

Tapping, Blocking and Point-to-Point circuits with SDN Mininet

Muyi Chen
Kaifeng Lu

1. Environment set up

1.1 Manual Operation of Network Data Plane

We first illustrate how in OpenFlow a freshly initiated network data plane, absent any controllers providing instructions, is unable to process any packets. For simulating the OpenFlow-based SDN network, we utilize a simulator called Mininet¹ to set up a simple star topology, where all hosts are connected via a central switch (see Figure 1). The switch in use is a production-grade software OpenFlow switch called Open vSwitch² (OVS).

To create this topology in Mininet, run the following command all in one line:

```
sudo mn --topo single,3 --mac --switch ovsk --controller remote
```

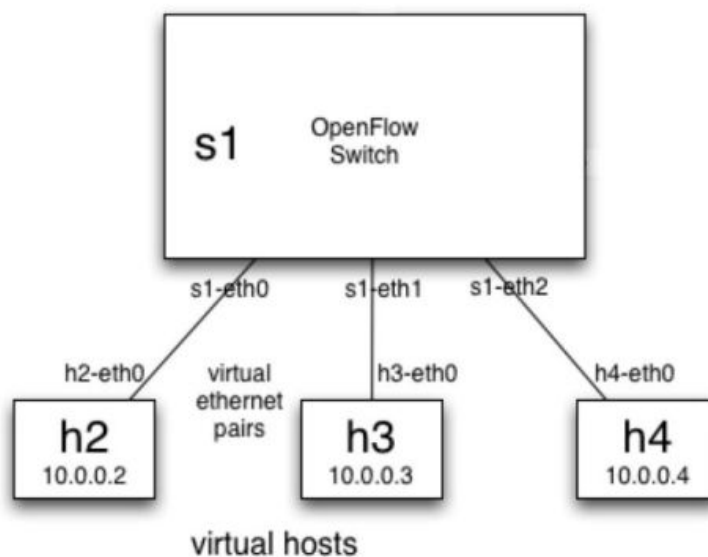


Figure 1: Star Network Topology

a. In Ubuntu command line, type in :

```
sudo mn --topo single,3 --mac --switch ovsk --controller remote
```

Which creates a OVS with a single switch s1 and 3 hosts. No controller inside, all controllers linked with remote.

Output:

```
ubuntu@ece361:~$ sudo mn --topo single,3 --mac --switch ovsk --controller remote
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1)
*** Configuring hosts
h1 h2 h3
*** Starting controller
*** Starting 1 switches
s1
*** Starting CLI:
mininet> █
```

Once in Mininet, it creates a console-like environment for you where each line starts with the prompt mininet>. You can run commands in each of the virtual hosts by typing the host name followed by the command () in the Mininet console, for example: mininet> h1 ifconfig

Command: mininet> h1

Output:

```
ubuntu@ece361: ~
*** Starting controller
*** Starting 1 switches
s1
*** Starting CLI:
mininet> h1 ifconfig
h1-eth0  Link encap:Ethernet  HWaddr 00:00:00:00:00:01
          inet addr:10.0.0.1  Bcast:10.255.255.255  Mask:255.0.0.0
          inet6 addr: fe80::200:ff:fe00:1/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0 frame:0
          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:468 (468.0 B)  TX bytes:796 (796.0 B)

lo       Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

mininet> █
```

- b. Try to ping from host h1 to host h2 by typing:
mininet> h1 ping -c4 h2

Which pings 4 icmp_sequence from h1 to h2

Output:

```
mininet> h1 ping -c4 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
From 10.0.0.1 icmp_seq=4 Destination Host Unreachable

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 3016ms
pipe 3
mininet> █
```

Q1: Do you observe any successful pings?

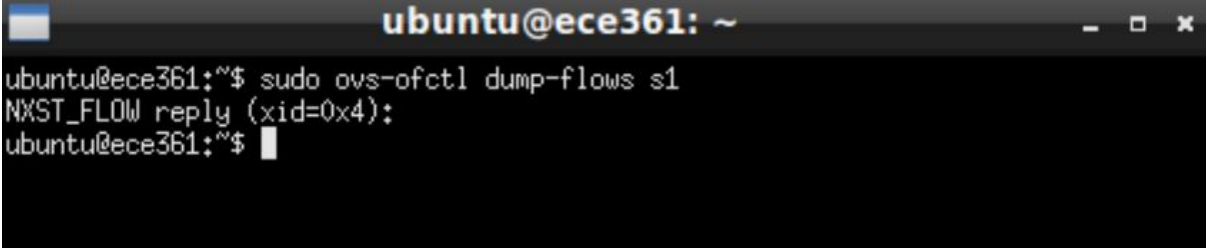
Ans: No, there are no successful pings. There is no connection between h1 and h2 set up through switch s1 yet.

- c. While keeping the terminal with Mininet active, open up a second terminal window. Dump the current state of the packet forwarding table in the switch by typing:

```
sudo ovs-ofctl dump-flows s1
```

The dump-flows instruction only shows all current forwarding information on switch s1, and does not change any data from the forwarding table.

Output:



```
ubuntu@ece361: ~
ubuntu@ece361:~$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
ubuntu@ece361:~$ █
```

- d. Now insert a flow to allow h1 to send packets to h2, and vice-versa, by issuing the following two commands:

```
sudo ovs-ofctl add-flow s1 in_port=1,actions=output:2
sudo ovs-ofctl add-flow s1 in_port=2,actions=output:1
```

Which establishes a connection between h1 and h2.

Output:

```
ubuntu@ece361:~$ sudo ovs-ofctl add-flow s1 in_port=1,actions=output:2
ubuntu@ece361:~$ sudo ovs-ofctl add-flow s1 in_port=2,actions=output:1
ubuntu@ece361:~$
```

- e. Dump the state of the packet forwarding table again.

```
sudo ovs-ofctl dump-flows s1
```

Q3: Note the differences and interpret the flow table.

```
ubuntu@ece361:~$ sudo ovs-ofctl add-flow s1 in_port=1,actions=output:2
ubuntu@ece361:~$ sudo ovs-ofctl add-flow s1 in_port=2,actions=output:1
ubuntu@ece361:~$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=148.167s, table=0, n_packets=0, n_bytes=0, in_port=1 actions=output:2
  cookie=0x0, duration=127.932s, table=0, n_packets=0, n_bytes=0, in_port=2 actions=output:1
ubuntu@ece361:~$
```

Ans: In the flow table we can observe 2 flows, linking h1 and h2, no data transmitted yet.

- f. Now in the Mininet terminal, try pinging from h1 to h2 again.

```
mininet> h1 ping -c4 h2
```

Output:

```
mininet> h1 ping -c2 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_req=1 ttl=64 time=0.177 ms
64 bytes from 10.0.0.2: icmp_req=2 ttl=64 time=0.051 ms

--- 10.0.0.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.051/0.114/0.177/0.063 ms
mininet> h1 ping -c4 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_req=1 ttl=64 time=0.178 ms
64 bytes from 10.0.0.2: icmp_req=2 ttl=64 time=0.042 ms
64 bytes from 10.0.0.2: icmp_req=3 ttl=64 time=0.067 ms
64 bytes from 10.0.0.2: icmp_req=4 ttl=64 time=0.039 ms

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2997ms
rtt min/avg/max/mdev = 0.039/0.081/0.178/0.057 ms
mininet>
```

Firstly pinged 2 packets of 64 bytes each, then 4 packets again. All packets received.

Q4: Are the pings successful? Why?

Ans: Yes , all pings are successful. Because a connection has already been set up in step d.

- g. Dump the flow table again to see the difference:

```
sudo ovs-ofctl dump-flows s1
```

Output:

```
ubuntu@ece361:~$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=736.992s, table=0, n_packets=9, n_bytes=714, in_port=1 actions=output:2
  cookie=0x0, duration=716.757s, table=0, n_packets=9, n_bytes=714, in_port=2 actions=output:1
ubuntu@ece361:~$
```

It shows there are packets being transferred through the 2 flows.

2. Starting Network Controller

Previously, you manually installed flows into the switch in order to define the network's behaviour. The next step is to start a network controller daemon process, which the switch will automatically connect to. The controller will be able to provide instructions to the switch on how it should handle its packets by installing the necessary flows on-the-fly.

The controller used in this lab is called Ryu, which comes pre-installed in the VM you are using. The default instructions it provides to the data plane emulates the logic of a simple layer-2 learning switch .

- a. Start the network controller (Ryu) by 3 issuing the following command from the second (non-Mininet) terminal:

```
ece361-start-controller switch
```

Output:

```
ubuntu@ece361:~$ ece361-start-controller switch
Started new Ryu instance with PID 1574
ubuntu@ece361:~$
```

A new controller with PID 1574 is started

- b. If at any time you want to stop it, you can issue the command:

```
Ece361-stop-controller
```

(Not performed here)

- c. With the controller started, dump the state of the packet forwarding table

```
sudo ovs-ofctl dump-flows s1
```

Output:

```
ubuntu@ece361:~$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=1290.895s, table=0, n_packets=9, n_bytes=714, in_port=1 actions=output:2
  cookie=0x0, duration=1270.66s, table=0, n_packets=9, n_bytes=714, in_port=2 actions=output:1
  cookie=0x0, duration=155.526s, table=0, n_packets=0, n_bytes=0, priority=65535, dl_dst=01:80:c2:00:00:0e, dl_type=0x88cc actions=CONTROLLER:51
ubuntu@ece361:~$
```

Notice a new controller cookie is shown in table

Q5: Note the differences and see if you can interpret the flow table.

Ans: for all cookies, the duration indicates the time period since it has been created. The flow table has index 0, the first 2 cookies are the manually setup connections between h1 and h2, and 9 packets (714 bytes in total) has been transmitted through this path. The third cookie shows the newly created Ryu controller, no data has been transmitted through it yet.

- d. Ping from any host to any other host

```
mininet> h1 ping -c4 h2
```

Output:

```
mininet> h1 ping -c4 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data:
64 bytes from 10.0.0.2: icmp_req=1 ttl=64 time=0.171 ms
64 bytes from 10.0.0.2: icmp_req=2 ttl=64 time=0.055 ms
64 bytes from 10.0.0.2: icmp_req=3 ttl=64 time=0.050 ms
64 bytes from 10.0.0.2: icmp_req=4 ttl=64 time=0.051 ms

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3001ms
rtt min/avg/max/mdev = 0.050/0.081/0.171/0.052 ms
mininet>
```

Successfully transmitted.

- e. Dump the packet forwarding table again to note the differences.

```
sudo ovs-ofctl dump-flows s1
```

Output:

```
ubuntu@ece361:~$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=1987.603s, table=0, n_packets=15, n_bytes=1190, in_port=1
  actions=output:2
  cookie=0x0, duration=1967.368s, table=0, n_packets=15, n_bytes=1190, in_port=2
  actions=output:1
  cookie=0x0, duration=852.234s, table=0, n_packets=0, n_bytes=0, priority=65535,
  dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:51
ubuntu@ece361:~$
```

Note: All data transmitted through the preset connection, NOT through the controller!

Q6: Record these differences and interpret them.

Ans: From the table we can see that all data pinged from h1 to h2 are through the prearranged connection instead of the controller. Compares to last time, more packets are sent through the connection represented by cookie 1&2. No data transmission in cookie 3, which is the Ryu controller. Also, aside of the 4 packets sent by the command “h1 ping -c4 h2”, 2 extra packets of transmission appeared in cookie 1&2. This may be because of the newly added controller.

3. Tapping Traffic

Through a programmatic interface provided to you, you are to override the default behaviour within the network. This abstraction is provided in the form of a Python library, `ryu_ofctl`, that allows you to call functions that can install and remove flows into the switch, hence, allowing you define the behaviour of the network via software. Note that the functionality of this library leverages the controller in the previous section, so ensure it is running first.

In this part of the lab, you will install flows in order to tap the traffic flowing between h1 and h3. This involves duplicating each packet being sent between h1 and h3, and sending the duplicate copy to h2, allowing h2 to monitor the inter-communications of h1 and h3. TASK From the non-Mininet terminal, use the Python `ryu_ofctl` library (a few sample workflows can be found in the Github repository) to write a short Python script that will install the necessary flows to implement the network tap.

Refer to the following README for examples:

https://github.com/t-lin/ryu_ofctl/blob/master/README.md

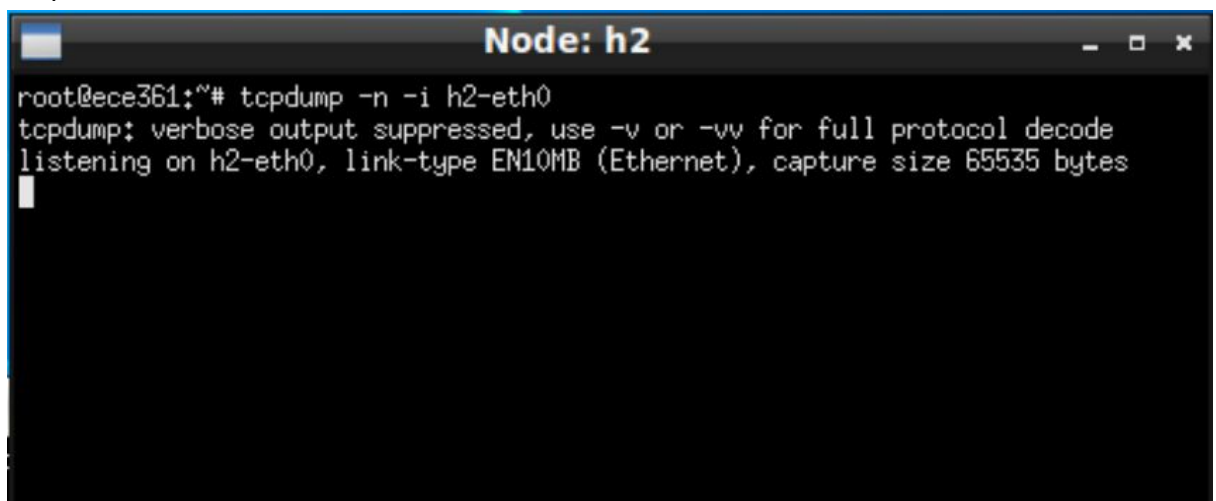
- a. To check if your code is working, it may help to open up a separate terminal session for just h2. To do this in the Mininet console, run:

```
mininet> xterm h2
```

In the resulting terminal that pops up, which resides within h2, you can listen on its network interface for all traffic (incoming or outgoing) by issuing the command:

```
tcpdump -n -i h2-eth0
```

Output:



```
Node: h2
root@ece361:~# tcpdump -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
```

This is in new terminal node h2

- b. Construct python program that lets h1 send packets to h2 and h3, and h3 sends packets to h1 and h2.
- h1 -> h2 & h1 -> h3
 - h3 -> h2 & h3 -> h1

Code:


```

1  import ryu_ofctl
2
3  #clean all existing flow
4  dpid = 1 #S1
5  ryu_ofctl.deleteAllFlows(dpid)
6
7  #flow 1: h1 to h2 and h3
8  flow = ryu_ofctl.FlowEntry()
9
10 #h1 output to h2
11 act = ryu_ofctl.OutputAction(2)
12 #listen from h1
13 flow.in_port = 1
14 #let flow output on h2
15 flow.addAction(act)
16 #push to system
17 ryu_ofctl.insertFlow(dpid,flow)
18
19 #h1 output to h3
20 act1 = ryu_ofctl.OutputAction(3)
21 #listen from h1
22 flow.in_port = 1
23 #let flow output on h3
24 flow.addAction(act1)
25 #push to s1
26 ryu_ofctl.insertFlow(dpid,flow)
27
28

```

```

28
29 #flow 2: h3 to h2 and h1
30 flow1 = ryu_ofctl.FlowEntry()
31
32 #listen from h3
33 flow1.in_port = 3
34 #let flow output on h2
35 flow1.addAction(act)
36 #push to s1
37 ryu_ofctl.insertFlow(dpid,flow1)
38
39 #h3 output to h1
40 act2 = ryu_ofctl.OutputAction(1)
41 #listen from h1
42 flow1.in_port = 3
43 #let flow output on h3
44 flow1.addAction(act2)
45 #push to system
46 ryu_ofctl.insertFlow(dpid,flow1)

```

Flow 1: h1 -> h2 & h1 -> h3

Flow 2: h3 -> h2 & h3 -> h1

- c. Compile the python code in the NON-Mininet terminal
- Change directory with command "cd <directory>" to where the python file "tapping_traffic.py" is.
 - Run command "python tapping_traffic.py"
 - Check flow table by "sudo ovs-ofctl dump-flows s1"

Output:

Step ii:

```
ubuntu@ece361:~/Desktop$ python tapping_traffic.py
ubuntu@ece361:~/Desktop$
```

Step iii:

```
ubuntu@ece361:~/Desktop$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=2.568s, table=0, n_packets=0, n_bytes=0, priority=40000,in
_port=1 actions=output:2,output:3
  cookie=0x0, duration=2.466s, table=0, n_packets=0, n_bytes=0, priority=40000,in
_port=3 actions=output:2,output:1
```

Note the port and output configuration

- d. To test your implementation, from the Mininet console, ping between h1 and h3. If the tapping is working, h2 should be able to see all the traffic between the two other hosts.

```
mininet> h1 ping -c4 h3
```

Output on mininet terminal:

```
mininet> h1 ping -c4 h3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data:
64 bytes from 10.0.0.3: icmp_req=1 ttl=64 time=0.233 ms
64 bytes from 10.0.0.3: icmp_req=2 ttl=64 time=0.066 ms
64 bytes from 10.0.0.3: icmp_req=3 ttl=64 time=0.065 ms
64 bytes from 10.0.0.3: icmp_req=4 ttl=64 time=0.074 ms

--- 10.0.0.3 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 0.065/0.109/0.233/0.072 ms
mininet>
```

Output on h2 terminal:

```
Node: h2

03:01:58.249767 ARP, Request who-has 10.0.0.3 tell 10.0.0.1, length 28
03:01:58.249813 ARP, Reply 10.0.0.3 is-at 00:00:00:00:00:03, length 28
03:01:58.249837 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 2030, seq 1, length 64
03:01:58.249865 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 2030, seq 1, length 64
03:01:59.248670 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 2030, seq 2, length 64
03:01:59.248698 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 2030, seq 2, length 64
03:02:00.251822 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 2030, seq 3, length 64
03:02:00.251848 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 2030, seq 3, length 64
03:02:01.254501 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 2030, seq 4, length 64
03:02:01.254536 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 2030, seq 4, length 64
03:02:03.256144 ARP, Request who-has 10.0.0.1 tell 10.0.0.3, length 28
03:02:03.256214 ARP, Reply 10.0.0.1 is-at 00:00:00:00:00:01, length 28
```

Success!

Note every sequence is caught twice because the first time h2 catches flow h1 -> h3, and then catches flow h3 -> h1. The ping command sends a packet out to a terminal and then asks the terminal to send it back in order to measure the round trip time.

4. Blocking Traffic

In this part, you will install flows to block the traffic flowing between h1 and h3. Blocking involves inserting a flow that matches on certain header fields, while leaving the actions empty. Note that the blocking should be specific to the communications between these two hosts (i.e. both h1 and h3 should still be able to ping h2).

- a. Please make sure the h2 terminal is still open

```
Node: h2
03:01:58.249767 ARP, Request who-has 10.0.0.3 tell 10.0.0.1, length 28
03:01:58.249813 ARP, Reply 10.0.0.3 is-at 00:00:00:00:00:03, length 28
03:01:58.249837 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 2030, seq 1, length 64
03:01:58.249865 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 2030, seq 1, length 64
03:01:59.248670 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 2030, seq 2, length 64
03:01:59.248698 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 2030, seq 2, length 64
03:02:00.251822 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 2030, seq 3, length 64
03:02:00.251848 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 2030, seq 3, length 64
03:02:01.254501 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 2030, seq 4, length 64
03:02:01.254536 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 2030, seq 4, length 64
03:02:03.256144 ARP, Request who-has 10.0.0.1 tell 10.0.0.3, length 28
03:02:03.256214 ARP, Reply 10.0.0.1 is-at 00:00:00:00:00:01, length 28
```

Nothing changed from previous output

- b. Construct python program that lets h1 send packets to h2 only, and h3 sends packets to h2 only.

Code:

```

1  import ryu_ofctl
2
3  #clean all existing flow
4  dpid = 1 #S1
5  ryu_ofctl.deleteAllFlows(dpid)
6
7  #flow 1: h1 to h2 and h3
8  flow = ryu_ofctl.FlowEntry()
9
10 #h1 output to h2
11 act = ryu_ofctl.OutputAction(2)
12 #listen from h1
13 flow.in_port = 1
14 #let flow output on h2
15 flow.addAction(act)
16 #push to system
17 ryu_ofctl.insertFlow(dpid,flow)
18
19
20 #flow 2: h3 to h2 and h1
21 flow1 = ryu_ofctl.FlowEntry()
22
23 #listen from h3
24 flow1.in_port = 3
25 #let flow output on h2
26 flow1.addAction(act)
27 #push to s1
28 ryu_ofctl.insertFlow(dpid,flow1)
29

```

Flow 1: h1 -> h2

Flow 2: h3 -> h2

c. Compile the python code in the NON-Mininet terminal

- i. Change directory with command “cd <directory>” to where the python file “tapping_traffic.py” is.
- ii. Run command “python blocking_traffic.py”
- iii. Check flow table by “sudo ovs-ofctl dump-flows s1”

Output:

Step ii:

```

ubuntu@ece361:~/Desktop$ python blocking_traffic.py
ubuntu@ece361:~/Desktop$

```

Step iii:

```
ubuntu@ece361:~/Desktop$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=26.53s, table=0, n_packets=0, n_bytes=0, priority=40000,in
  _port=1 actions=output:2
  cookie=0x0, duration=26.335s, table=0, n_packets=0, n_bytes=0, priority=40000,i
  n_port=3 actions=output:2
ubuntu@ece361:~/Desktop$
```

Note the port and output configuration

- d. To test your implementation, from the Mininet console, ping between h1 and h3. If the tapping is working, h2 should be able to see all the traffic between the two other hosts.

```
mininet> h1 ping -c4 h3
```

Output on mininet terminal:

```
mininet> h1 ping -c4 h3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
From 10.0.0.1 icmp_seq=4 Destination Host Unreachable

--- 10.0.0.3 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 3001ms
pipe 3
mininet>
```

Now h1 to h3 is blocked

Then try:

```
mininet> h1 ping -c4 h2
```

Output on mininet terminal:

```
mininet> h1 ping -c4 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_req=1 ttl=64 time=23.5 ms
64 bytes from 10.0.0.2: icmp_req=2 ttl=64 time=0.068 ms
64 bytes from 10.0.0.2: icmp_req=3 ttl=64 time=0.036 ms
64 bytes from 10.0.0.2: icmp_req=4 ttl=64 time=0.085 ms

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3001ms
rtt min/avg/max/mdev = 0.036/5.928/23.525/10.159 ms
mininet>
```

Output on h2 terminal:

```
03:52:49.861556 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 2090, seq 1, length 64
03:52:49.861573 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 2090, seq 1, length 64
03:52:50.840211 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 2090, seq 2, length 64
03:52:50.840231 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 2090, seq 2, length 64
03:52:51.839197 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 2090, seq 3, length 64
03:52:51.839208 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 2090, seq 3, length 64
03:52:52.839365 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 2090, seq 4, length 64
03:52:52.839394 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 2090, seq 4, length 64
03:52:54.871073 ARP, Request who-has 10.0.0.1 tell 10.0.0.2, length 28
03:52:54.871304 ARP, Reply 10.0.0.1 is-at 00:00:00:00:00:01, length 28
```

Success!

Note every sequence is caught twice for the same reason as above. The ping command sends a packet out to a terminal and then asks the terminal to send it back in order to measure the round trip time.

It works the same way for:

```
mininet> h3 ping -c4 h2
```

Please test it on your own