SPRING 2019

EE-555 PROJECT

OpenFlow Protocol

BINGJUN WANG

KRISHNA SAI TARUN PASUPULETI

ABSTRACT

This project is an introduction to the concept of OpenFlow protocol which is widely used in the Software Defined Networking (SDN). SDN is a technological approach where network software can be remotely accessed by network devices without embedding them into the devices and be also can monitored with the help of a controller. Here, we are going to construct an SDN environment with a controller, learning switch and a learning router to show the working of an SDN unit. We use Mininet, which is a network emulation platform for creating the OpenFlow network. The OpenFlow working is shown in the image below [1]

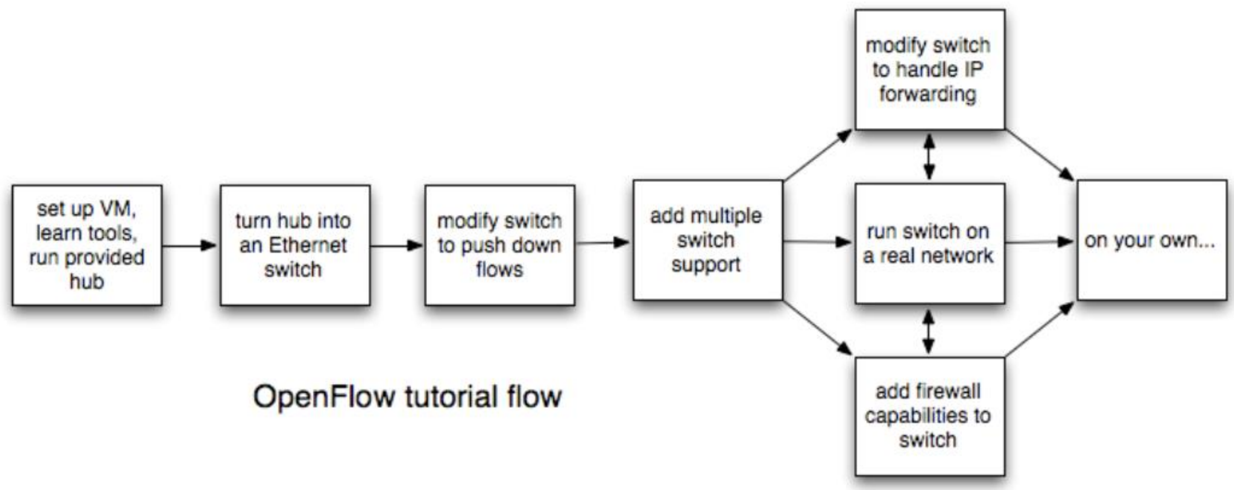


Figure 1: The flow of OpenFlow learning

IMPLEMENTATION OF THE PROJECT

REQUIREMENTS:

This project requires some software to be installed on the host machine. These are:

1. Virtualization Software: Virtual Box software for this project.
2. SSH Terminal: Putty is used to connect to OpenFlow on the mininet running on virtualization software.
3. X server: X Launch is used to connect to the hosts running in the network (virtual network).
4. Operating System/ Platform for the OpenFlow network: Mininet

The detailed steps to get the project running is given below:

SET UP THE VIRTUAL MACHINE

Once you have downloaded the .ovf image,

- **Startup VirtualBox, then select File --> Import Appliance and select the .ovf image that you downloaded.**

You may also be able to simply double-click the .ovf file to open it up in your installed virtualization program.

- **Next, press the "Import" button.**

If you are running VirtualBox, you should make sure your VM has two network interfaces. One should be a NAT interface that it can use to access the Internet, and the other should be a host-only interface to enable it to communicate with the host machine. To Set up a Host-Only Network in VirtualBox. First, add a host-only network interface to VM. Next, bootup VM and make sure that this new interface has showed up as eth1(if not so, verify the network adapter).

VM settings --> Network --> Adapter 2 --> Enable adapter and check Host only.

To allow network access in the VM, execute [2]:

sudo dhclient eth1

LEARNING THE DEVELOPMENT TOOL

The OpenFlow VM includes several OpenFlow-specific utilities pre-installed. Their short descriptions are:

- **OpenFlow Controller**: This sits over the OpenFlow interface. The OpenFlow controller acts as an Ethernet learning switch in combination with an OpenFlow switch.
- **OpenFlow Switch**: This sits below the OpenFlow interface. A user-space software switch is included in the OpenFlow reference distribution. Open vSwitch is another software but it is a kernel-based switch, while there is a number of hardware switches available.

- **ovs-ofctl**: This is a command-line utility that sends quick OpenFlow messages and can be used for viewing switch port and flow stats or manually inserting flow entries.
- **Wireshark**: This is a general graphical utility for viewing packets.
- **iperf**: This is a general command-line utility for testing the speed of a TCP connection.
- **Mininet**: This is a network emulation platform that needs to be run over the Virtualization software. Mininet creates a virtual OpenFlow network - controller, switches, hosts, and links - on a single real or virtual machine.
- **cbench**: This is a utility for testing the flow setup rate of OpenFlow controllers.

Initially we need to create a network and this can done by the following command:

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

With this command, mininet creates the following network:

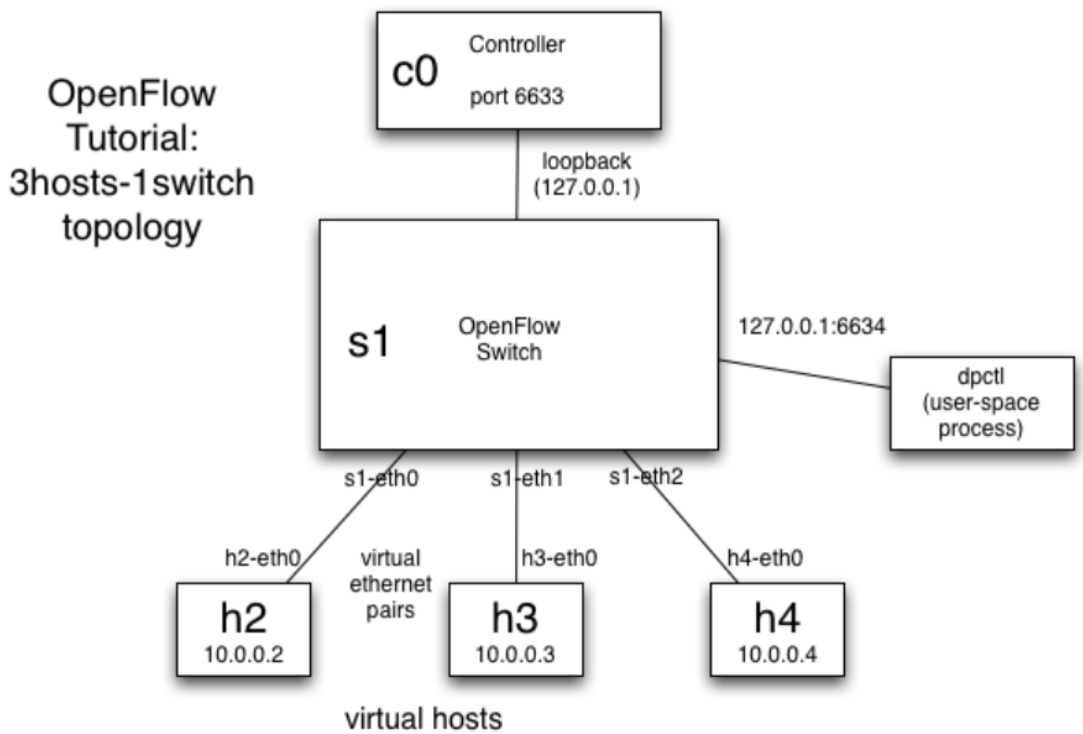


Figure 2: 3 hosts – 1 switch topology

Here's what Mininet just did as shown in figure 2 and figure 3:

- Created 3 virtual hosts, each with a separate IP address.
- Created a single OpenFlow software switch in the kernel with 3 ports.
- Connected each virtual host to the switch with a virtual ethernet cable.
- Set the MAC address of each host equal to its IP.

- Configure the OpenFlow switch to connect to a remote controller.

```
mininet@mininet-vm:~$ sudo mn --topo single,3 --mac --switch ovsk --controller r
emote
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Unable to contact the remote controller at 127.0.0.1:6633
Setting remote controller to 127.0.0.1:6653
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1)
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet>
```

Figure 3: Creating the topology on mininet.

Some useful mininet commands are:

To see the list of nodes in the created network:

mininet> nodes

To get some help and view all the commands:

mininet> help

To check the IP of a virtual host:

mininet> h1 ifconfig

To spawn an xterm for one or more virtual hosts

mininet> xterm h1 h2

To close we can use *exit* and once that is done we can clear all the residual states and processes using

$ sudo mn -c

ovs-ofctl Example Usage

Create a second SSH window and run the following command to connect to the switch and view its table:

$ sudo ovs-ofctl show s1

$ ovs-ofctl dump-flows s1

The output will be as below:

mininet@mininet-vm: ~

```
login as: mininet
mininet@192.168.56.101's password:
Welcome to Ubuntu 14.04.4 LTS (GNU/Linux 4.2.0-27-generic i686)

 * Documentation:  https://help.ubuntu.com/
Last login: Mon Apr 29 20:53:39 2019 from 192.168.56.1
mininet@mininet-vm:~$ sudo ovs-ofctl show s1
OFPT_FEATURES_REPLY (xid=0x2): dpid:0000000000000001
n_tables:254, n_buffers:256
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS ARP_MATCH_IP
actions: OUTPUT SET_VLAN_VID SET_VLAN_PCP STRIP_VLAN SET_DL_SRC SET_DL_DST SET_N
W_SRC SET_NW_DST SET_NW_TOS SET_TP_SRC SET_TP_DST ENQUEUE
 1(s1-eth1): addr:ae:b7:1b:77:21:25
     config:     0
     state:      0
     current:    10GB-FD COPPER
     speed: 10000 Mbps now, 0 Mbps max
 2(s1-eth2): addr:26:ed:04:1f:6d:1a
     config:     0
     state:      0
     current:    10GB-FD COPPER
     speed: 10000 Mbps now, 0 Mbps max
 3(s1-eth3): addr:8a:f7:55:a6:42:e6
     config:     0
     state:      0
     current:    10GB-FD COPPER
     speed: 10000 Mbps now, 0 Mbps max
 LOCAL(s1): addr:1e:30:6b:d6:bc:33
     config:     PORT_DOWN
     state:      LINK_DOWN
     speed: 0 Mbps now, 0 Mbps max
OFPT_GET_CONFIG_REPLY (xid=0x4): frags=normal miss_send_len=0
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
mininet@mininet-vm:~$
```

Figure 4: Working of ovs-ofctl

However, since we haven't started any controller yet, the flow-table should be empty, as shown above in figure 4.

PING TEST

Try to ping h2 from h1 from the Mininet console using below command:

mininet> h1 ping -c3 h2

Ping test is unsuccessful, due to emptiness of switch flow table. Also, there is no controller connected to the switch and therefore the switch cannot make decision on what to do with incoming traffic, which results in ping failure as shown in figure 5.

```
mininet> h1 ping -c3 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

--- 10.0.0.2 ping statistics ---
3 packets transmitted, 0 received, +3 errors, 100% packet loss, time 2014ms
pipe 3
mininet>
```

Figure 5: Ping Failure

The following commands can be used to manually install the required flows. This allows for packets coming at port 1 to be forwarded to port 2 and vice-versa.

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

This is can be verified by checking the flow-table using this command:

ovs-ofctl dump-flows s1

The output will be as shown in figure 6.

```
mininet@mininet-vm:~$ sudo ovs-ofctl add-flow s1 in_port=1,actions=output:2
mininet@mininet-vm:~$ sudo ovs-ofctl add-flow s1 in_port=2,actions=output:1
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=41.704s, table=0, n_packets=0, n_bytes=0, idle_age=41, in_port=1 actions=output:2
 cookie=0x0, duration=21.692s, table=0, n_packets=0, n_bytes=0, idle_age=21, in_port=2 actions=output:1
mininet@mininet-vm:~$
```

Figure 6: Adding flows to the switch table.

Now if we try the ping test using the following command, the output will be as in figure 7 with 0% packet loss.

mininet> h1 ping -c3 h2

```
mininet> h1 ping -c3 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=0.756 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.067 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.068 ms

--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2001ms
rtt min/avg/max/mdev = 0.067/0.297/0.756/0.324 ms
mininet>
```

Figure 7: Ping Test

CREATE A LEARNING SWITCH

We will be POX for implementing the rest of the controllers. POX is a Python-based SDN controller platform geared towards research and education.

We will start our implementation with the help of working of a basic hub. A hub is a device which just forwards the data it receives to all the ports apart from the port from which it received the data.

Firstly, let us kill the current controller and kill all the residue data that is there.

This can be done using the commands:

$ sudo killall controller
$ sudo mn -c

```
mininet@mininet-vm:~$ sudo mn -c
*** Removing excess controllers/ofprotocols/ofdatapaths/pings/noxes
killall controller ofprotocol ofdatapath ping nox_core lt-nox_core ovs-openflowd ovs-controller udpbwtest mnexec ivs 2> /dev/null
killall -9 controller ofprotocol ofdatapath ping nox_core lt-nox_core ovs-openflowd ovs-controller udpbwtest mnexec ivs 2> /dev/null
pkill -9 -f "sudo mnexec"
*** Removing junk from /tmp
rm -f /tmp/vconn* /tmp/vlogs* /tmp/*.out /tmp/*.log
*** Removing old X11 tunnels
*** Removing excess kernel datapaths
ps ax | egrep -o 'dp[0-9]+' | sed 's/dp/nl:/'
***  Removing OVS datapaths
ovs-vsctl --timeout=1 list-br
ovs-vsctl --if-exists del-br s1
ovs-vsctl --timeout=1 list-br
*** Removing all links of the pattern foo-ethX
ip link show | egrep -o '([-_.[:alnum:]]+-eth[[:digit:]]+)'
( ip link del s1-eth1;ip link del s1-eth2;ip link del s1-eth3 ) 2> /dev/null
ip link show
*** Killing stale mininet node processes
pkill -9 -f mininet:
*** Shutting down stale tunnels
pkill -9 -f Tunnel=Ethernet
pkill -9 -f .ssh/mn
rm -f ~/.ssh/mn/*
*** Cleanup complete.
mininet@mininet-vm:~$
```

Figure 8: Clearing all the processes.

Once this is done let us create the topology using the following command:

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

```
Setting remote controller to 127.0.0.1:6653
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1)
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet>
```

Figure 9: Topology for the network

Now, we can run the given source code for basic hub behavior using:

$./pox.py log.level --DEBUG misc.of_tutorial

The controller will running as shown below in figure 10:

```
mininet@mininet-vm:~$ cd
mininet@mininet-vm:~$ cd pox
mininet@mininet-vm:~/pox$ ./pox.py log.level --DEBUG misc.of_tutorial
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
DEBUG:core:POX 0.2.0 (carp) going up...
DEBUG:core:Running on CPython (2.7.6/Oct 26 2016 20:32:47)
DEBUG:core:Platform is Linux-4.2.0-27-generic-i686-with-Ubuntu-14.04-trusty
INFO:core:POX 0.2.0 (carp) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
```

Figure 10: Running the controller

Once the controller has been setup, verify the behavior of the hub using **tcpdump.**

Keep Xming on and open Xterm using:

Xterm h1 h2 h3

```
mininet> xterm h1 h2 h3
mininet> xterm h1 h2 h3
mininet>
```

```
"Node: h1"
root@mininet-vm:~
```

```
"Node: h2"
root@mininet-vm:~#
```

```
"Node: h3"
root@mininet-vm:~#
```

Figure 11: Xterm

Then use tcpdump to set up host 2 and host 3.

tcpdump -XX -n -i h2-eth0
tcpdump -XX -n -i h3-eth0

```
"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 262144 bytes
```

```
"Node: h3"
root@mininet-vm:~# tcpdump -XX -n -i h3-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h3-eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
```

Figure 12: Xterm working

Now, try to ping from h1 in the xterm of h1:

ping -c1 10.0.0.2

This packets now go to controller, and later are flooded out all directions except the one that it receives from. This can be seen by observing identical ARP and ICMP messages in both xterms running tcpdump, for h2 & h3, as shown in figure 13.

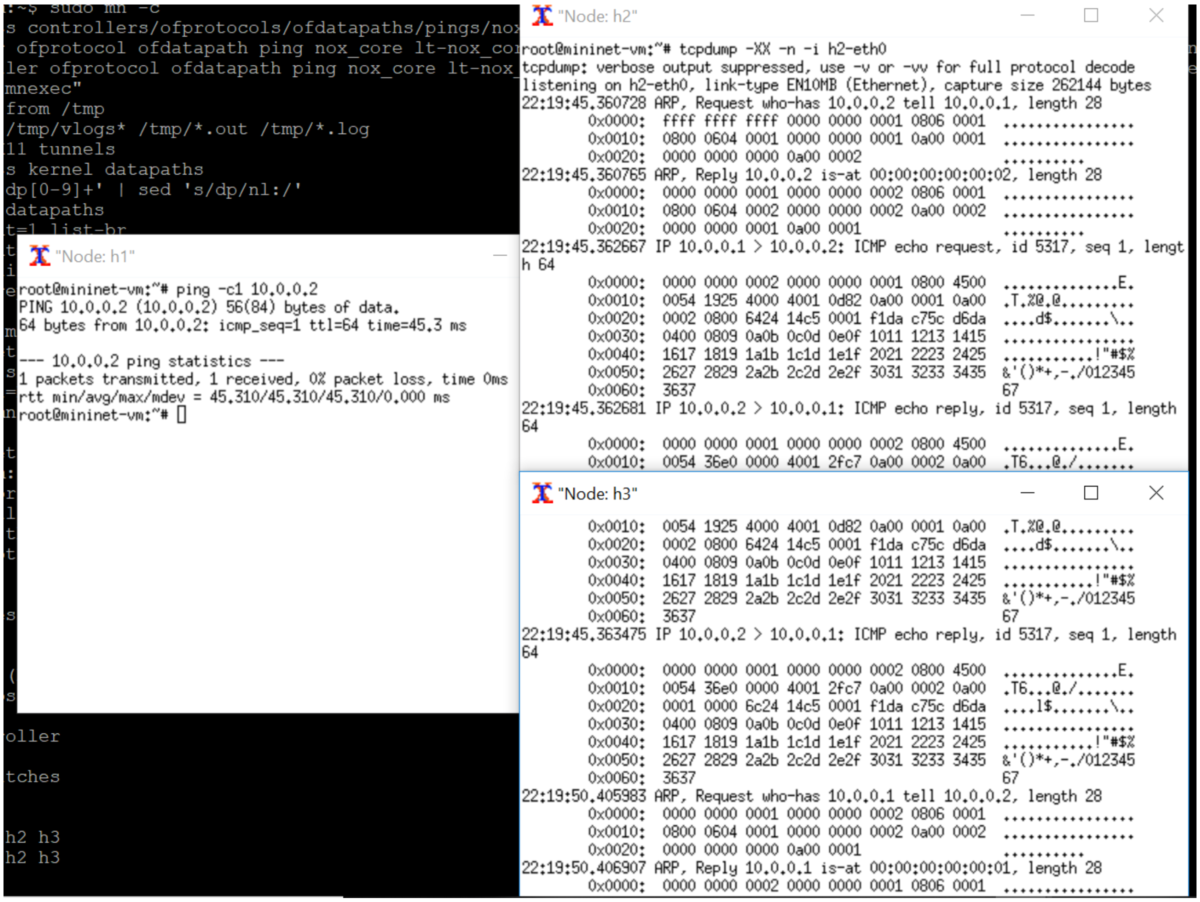


Figure 13: Testing of hub behavior.

Then, we do the experiment when the non-existing host doesn't reply by typing the command below. We know that 10.0.0.5 does not exist.

ping -c1 10.0.0.5

```
root@mininet-vm:~# ping -c1 10.0.0.5
PING 10.0.0.5 (10.0.0.5) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable

--- 10.0.0.5 ping statistics ---
1 packets transmitted, 0 received, +1 errors, 100% packet loss, time 0ms

root@mininet-vm:~# 
```

Figure 14: Testing hub behavior.

Benchmark Hub Controller w/iperf

Now let us check the reachability of the controller:

mininet> pingall

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

Figure 15: Pingall

We can see that all the packets have been received and hence we can go ahead and test the bandwidth with iperf:

minitet> iperf

```
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h3
*** Results: ['16.9 Mbits/sec', '17.9 Mbits/sec']
mininet>
```

Figure 16: IPERF

SWITCH CONTROLLER

Now, let us write the code for the behavior of a switch as we know that our hub is working properly.

We call the **act_like_switch()** function rather than the **act_like_hub()** function. We need to write our code in this function with packet as an argument which resembles an incoming packet to the switch.

Once the code is written we need to test the functionality of a switch. A switch only floods the packet to all interfaces only if it does not know the MAC address of the destination host. So, we need to store the host address and its MAC address of an incoming packet in a switch table for its future reference which emulates the learning switch functionality.

First we will run the controller. We have named our file p1_final.py as the code for the switch controller. We start the controller by:

$./pox.py log.level --DEBUG misc.p1_final

The controller starts, and this can be seen in figure 17.

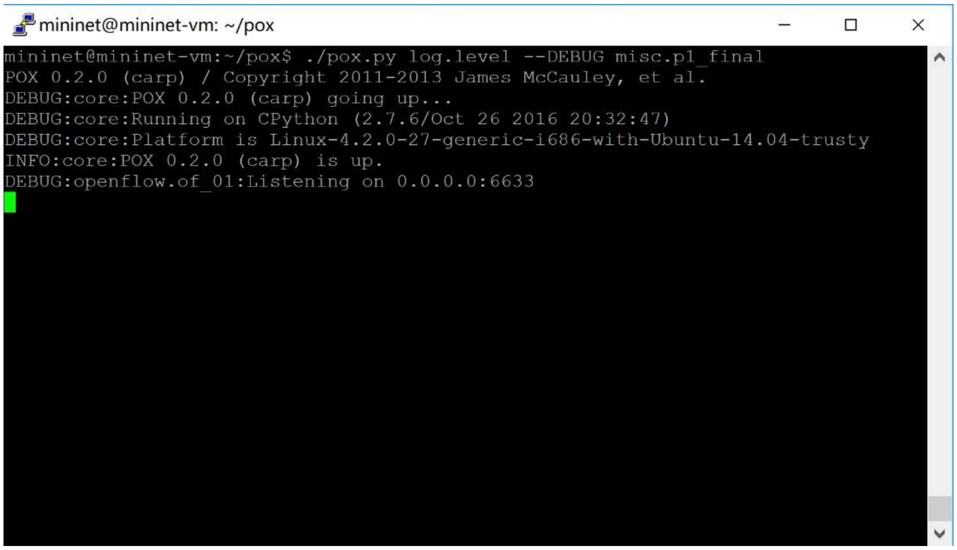


Figure 17: Run the controller successfully.

To verify the functionality of the switch, we follow the same method as we did for the hub. We use h2 and h3 in tcpdump modes and we then ping them from h1.

First let us ping h2 from h1 first. Ideally the switch should send ARP broadcasts to all interfaces as it does not know anything initially.

ping -c1 10.0.0.2

The results are as expected as shown in figure 18.

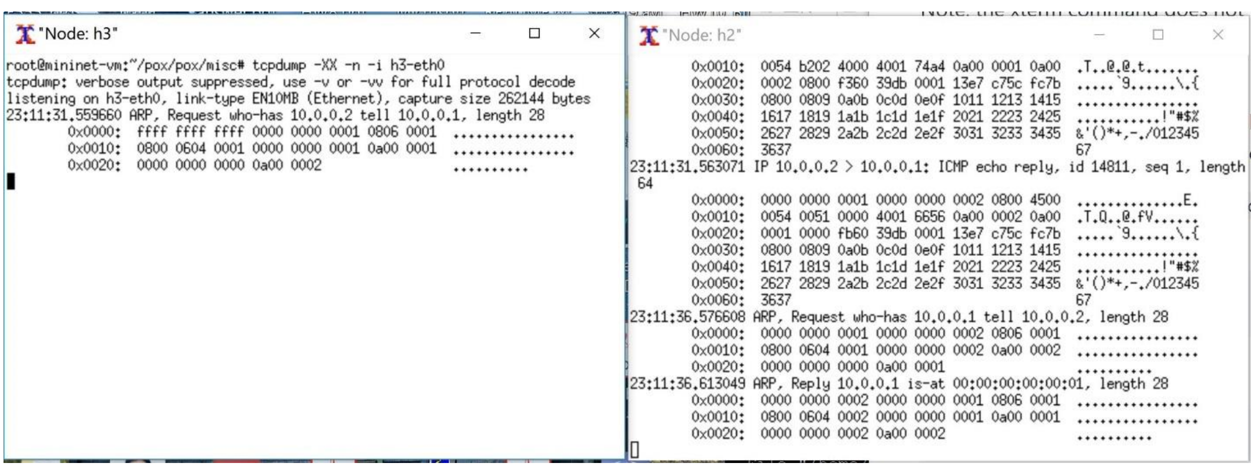


Figure 18: Results of the first ping.

Now let us ping h2 again from h1. Ideally the switch should have learnt about the MAC address of h2 and it should directly send the packets and not send anything to h3.

ping -c1 10.0.0.2

The results are as expected as shown in figure 19.

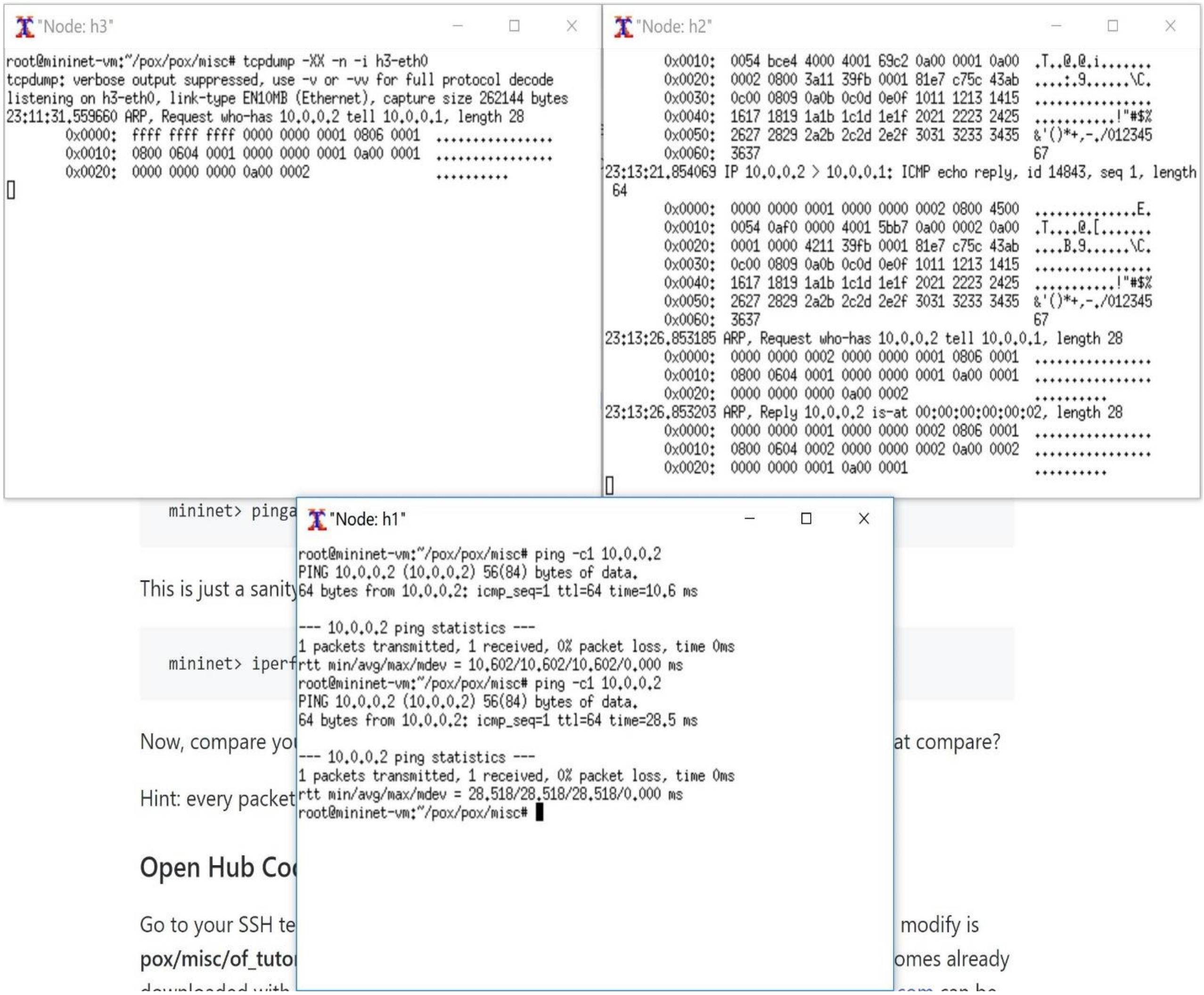


Figure 19: Results of the second ping.

Now let us test pinging an address that is not in the network. Ideally the switch must send ARP packets to find the MAC address of the given host address and then give. This is seen in figure 20.

ping -c1 10.0.0.5

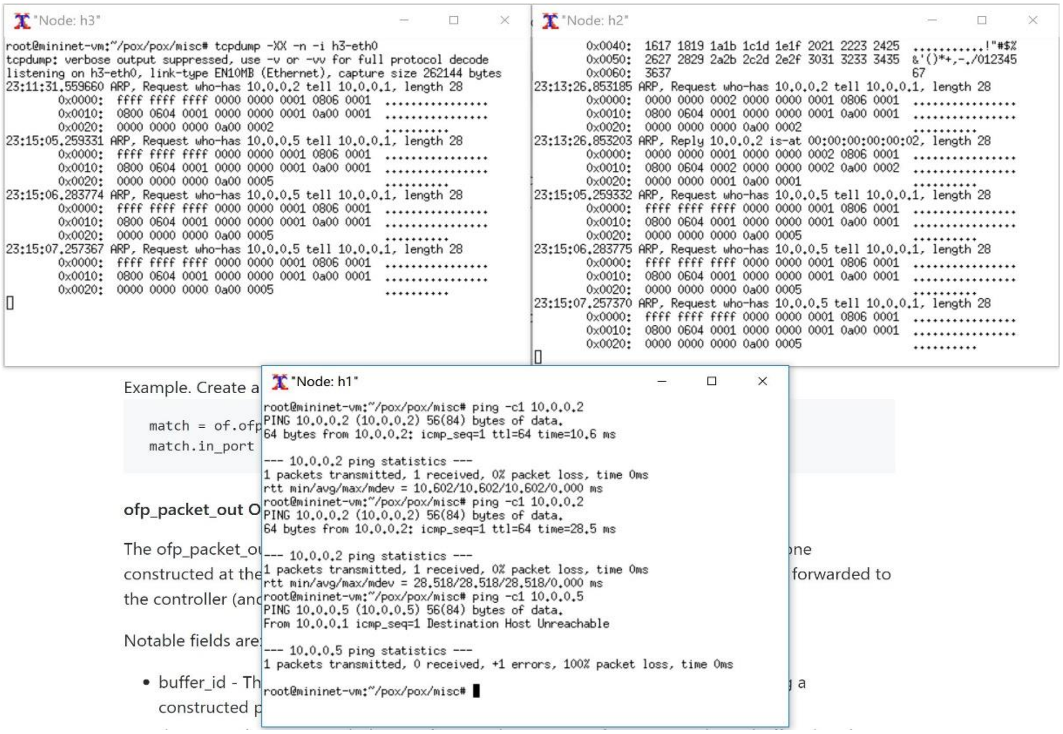


Figure 20: Testing the switch controller.

The reachability and the bandwidth can be checked as mentioned before and the results are as shown in figure 21.

mininet> pingall

mininet> iperf

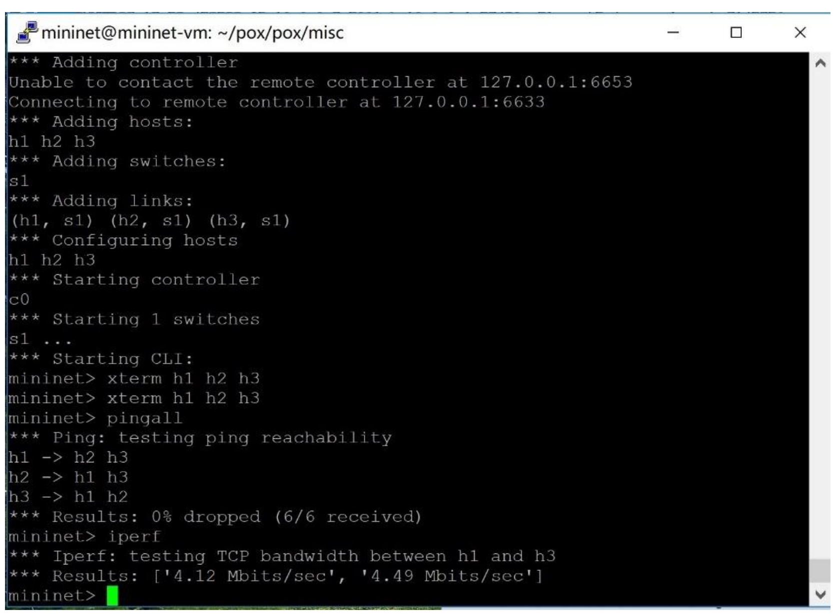


Figure 21: PINGALL and IPERF.

ROUTER EXERCISE

In this exercise we show that a layer 3 switch or a router like functionality can implemented using OpenFlow. Here we create a router controller for a topology given by:

sudo mn --custom mytopo.py --topo mytopo --mac –switch ovsk --controller remote

which generates a topology as shown in figure 22.

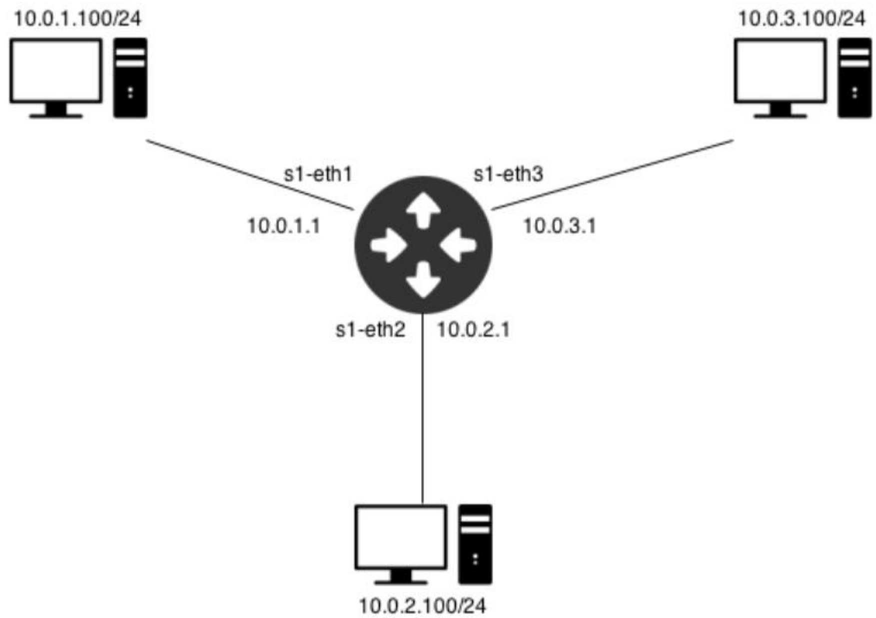


Figure 22: Topology for router exercise.

The topology is created by mininet as shown in figure 23.

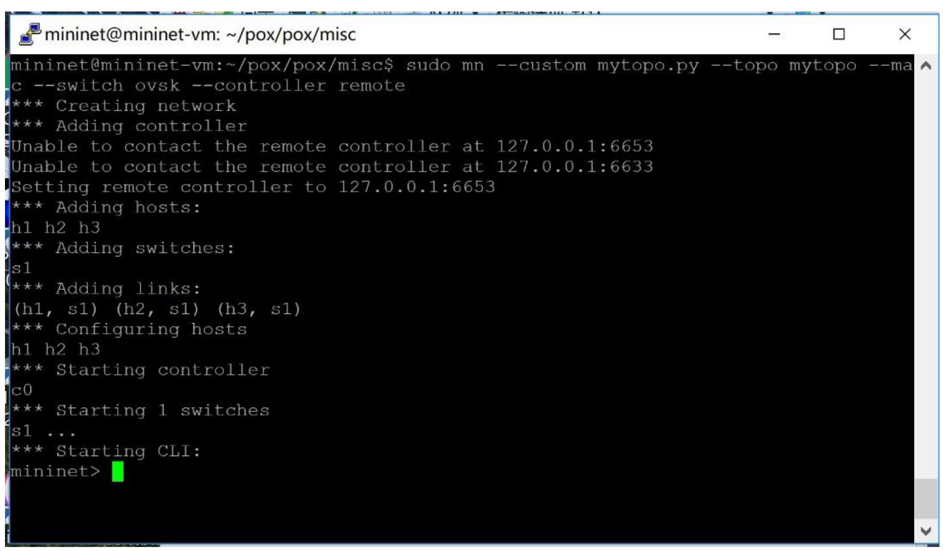
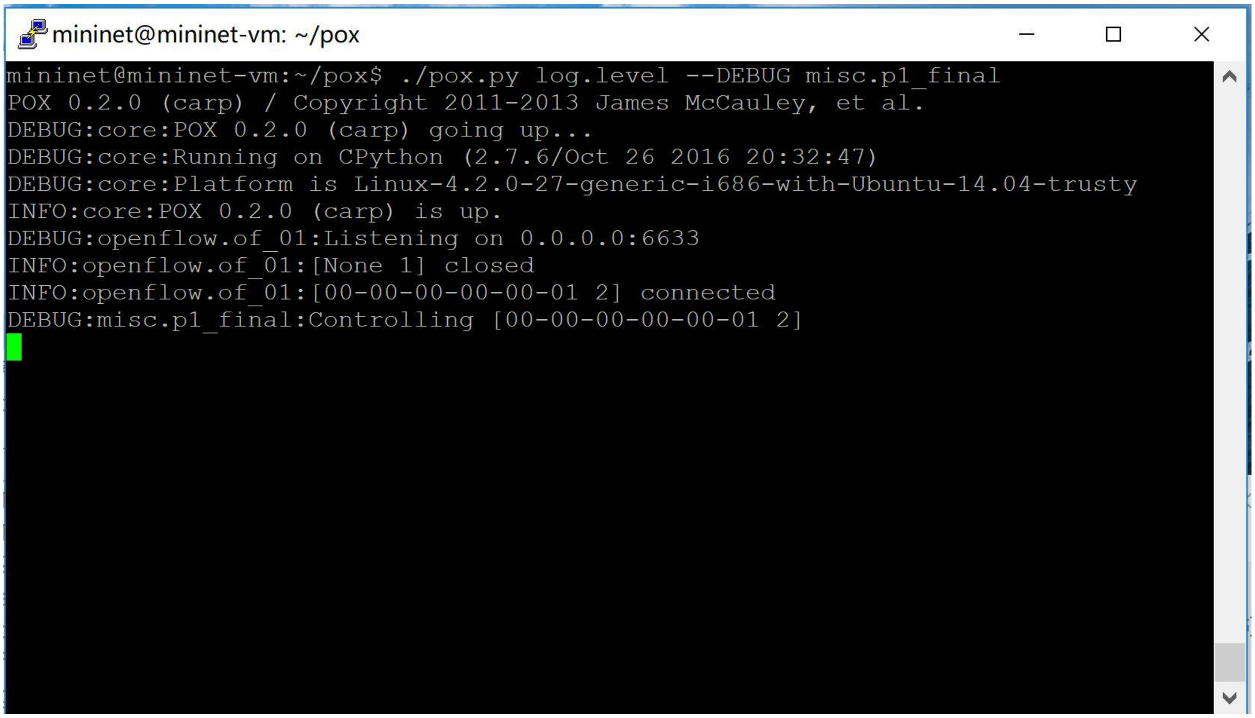


Figure 23: Topology on mininet.

We implemented a new function called **act_like_router()** and we called this function for the functionality of the controller. We named the file p1_final.py and the controller is successfully running as shown in figure 24.

$./pox.py log.level --DEBUG misc.p1_final



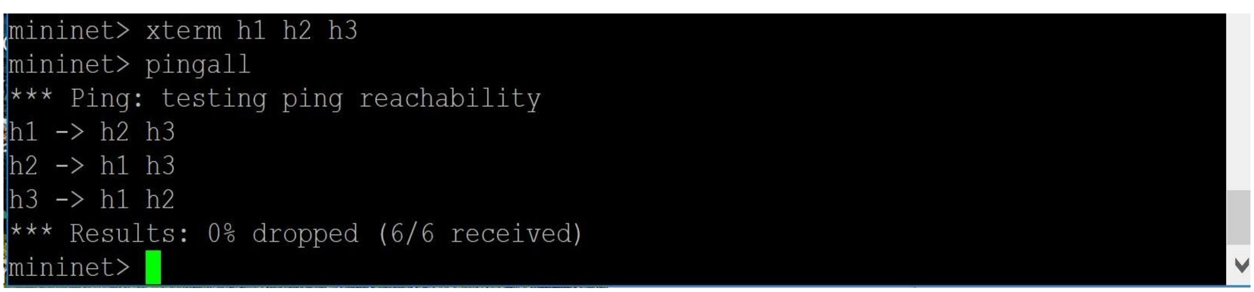
```
mininet@mininet-vm: ~/pox
mininet@mininet-vm:~/pox$ ./pox.py log.level --DEBUG misc.p1_final
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
DEBUG:core:POX 0.2.0 (carp) going up...
DEBUG:core:Running on CPython (2.7.6/Oct 26 2016 20:32:47)
DEBUG:core:Platform is Linux-4.2.0-27-generic-i686-with-Ubuntu-14.04-trusty
INFO:core:POX 0.2.0 (carp) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[None 1] closed
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
DEBUG:misc.p1_final:Controlling [00-00-00-00-00-01 2]
```

Figure 24: Running the controller for the router exercise.

Once the controller is setup, we run the ping test to see if all the addresses are reachable.

The ping test yields the results as shown in figure 25 which is correct functionality.

mininet> pingall



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

Figure 25: Pingall

Now, let us test what happens when an unknown address is pinged that does not exist in the network. Ideally the functionality of a router is to reply with an ICMP message saying that the destination is not reachable. This can be confirmed in figure 26.

mininet> h1 ping -c1 100.100.100.100

"Node: h1"

```
00:05:31.329070 ARP, Reply 10.0.1.1 is-at ef:ef:ef:ef:ef:ef, length 28
        0x0000:  0000 0000 0001 efef efef efef 0806 0001  ................
        0x0010:  0800 0604 0002 efef efef efef 0a00 0101  ................
        0x0020:  0000 0000 0001 0a00 0164                 .........d
00:05:43.260727 IP 10.0.1.100 > 100.100.100.100: ICMP echo request, id 3221, seq
 1, length 64
        0x0000:  efef efef efef 0000 0000 0001 0800 4500  ..............E.
        0x0010:  0054 2e21 4000 4001 385c 0a00 0164 6464  .T.!@.@.8\...ddd
        0x0020:  6464 0800 fb1b 0c95 0001 c7f3 c75c 72fa  dd...........\r.
        0x0030:  0300 0809 0a0b 0c0d 0e0f 1011 1213 1415  ................
        0x0040:  1617 1819 1a1b 1c1d 1e1f 2021 2223 2425  ...........!"#$%
        0x0050:  2627 2829 2a2b 2c2d 2e2f 3031 3233 3435  &'()*+,-./012345
        0x0060:  3637                                     67
00:05:43.303398 IP 100.100.100.100 > 10.0.1.100: ICMP net 100.100.100.100 unreac
hable, length 92
        0x0000:  0000 0000 0001 efef efef efef 0800 4500  ..............E.
        0x0010:  0070 f0bd 0000 4001 b5a3 6464 6464 0a00  .p....@...dddd..
        0x0020:  0164 0300 fcff 0000 0000 4500 0054 2e21  .d........E..T.!
        0x0030:  4000 4001 385c 0a00 0164 6464 6464 0800  @.@.8\...ddddd..
        0x0040:  fb1b 0c95 0001 c7f3 c75c 72fa 0300 0809  .........\r.....
        0x0050:  0a0b 0c0d 0e0f 1011 1213 1415 1617 1819  ................
        0x0060:  1a1b 1c1d 1e1f 2021 2223 2425 2627 2829  .......!"#$%&'()
        0x0070:  2a2b 2c2d 2e2f 3031 3233 3435 3637       *+,-./01234567
```

```
mininet> h1 ping -c1 100.100.100.100
PING 100.100.100.100 (100.100.100.100) 56(84) bytes of data.
From 100.100.100.100 icmp_seq=1 Destination Net Unreachable

--- 100.100.100.100 ping statistics ---
1 packets transmitted, 0 received, +1 errors, 100% packet loss, time 0ms
```

Figure 26: ICMP functionality

When we run the normal ping test between h1 and h2, there needs to be ICMP Echo requests and ICMP Echo replies between both the host and should go through the router. This can be verified in figure 27.

There will also be ARP packets as the router does not know about the MAC addresses of the host. These are stored in an IP to MAC table and is implemented in the controller code.

mininet> h1 ping -c1 10.0.2.100

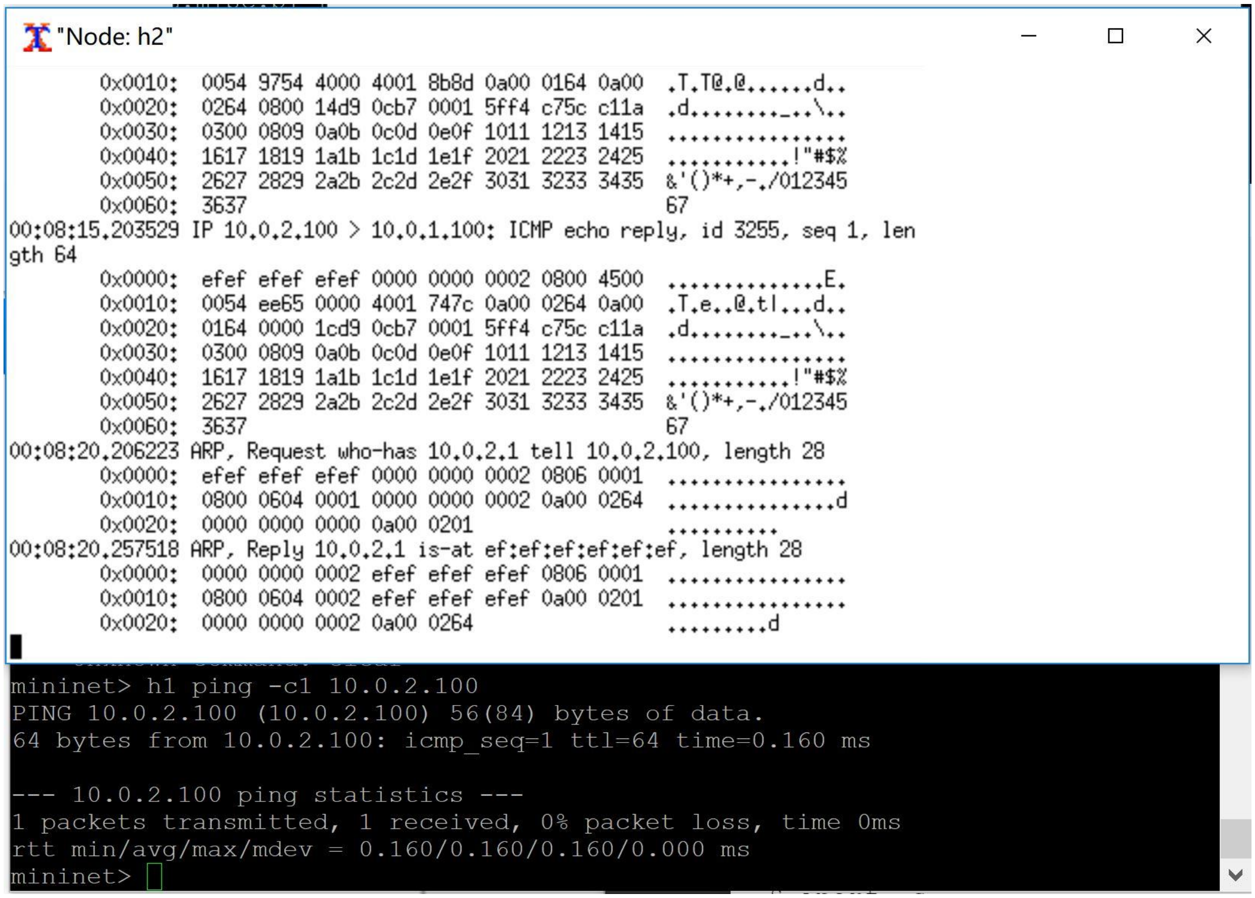


Figure 27: ICMP Echo functionality with ping test.

The bandwidth between h1 and h3 was tested using TCP connections and running. The results are shown in figure 28.

mininet> iperf

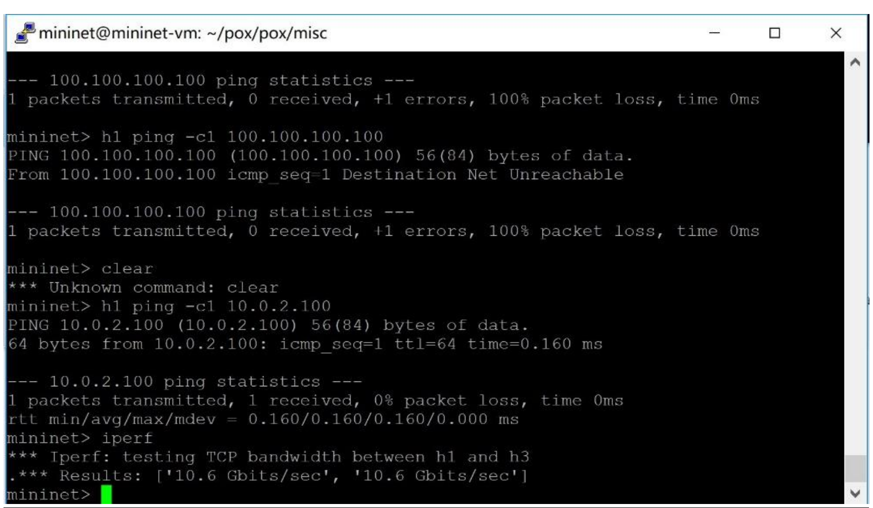


Figure 28: IPERF

ADVANCED TOPOLOGY

In this exercise the complexity of the topology is increased, and one controller is used to control 2 routers than 1 router. The topology is different as shown in figure 29.

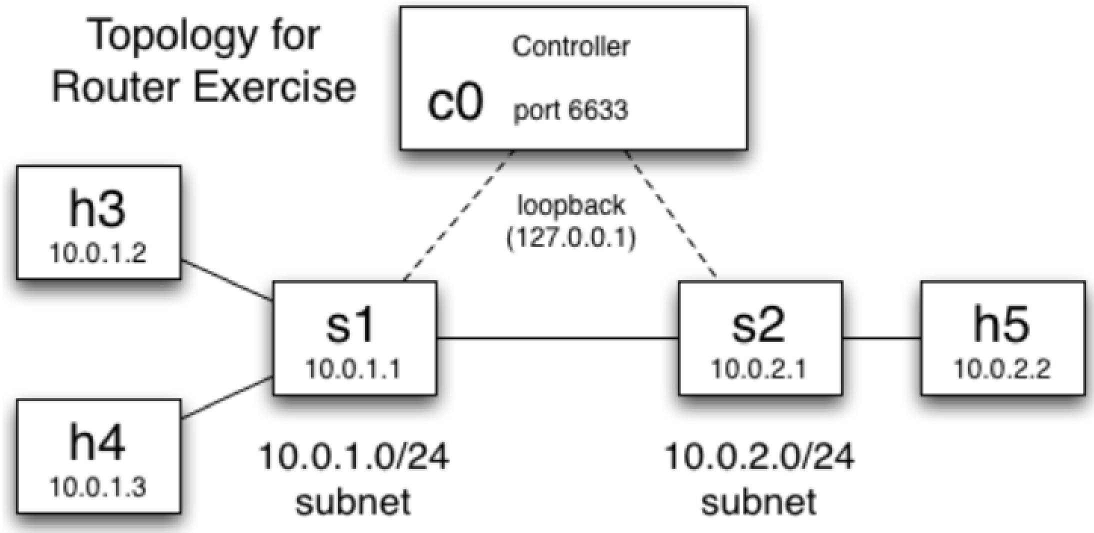


Figure 29: Advanced topology with 2 routers

The challenge here is to maintain 2 routers with one controller and address different subnets. First let us create the topology using

sudo mn --custom advtopo.py --topo advtopo --mac –switch ovsk --controller remote

Mininet creates a topology as shown in figure 30.

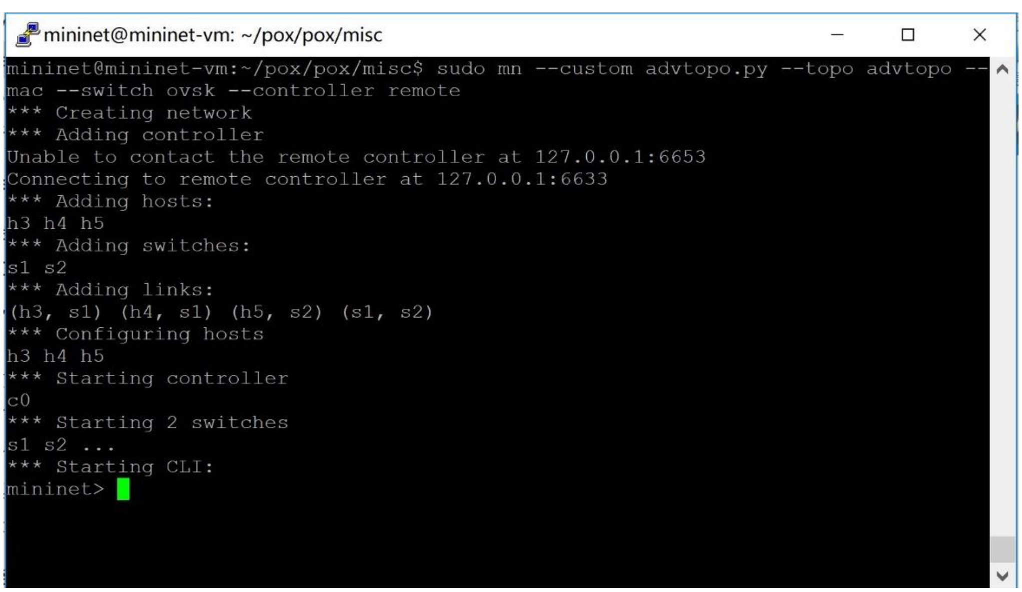


Figure 30: Topology on mininet.

We implemented a new function called **act_like_router()** and we called this function for the functionality of the controller. We named the file p2_final.py and the controller is successfully running as shown in figure 31.

$./pox.py log.level --DEBUG misc.p2_final

mininet@mininet-vm: ~/pox

```
mininet@mininet-vm:~/pox$ ./pox.py log.level --DEBUG misc.p2_final
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
DEBUG:core:POX 0.2.0 (carp) going up...
DEBUG:core:Running on CPython (2.7.6/Oct 26 2016 20:32:47)
DEBUG:core:Platform is Linux-4.2.0-27-generic-i686-with-Ubuntu-14.04-trusty
INFO:core:POX 0.2.0 (carp) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[None 1] closed
INFO:openflow.of_01:[00-00-00-00-00-01 3] connected
DEBUG:misc.p2_final:Controlling [00-00-00-00-00-01 3]
INFO:openflow.of_01:[00-00-00-00-00-02 2] connected
DEBUG:misc.p2_final:Controlling [00-00-00-00-00-02 2]
```

Figure 31: The controller starts running.

Once the controller is setup, we run the ping test to see if all the addresses are reachable.

The ping test yields the results as shown in figure 32 which is correct functionality.

mininet> pingall

```
mininet> pingall
*** Ping: testing ping reachability
h3 -> h4 h5
h4 -> h3 h5
h5 -> h3 h4
*** Results: 0% dropped (6/6 received)
mininet>
```

Figure 32: Pingall

Now, let us test what happens when an unknown address is pinged that does not exist in the network. Ideally the functionality of a router is to reply with an ICMP message saying that the destination is not reachable. This can be confirmed in figure 33.

mininet> h3 ping -c1 100.100.100.100

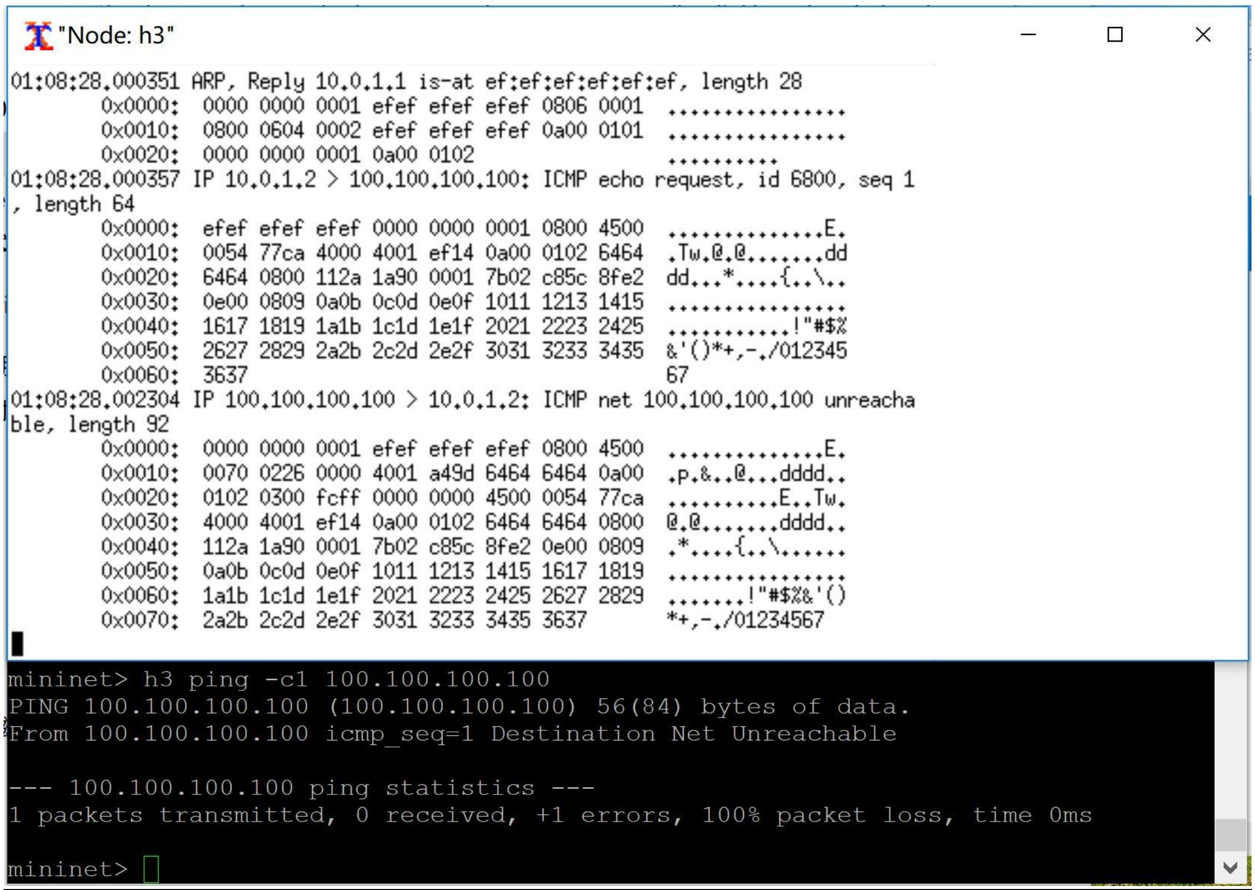


Figure 33: Destination Unreachable – ICMP.

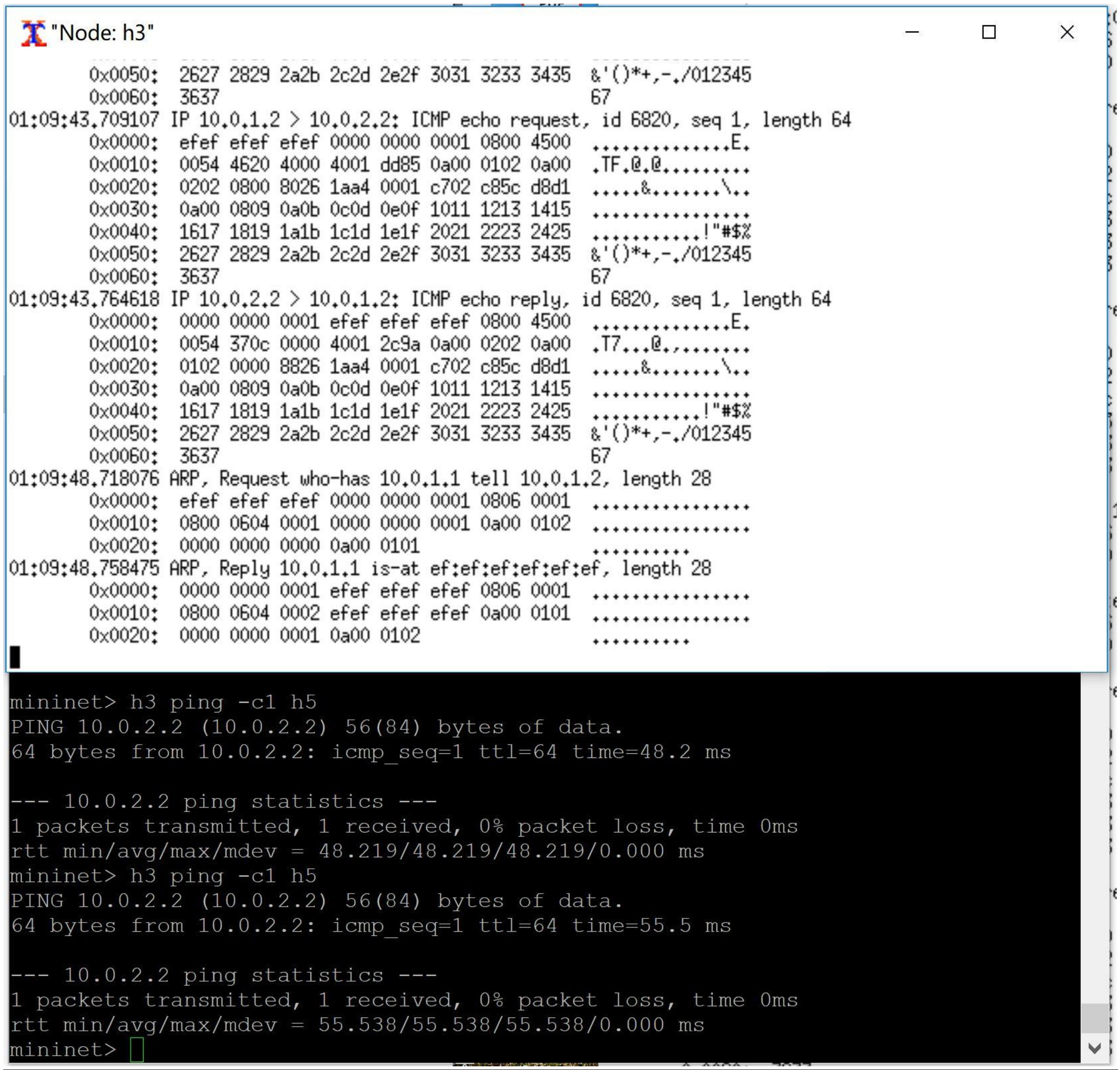
When we run the normal ping test between h3 and h4, there needs to be ICMP Echo requests and ICMP Echo replies between both the host and should go through the routers. This can be verified in figure 34.

There will also be ARP packets as the router does not know about the MAC addresses of the host. These are stored in an IP to MAC table and is implemented in the controller code.

But when you ping the same nodes again, there will be only ICMP Echo messages and no ARP as the routers have already learnt the MAC addresses and have made an entry into their tables.

mininet> h3 ping -c1 h5

mininet> h3 ping -c1 h5



```
"Node: h3"
        0x0050:  2627 2829 2a2b 2c2d 2e2f 3031 3233 3435  &'()*+,-./012345
        0x0060:  3637                                     67
01:09:43.709107 IP 10.0.1.2 > 10.0.2.2: ICMP echo request, id 6820, seq 1, length 64
        0x0000:  efef efef efef 0000 0000 0001 0800 4500  ..............E.
        0x0010:  0054 4620 4000 4001 dd85 0a00 0102 0a00  .TF.@.@.........
        0x0020:  0202 0800 8026 1aa4 0001 c702 c85c d8d1  .....&.......\..
        0x0030:  0a00 0809 0a0b 0c0d 0e0f 1011 1213 1415  ................
        0x0040:  1617 1819 1a1b 1c1d 1e1f 2021 2223 2425  ...........!"#$%
        0x0050:  2627 2829 2a2b 2c2d 2e2f 3031 3233 3435  &'()*+,-./012345
        0x0060:  3637                                     67
01:09:43.764618 IP 10.0.2.2 > 10.0.1.2: ICMP echo reply, id 6820, seq 1, length 64
        0x0000:  0000 0000 0001 efef efef efef 0800 4500  ..............E.
        0x0010:  0054 370c 0000 4001 2c9a 0a00 0202 0a00  .T7...@.,.......
        0x0020:  0102 0000 8826 1aa4 0001 c702 c85c d8d1  .....&.......\..
        0x0030:  0a00 0809 0a0b 0c0d 0e0f 1011 1213 1415  ................
        0x0040:  1617 1819 1a1b 1c1d 1e1f 2021 2223 2425  ...........!"#$%
        0x0050:  2627 2829 2a2b 2c2d 2e2f 3031 3233 3435  &'()*+,-./012345
        0x0060:  3637                                     67
01:09:48.718076 ARP, Request who-has 10.0.1.1 tell 10.0.1.2, length 28
        0x0000:  efef efef efef 0000 0000 0001 0806 0001  ................
        0x0010:  0800 0604 0001 0000 0000 0001 0a00 0102  ................
        0x0020:  0000 0000 0000 0a00 0101                 ..........
01:09:48.758475 ARP, Reply 10.0.1.1 is-at ef:ef:ef:ef:ef:ef, length 28
        0x0000:  0000 0000 0001 efef efef efef 0806 0001  ................
        0x0010:  0800 0604 0002 efef efef efef 0a00 0101  ................
        0x0020:  0000 0000 0001 0a00 0102                 ..........

mininet> h3 ping -c1 h5
PING 10.0.2.2 (10.0.2.2) 56(84) bytes of data.
64 bytes from 10.0.2.2: icmp_seq=1 ttl=64 time=48.2 ms

--- 10.0.2.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 48.219/48.219/48.219/0.000 ms
mininet> h3 ping -c1 h5
PING 10.0.2.2 (10.0.2.2) 56(84) bytes of data.
64 bytes from 10.0.2.2: icmp_seq=1 ttl=64 time=55.5 ms

--- 10.0.2.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 55.538/55.538/55.538/0.000 ms
mininet>
```

Figure 33: Ping test.

The packet exchange for the same can be seen in figure 34 in for h5.

"Node: h5"

```
root@mininet-vm:~/pox/pox/misc# tcpdump -XX -n -i h5-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h5-eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
01:09:37.412895 ARP, Request who-has 10.0.2.2 (ff:ff:ff:ff:ff:ff) tell 10.0.2.1,
 length 28
        0x0000:  ffff ffff ffff efef efef efef 0806 0001  ................
        0x0010:  0800 0604 0001 efef efef efef 0a00 0201  ................
        0x0020:  ffff ffff ffff 0a00 0202                ..........
01:09:37.412924 ARP, Reply 10.0.2.2 is-at 00:00:00:00:00:03, length 28
        0x0000:  efef efef efef 0000 0000 0003 0806 0001  ................
        0x0010:  0800 0604 0002 0000 0000 0003 0a00 0202  ................
        0x0020:  efef efef efef 0a00 0201                ..........
01:09:37.415101 IP 10.0.1.2 > 10.0.2.2: ICMP echo request, id 6817, seq 1, lengt
h 64
        0x0000:  0000 0000 0003 efef efef efef 0800 4500  ..............E.
        0x0010:  0054 41e3 4000 4001 e1c2 0a00 0102 0a00  .TA.@.@.........
        0x0020:  0202 0800 c449 1aa1 0001 c102 c85c 9fb1  .....I.......\..
        0x0030:  0500 0809 0a0b 0c0d 0e0f 1011 1213 1415  ................
        0x0040:  1617 1819 1a1b 1c1d 1e1f 2021 2223 2425  ...........!"#$%
        0x0050:  2627 2829 2a2b 2c2d 2e2f 3031 3233 3435  &'()*+,-./012345
        0x0060:  3637                                     67
01:09:37.415110 IP 10.0.2.2 > 10.0.1.2: ICMP echo reply, id 6817, seq 1, length
64
        0x0000:  efef efef efef 0000 0000 0003 0800 4500  ..............E.
        0x0010:  0054 3405 0000 4001 2fa1 0a00 0202 0a00  .T4...@./.......
        0x0020:  0102 0000 cc49 1aa1 0001 c102 c85c 9fb1  .....I.......\..
        0x0030:  0500 0809 0a0b 0c0d 0e0f 1011 1213 1415  ................
        0x0040:  1617 1819 1a1b 1c1d 1e1f 2021 2223 2425  ...........!"#$%
        0x0050:  2627 2829 2a2b 2c2d 2e2f 3031 3233 3435  &'()*+,-./012345
        0x0060:  3637                                     67
01:09:42.429711 ARP, Request who-has 10.0.2.1 tell 10.0.2.2, length 28
        0x0000:  efef efef efef 0000 0000 0003 0806 0001  ................
        0x0010:  0800 0604 0001 0000 0000 0003 0a00 0202  ................
        0x0020:  0000 0000 0000 0a00 0201                ..........
01:09:42.468883 ARP, Reply 10.0.2.1 is-at ef:ef:ef:ef:ef:ef, length 28
        0x0000:  0000 0000 0003 efef efef efef 0806 0001  ................
        0x0010:  0800 0604 0002 efef efef efef 0a00 0201  ................
        0x0020:  0000 0000 0003 0a00 0202                ..........
01:09:43.761625 IP 10.0.1.2 > 10.0.2.2: ICMP echo request, id 6820, seq 1, lengt
h 64
        0x0000:  0000 0000 0003 efef efef efef 0800 4500  ..............E.
        0x0010:  0054 4620 4000 4001 dd85 0a00 0102 0a00  .TF.@.@.........
        0x0020:  0202 0800 8026 1aa4 0001 c702 c85c d8d1  .....&.......\..
        0x0030:  0a00 0809 0a0b 0c0d 0e0f 1011 1213 1415  ................
        0x0040:  1617 1819 1a1b 1c1d 1e1f 2021 2223 2425  ...........!"#$%
        0x0050:  2627 2829 2a2b 2c2d 2e2f 3031 3233 3435  &'()*+,-./012345
        0x0060:  3637                                     67
01:09:43.761637 IP 10.0.2.2 > 10.0.1.2: ICMP echo reply, id 6820, seq 1, length
64
        0x0000:  efef efef efef 0000 0000 0003 0800 4500  ..............E.
        0x0010:  0054 370c 0000 4001 2c9a 0a00 0202 0a00  .T7...@.,.......
        0x0020:  0102 0000 8826 1aa4 0001 c702 c85c d8d1  .....&.......\..
        0x0030:  0a00 0809 0a0b 0c0d 0e0f 1011 1213 1415  ................
        0x0040:  1617 1819 1a1b 1c1d 1e1f 2021 2223 2425  ...........!"#$%
        0x0050:  2627 2829 2a2b 2c2d 2e2f 3031 3233 3435  &'()*+,-./012345
        0x0060:  3637                                     67
```

Figure 34: ICMP and ARP for h5.

FIREWALL

Here we are implementing a layer 4 operation based on the concept of working of the firewall. The job of the firewall is to block some connections that are considered malicious. This can be implemented here by blocking a few ports from connections and hence making sure no communication takes place between this port any other port it wants to associate with. For our demonstration let us consider port number ‘5001’ as the blocked port and we need to block any communication associated with this port. Let us use the first topology we used for the router exercise.

Our firewall code file name is firewall.py. Start the firewall controller using the command

$./pox.py log.level --DEBUG misc.p2_final

Open h2 and h3 using ***xterm h2 h3*** and use the following commands in h2 and h3 windows respectively:

$ iperf -s

$ iperf -c 10.0.2.100

The resulting output matched the expectation as shown in figure 35.

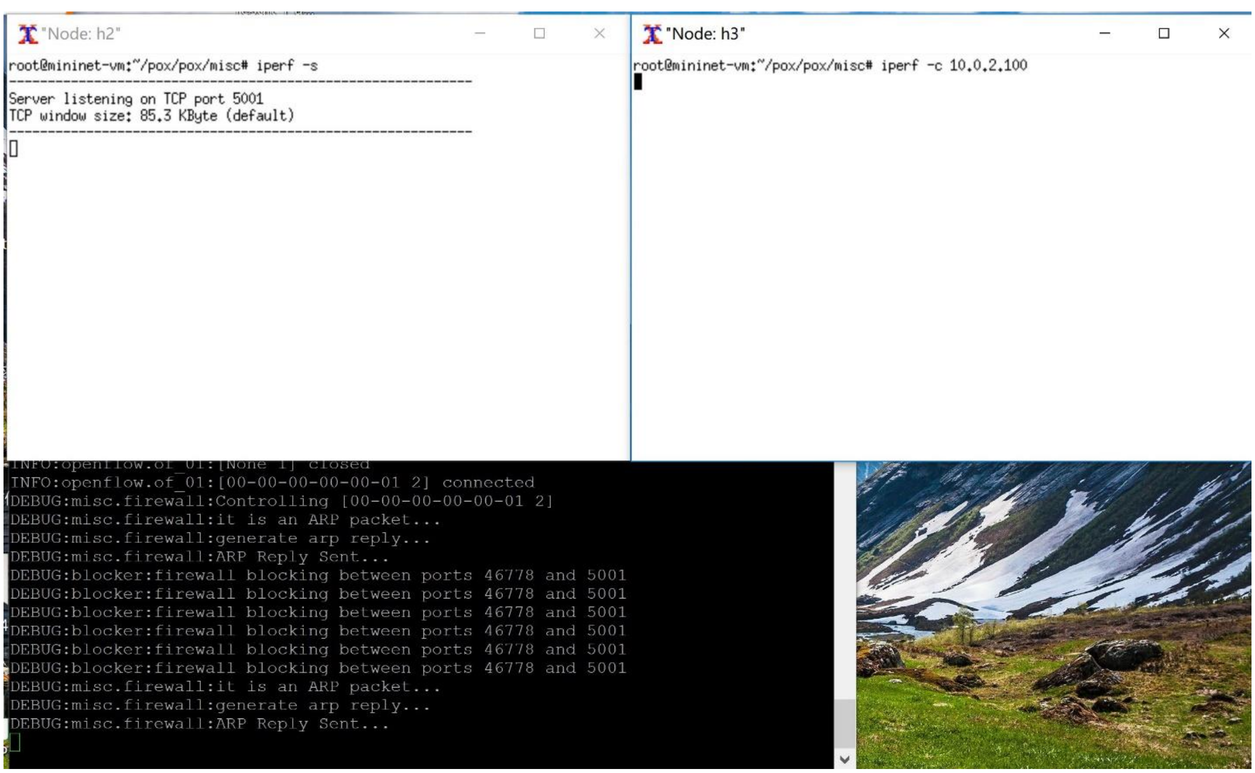


Figure 35: Firewall controller working and output.

CONCLUSION

This project helped us get a better understanding of the layer 2 and layer 3 switches in general and most importantly introduced us to the concept of Software Defined Networking which is a booming field in the current industry. We learnt how to implement SDN using POX libraries and gained a hands-on experience in python based OpenFlow programming. We were introduced to the concepts of virtual networks through mininet and learnt how to use it. We created a learning switch and a basic router in part 1 of the project. Later we worked with a more advanced topology in part 2 of the project. We implemented some layer 4 functionality as well with respect to port blocking and simulated the actions of a firewall. Overall it was a great learning experience for us aspiring network engineers.

REFERENCES

[1] https://github.com/mininet/openflow-tutorial/wiki/Set-up-Virtual-Machine

[2] https://github.com/mininet/openflow-tutorial/wiki/Learn-Development-Tools