h1 to ping h2. We can see that in both h2 and h3 they have the same ARP and ICMP packet, because hub use broadcast to communicate.



"Node: h2"

```
listening on h2-eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
07:56:25.996650 ARP, Request who-has 10.0.0.2 tell 10.0.0.1, length 28
        0x0000:  ffff ffff ffff 0000 0000 0001 0806 0001  ................
        0x0010:  0800 0604 0001 0000 0000 0001 0a00 0001  ................
        0x0020:  0000 0000 0000 0a00 0002                 ..........
07:56:25.996673 ARP, Reply 10.0.0.2 is-at 00:00:00:00:00:02, length 28
        0x0000:  0000 0000 0001 0000 0000 0002 0806 0001  ................
        0x0010:  0800 0604 0002 0000 0000 0002 0a00 0002  ................
        0x0020:  0000 0000 0001 0a00 0001                 ..........
07:56:26.004402 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 2959, seq 1, lengt
h 64
```

"Node: h3"

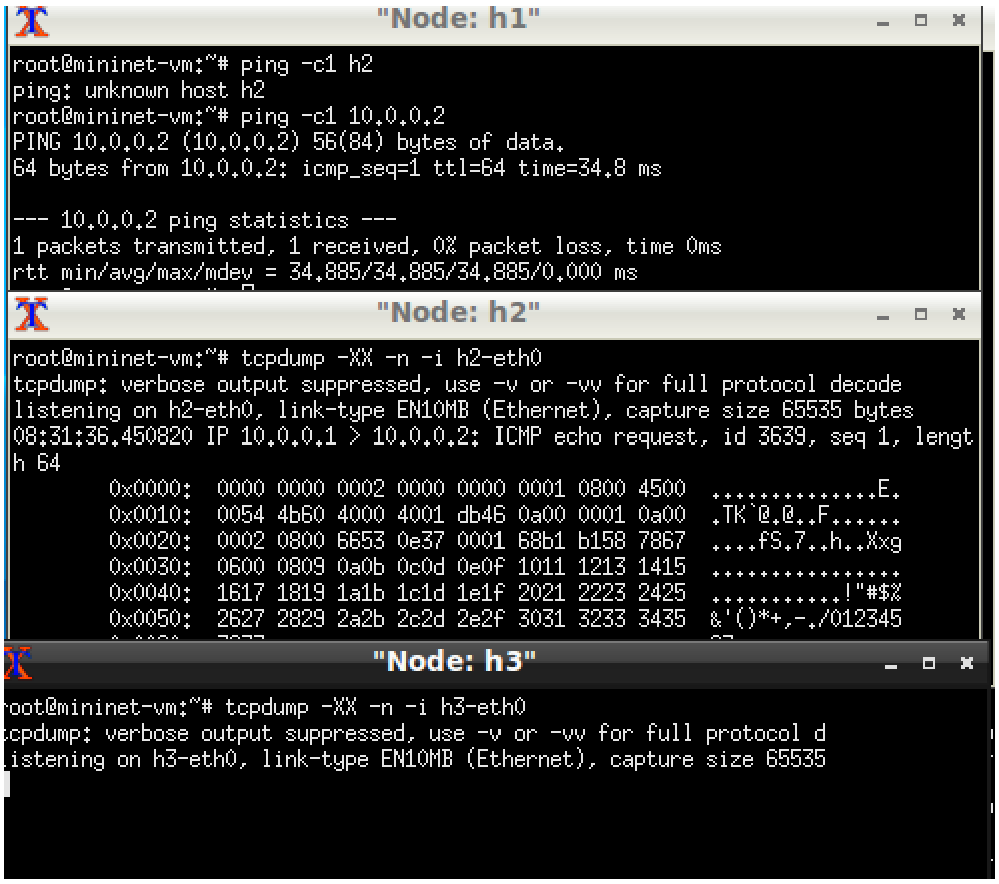
```
        0x0010:  0054 4b4c 4000 4001 db5a 0a00 0001 0a00  .TKL@.@..Z......
        0x0020:  0002 0800 9a61 0b8f 0001 29a9 b158 7d09  .....a....)..X}.
        0x0030:  0f00 0809 0a0b 0c0d 0e0f 1011 1213 1415  ................
        0x0040:  1617 1819 1a1b 1c1d 1e1f 2021 2223 2425  ............!"#$%
        0x0050:  2627 2829 2a2b 2c2d 2e2f 3031 3233 3435  &'()*+,-./012345
        0x0060:  3637                                     67
07:56:26.008209 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 2959, seq 1, length
64
        0x0000:  0000 0000 0001 0000 0000 0002 0800 4500  ..............E.
        0x0010:  0054 0ccc 0000 4001 59db 0a00 0002 0a00  .T....@.Y.......
```

e. (70P) Open up the file `pox/pox/misc/of_tutorial.py` in the editor of your choice. In the Python code you will see that we are currently operating our controller with the help of the `act_like_hub()` method. Your task in this exercise is to

i. (10P) modify the controller to use `act_like_switch()` instead.

ii. (40P) implement `act_like_switch()` so that your controller actually acts like a switch. There are some helpful hints in the code.

After Change the code we get this:



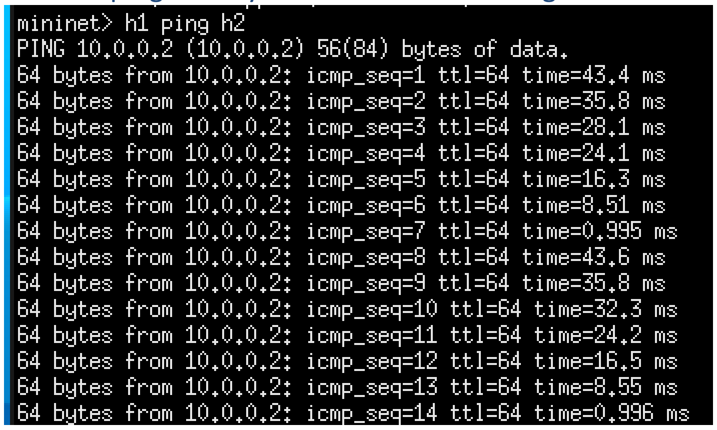
```
"Node: h1"
root@mininet-vm:~# ping -c1 h2
ping: unknown host h2
root@mininet-vm:~# ping -c1 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=34.8 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 34.885/34.885/34.885/0.000 ms
```

```
"Node: h2"
root@mininet-vm:~# tcpdump -XX -n -i h2-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h2-eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
08:31:36.450820 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 3639, seq 1, lengt
h 64
        0x0000:  0000 0000 0002 0000 0000 0001 0800 4500  ..............E.
        0x0010:  0054 4b60 4000 4001 db46 0a00 0001 0a00  .TK`@.@..F......
        0x0020:  0002 0800 6653 0e37 0001 68b1 b158 7867  ....fS.7..h..Xxg
        0x0030:  0600 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
```

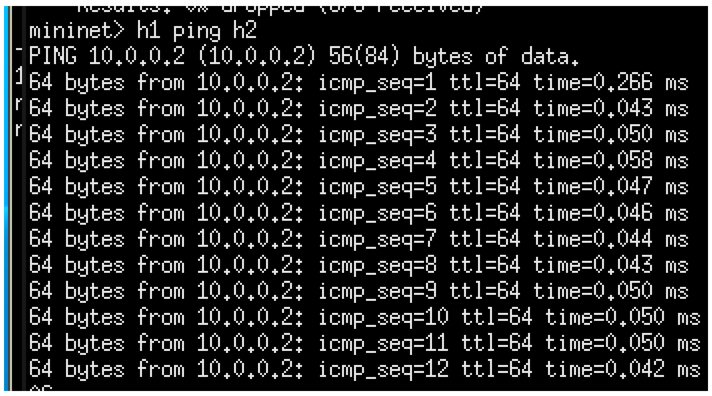
```
"Node: h3"
oot@mininet-vm:~# tcpdump -XX -n -i h3-eth0
cpdump: verbose output suppressed, use -v or -vv for full protocol d
istening on h3-eth0, link-type EN10MB (Ethernet), capture size 65535
```

When h1 ping h2, h3 will not receive any packet. It act like a switch (Only broadcast when learning, not broadcasting everything).
And the ping latency is decrease but raise again.



```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=43.4 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=35.8 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=28.1 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=24.1 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=16.3 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=8.51 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.995 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=43.6 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=35.8 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=32.3 ms
64 bytes from 10.0.0.2: icmp_seq=11 ttl=64 time=24.2 ms
64 bytes from 10.0.0.2: icmp_seq=12 ttl=64 time=16.5 ms
64 bytes from 10.0.0.2: icmp_seq=13 ttl=64 time=8.55 ms
64 bytes from 10.0.0.2: icmp_seq=14 ttl=64 time=0.996 ms
```

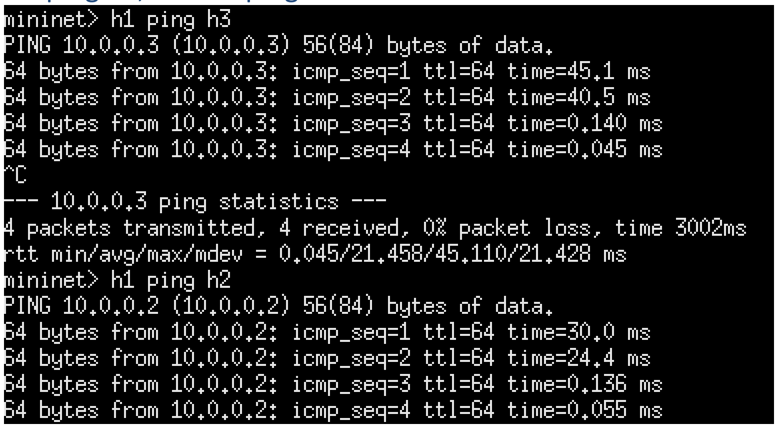
iii. (20P) It is somewhat inefficient to let the controller decide the fate of every single packet. Implement an **act_like_flow_switch()** method in which your controller instead installs flow-rules into the switch for all packets that are initiually flow-table misses. This will improve the performance of the forwarding on subsequent packets. You can use the POX API and the POX documentation regarding OpenFlow here (hint: have a very close look at **ofp_flow_mod**).
After implement act_like_flow_switch(), use h1 ping h2, we get the result in following:

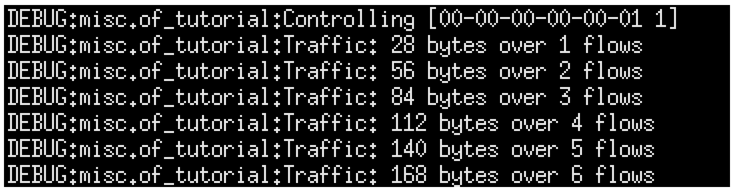

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.266 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.043 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.050 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.058 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.047 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.046 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.044 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.043 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=0.050 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=0.050 ms
64 bytes from 10.0.0.2: icmp_seq=11 ttl=64 time=0.050 ms
64 bytes from 10.0.0.2: icmp_seq=12 ttl=64 time=0.042 ms
```
The code is in the attachment of_tutorial.py with this solution.

3. POX Extension (50P)

(45P) Extend your POX controller so that it also acts as a traffic monitor when acting as a flow learning switch (act_like_flow_switch()). The monitor should simply count all the traffic in bytes that is going to or coming from host h1.

H1 ping h3, and h1 ping h2


```
mininet> h1 ping h3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=45.1 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=40.5 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=0.140 ms
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=0.045 ms
^C
--- 10.0.0.3 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3002ms
rtt min/avg/max/mdev = 0.045/21.458/45.110/21.428 ms
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=30.0 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=24.4 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.136 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.055 ms
```
Observed the traffic:


```
DEBUG:misc.of_tutorial:Controlling [00-00-00-00-00-01 1]
DEBUG:misc.of_tutorial:Traffic: 28 bytes over 1 flows
DEBUG:misc.of_tutorial:Traffic: 56 bytes over 2 flows
DEBUG:misc.of_tutorial:Traffic: 84 bytes over 3 flows
DEBUG:misc.of_tutorial:Traffic: 112 bytes over 4 flows
DEBUG:misc.of_tutorial:Traffic: 140 bytes over 5 flows
DEBUG:misc.of_tutorial:Traffic: 168 bytes over 6 flows
```
The code is in the attachment of_tutorial.py

(5P) Why is it not sufficient to count traffic in handle_packet_in()?

Because it only shows the traffic that the switch does not know how to deal with. It actually shows the traffic between switch and controller corresponding to h1. If a packet fit the flow table rules, it will not be counted.