EE 555 FINAL PROJECT: OPENFLOW MININET

SUBMITTED BY:

Amrutha Kumaraswamy (akumaras@usc.edu)

Anjana Nair (anjanana@usc.edu)

AIM:

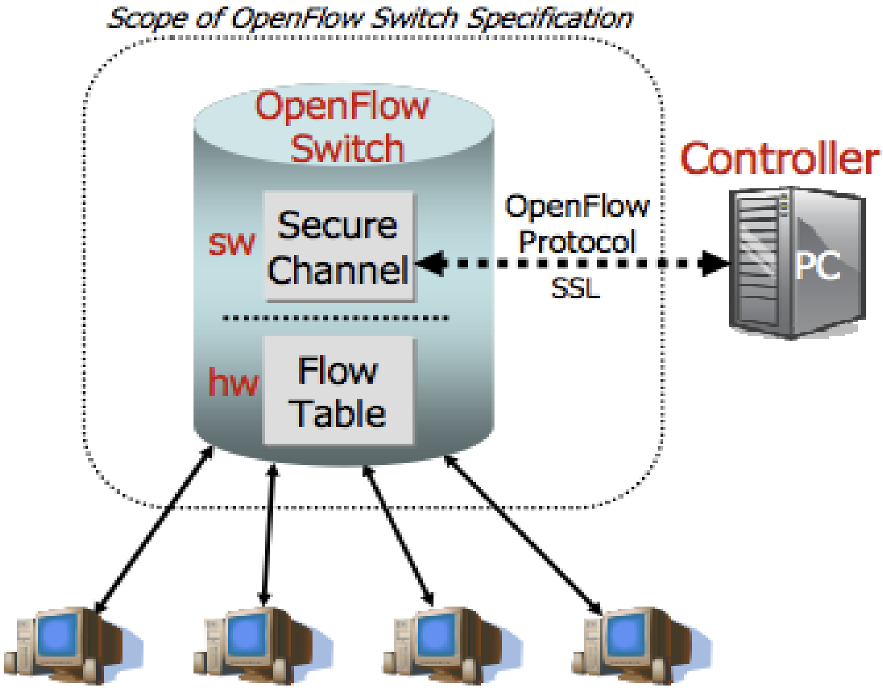
To use OpenFlow and Mininet tools to build a virtual SDN network. We learn the basics on using the tools and create a learning switch. After set up and observing how the hub works, we create a learning switch, then we build a static router implementation for a given topology using POX (a Python-based SDN controller platform).

INTRODUCTION:

Openflow is a communications protocol that gives access to the forwarding plane of a network switch or router over the network. It is used to govern the communication between a controller and the switch in a software defined network (SDN) environment. It is an open standard network protocol used to manage traffic between commercial Ethernet switches, routers and wireless access points.

OpenFlow enables software-defined networking (SDN) for programmable networks and is based on an Ethernet switch, with an internal flow-table and a standardized interface to add and remove flow entries. The controllers are distinct from the switches. This separation of the control from the forwarding allows for more sophisticated traffic management than is feasible using access control lists (ACLs) and routing protocols.

Compared to a conventional switch the OpenFlow switch separates the control path and the data path. The controller makes the routing decisions, while the data remains on the switch. It is through this protocol that the switch and the controller communicate, which is the basis of Software Define Networks.



MININET

Mininet is a software emulator for prototyping a large network on a single machine. It runs a collection of end-hosts, switches, routers, and links on a single Linux kernel. Mininet can be used to quickly create a realistic virtual network running actual kernel, switch and software application code on a personal computer. Mininet allows the user to quickly create, interact with, customize and share a software-defined network (SDN) prototype to simulate a network topology that uses Openflow switches.

A Mininet host behaves just like a real machine; you can ssh into it (if you start up sshd and bridge the network to your host) and run arbitrary programs (including anything that is installed on the underlying Linux system). In short, Mininet's virtual hosts, switches, links, and controllers are the real thing – they are just created using software rather than hardware – and for the most part their behavior is like discrete hardware elements.

IMPLEMENTATION

INSTALLING REQUIRED SOFTWARE

- A virtual machine image (OVF) is downloaded in the beginning

- A virtual box is installed and the OVF image is imported into the virtual box

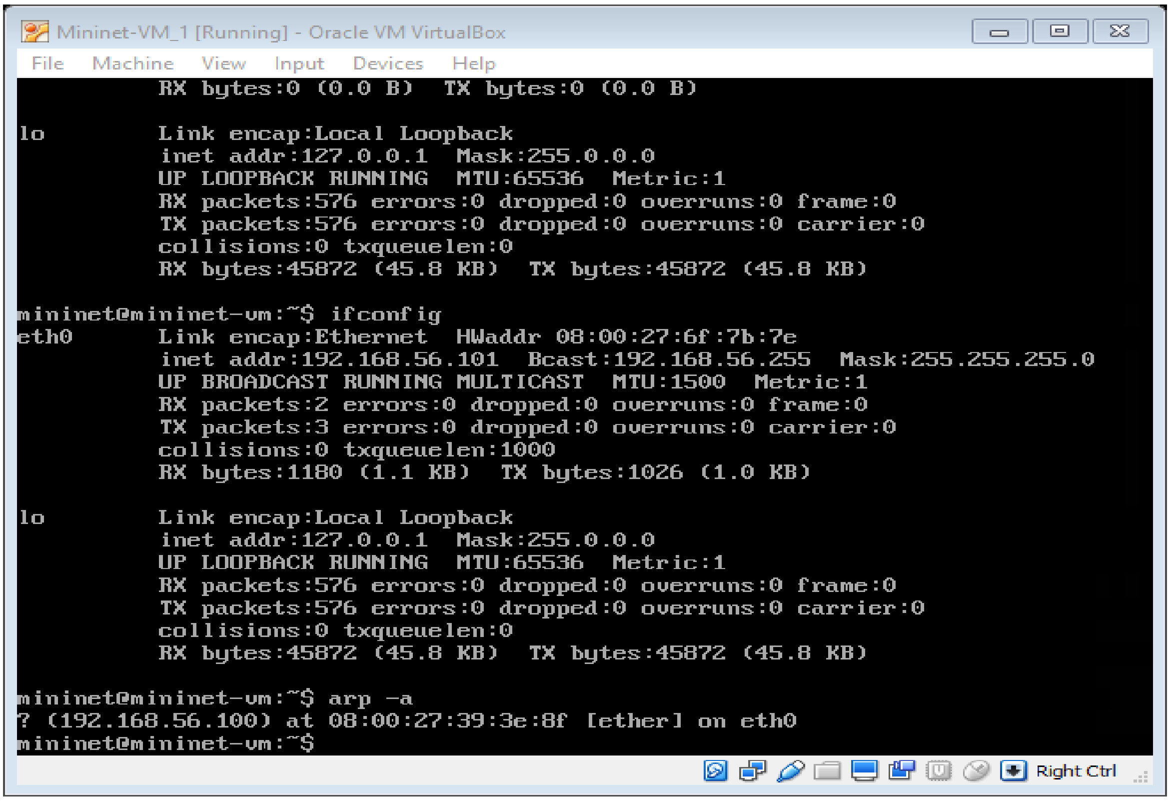
- Select the VM. Go to Settings tab. Go to Network ->Adapter 2. Select the Enable Adapter box and attach it to the “host-only-network”. Make sure the DHCP client is set

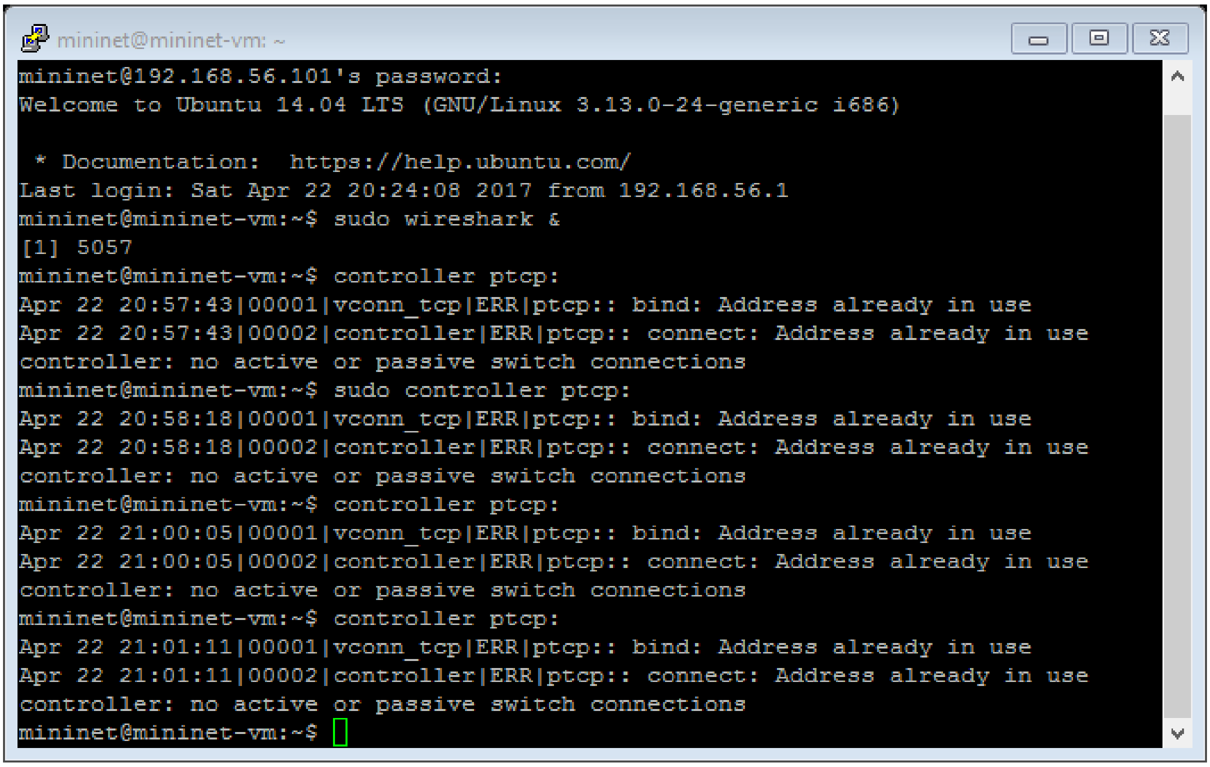
- Press the “**Start**” icon

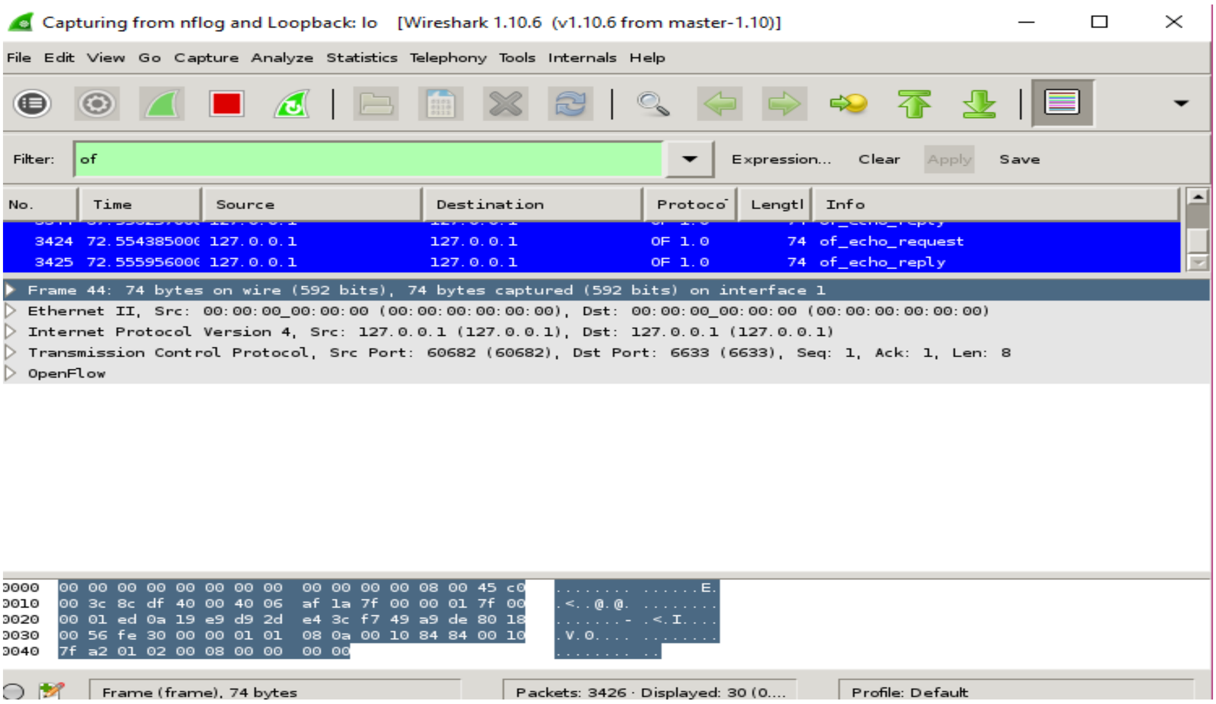
- In the VM console window, log in with the user name and password both as “**mininet**”

- An X11 session can be started in the VM console window by typing “**startx**”

- Add a host-only interface to your VM by typing the command “**sudo dhclient eth1**”







CONTROLLER USED: POX

POX is a Python-based SDN controller platform geared towards research and education. We are provided with starter code for a hub controller. After getting familiar with it, the provided hub was modified to act as an L2 learning switch. In this application, the switch will examine

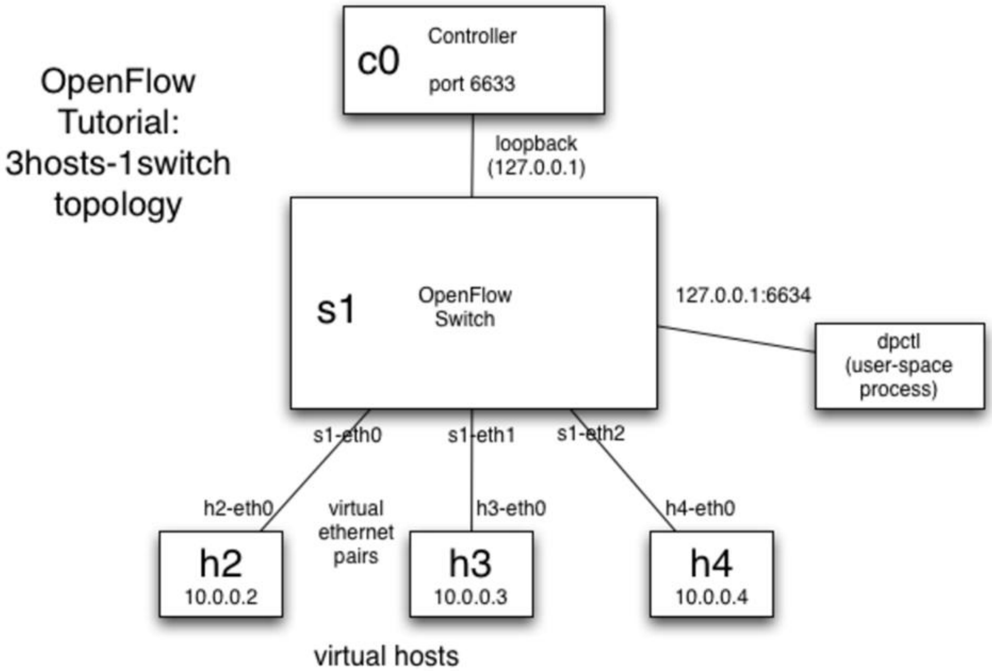
each packet and learn the source-port mapping. Thereafter, the source MAC address will be associated with the port. If the destination of the packet is already associated with some port, the packet will be sent to the given port, else it will be flooded on all ports
of the switch.
This is then converted into a flow-based switch, where seeing a packet with a known source and destination causes a flow entry to get pushed down. A mininet configuration below is loaded for both switch and hub using
command:

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

PART 1: HUB & SWITCH IMPLEMENTATION

By running the below code the python file with each time functioning as hub or
a switch: (by commenting out the line which calls the each of the functionality)

mininet@mininet-vm:~/pox$ ./pox.py log.level --DEBUG misc.of_switch

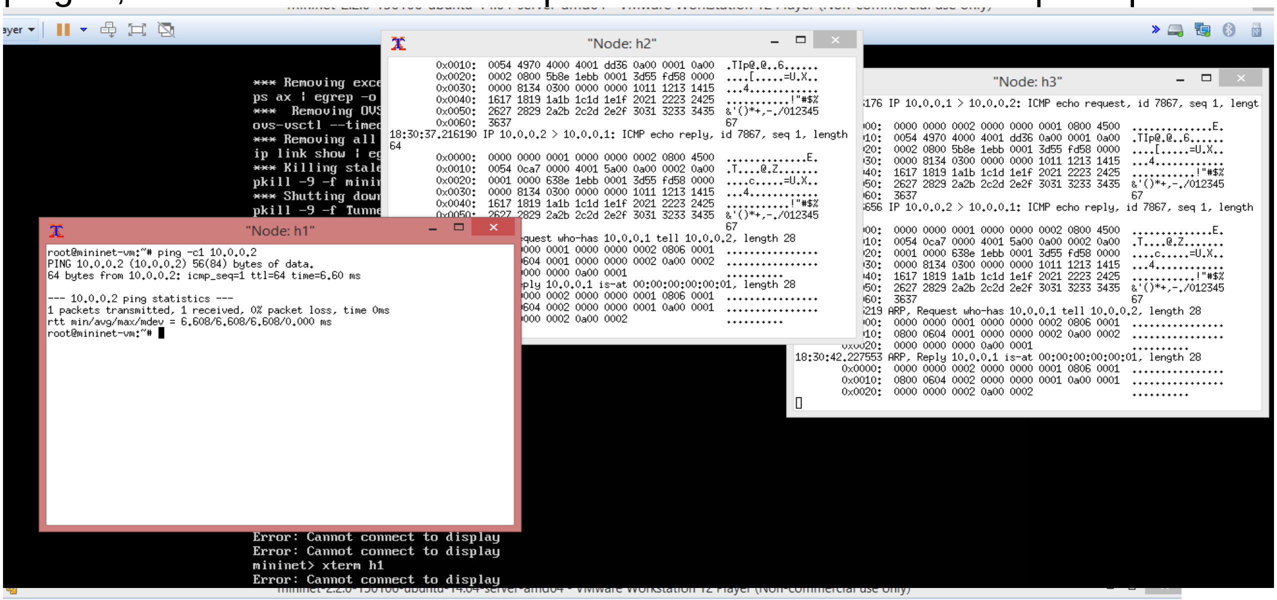


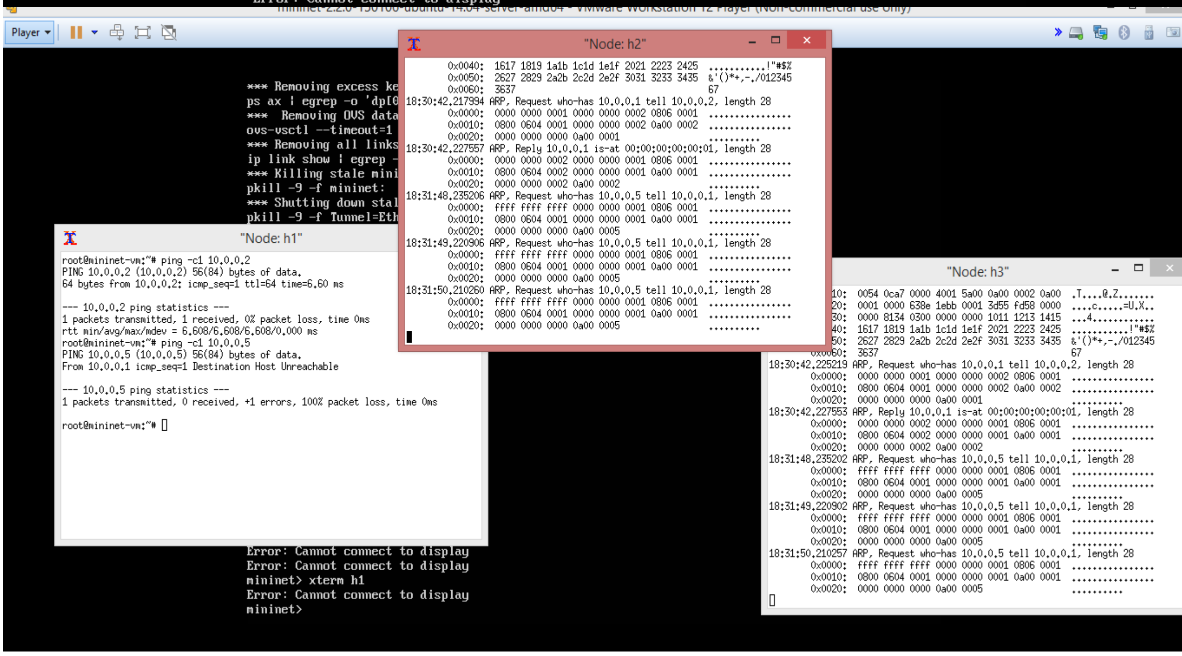
The controller performs the following actions:

1. It will construct ARP replies and forward them out the appropriate ports when Ethernet broadcasts are forwarded to it.
2. Once ARP has been handled, we need to handle routing for the static configuration.
3. It may receive ICMP echo (ping) requests for each switch, which will be responded to with ICMP network unreachable messages.

Hub Implementation

On running the hub code and using the xterm and terminal to verify the behavior we get the following output. Here all the incoming packets are sent out to all the ports. This is verified by tcpdump. If unknown host is pinged, then there are 3 ARP requests seen on each host in tcpdump.





Flow Mods for s1 :

mininet@mininet-vm: ~

```
     state:       0
     current:     10GB-FD COPPER
     speed: 10000 Mbps now, 0 Mbps max
 LOCAL(s1): addr:16:da:2b:98:b9:4f
     config:      0
     state:       0
     speed: 0 Mbps now, 0 Mbps max
OFPT_GET_CONFIG_REPLY (xid=0x4): frags=normal miss_send_len=0
mininet@mininet-vm:~$ ovs-ofctl dump-flows s1
ovs-ofctl: /var/run/openvswitch/s1.mgmt: failed to open socket (Permission denie
d)
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
mininet@mininet-vm:~$ ovs-ofctl add-flow s1 in_port=1,actions=output:2
ovs-ofctl: /var/run/openvswitch/s1.mgmt: failed to open socket (Permission denied)
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=36.737s, table=0, n_packets=0, n_bytes=0, idle_age=36, in_port=
1 actions=output:2
 cookie=0x0, duration=17.738s, table=0, n_packets=0, n_bytes=0, idle_age=17, in_port=
2 actions=output:1
mininet@mininet-vm:~$
```

mininet@mininet-vm: ~/pox

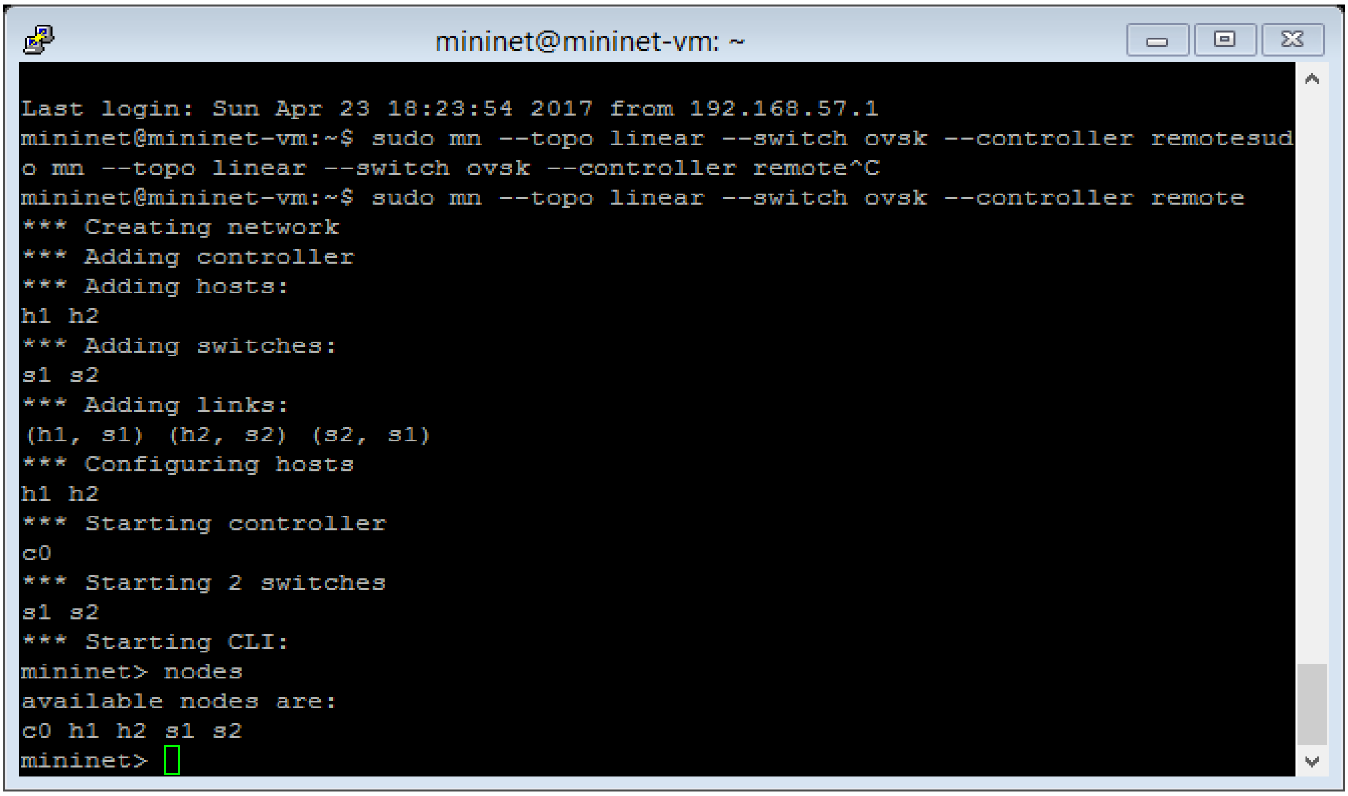
```
mininet@mininet-vm:~$ git clone http://github.com/noxrepo/pox
fatal: destination path 'pox' already exists and is not an empty directory.
mininet@mininet-vm:~$
mininet@mininet-vm:~$ cd pox/
mininet@mininet-vm:~/pox$ git branch
* carp
mininet@mininet-vm:~/pox$ git checkout betta
Branch betta set up to track remote branch betta from origin.
Switched to a new branch 'betta'
mininet@mininet-vm:~/pox$ git branch
* betta
  carp
mininet@mininet-vm:~/pox$ /pox.py log.level --DEBUG misc.of_tutorial
-bash: /pox.py: No such file or directory
mininet@mininet-vm:~/pox$ ./pox.py log.level --DEBUG misc.of_tutorial
POX 0.1.0 (betta) / Copyright 2011-2013 James McCauley, et al.
DEBUG:core:POX 0.1.0 (betta) going up...
DEBUG:core:Running on CPython (2.7.6/Mar 22 2014 22:59:56)
DEBUG:core:Platform is Linux-3.13.0-24-generic-x86_64-with-Ubuntu-14.04-trusty
INFO:core:POX 0.1.0 (betta) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
DEBUG:misc.of_tutorial:Controlling [00-00-00-00-00-01 1]
```

CREATING A LEARNING 3 SWITCH

SUPPORT MULTIPLE SWITCHES

Start mininet with a different topology.

In the Mininet console: **mininet> exit $ sudo mn --topo linear --switch ovsk --controller remote**



PART 1: ROUTER IMPLEMENTATION

In this exercise, we make a static layer-3 switch. It's not exactly a router, because it won't decrement the IP TTL and recomputed the checksum at each hop (so traceroute won't work). It will match on masked IP prefix ranges, just like a real router.

From RFC 1812:

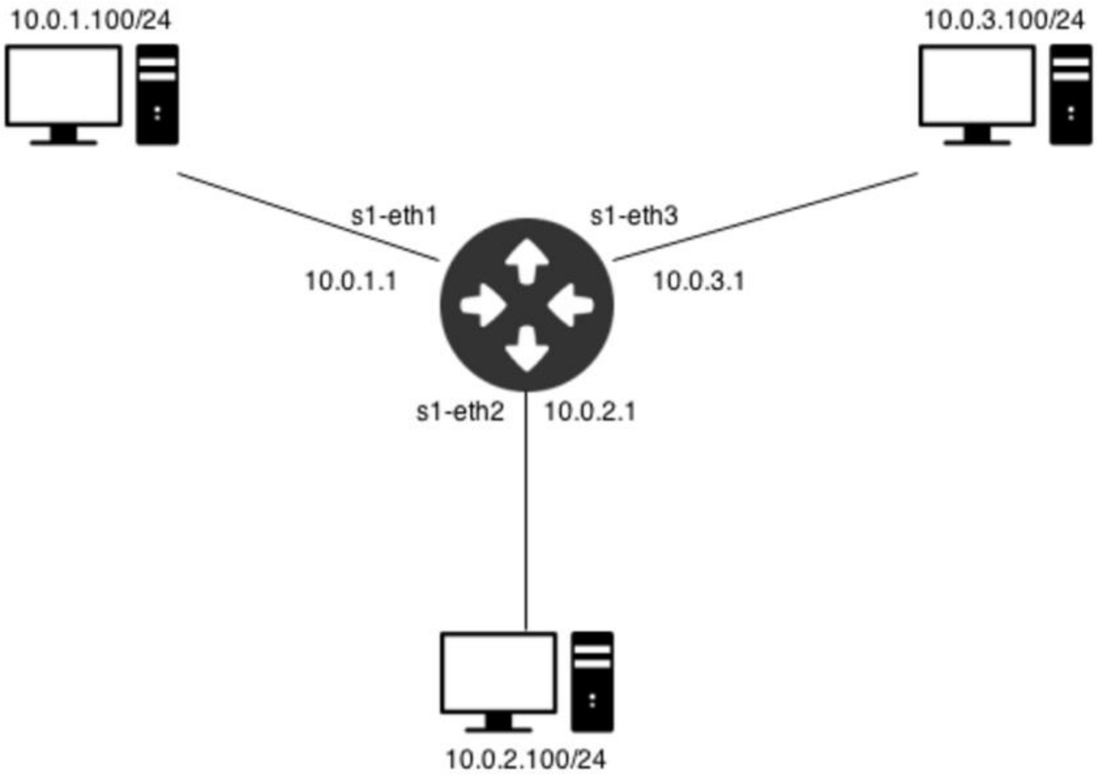
An IP router can be distinguished from other sorts of packet switching devices in that a router examines the IP protocol header as part of the switching process. It generally removes the Link Layer header a message was received with, modifies the IP header, and replaces the Link Layer header for retransmission.

To simplify this exercise, the "router" is a completely static one. With no BGP or OSPF, we don’t have to send or receive route table updates.

Each network node will have a configured subnet. If a packet is destined for a host within that
subnet, the node acts as a switch and forwards the packet with no changes, to a known port or
broadcast, just like in the previous exercise. If a packet is destined for some IP address for which
the router knows the next hop; it should modify the layer-2 destination and forward the packet to
the correct port.

Since this is a Static Router the routing table is formulated as:
self.routingTable = [['10.0.1.100', '10.0.1.100', 's1-eth1', '10.0.1.1', 1],['10.0.2.100', '10.0.2.100', 's1-eth2', '10.0.2.1', 2],['10.0.3.100', '10.0.3.100', 's1-eth3', '10.0.3.1', 3]]

The configuration is setup and “mytopo” file was modified as per the below image. ‘addHost’
and ‘addSwitch’ were used and the IP addresses and their respective network id’s were set.



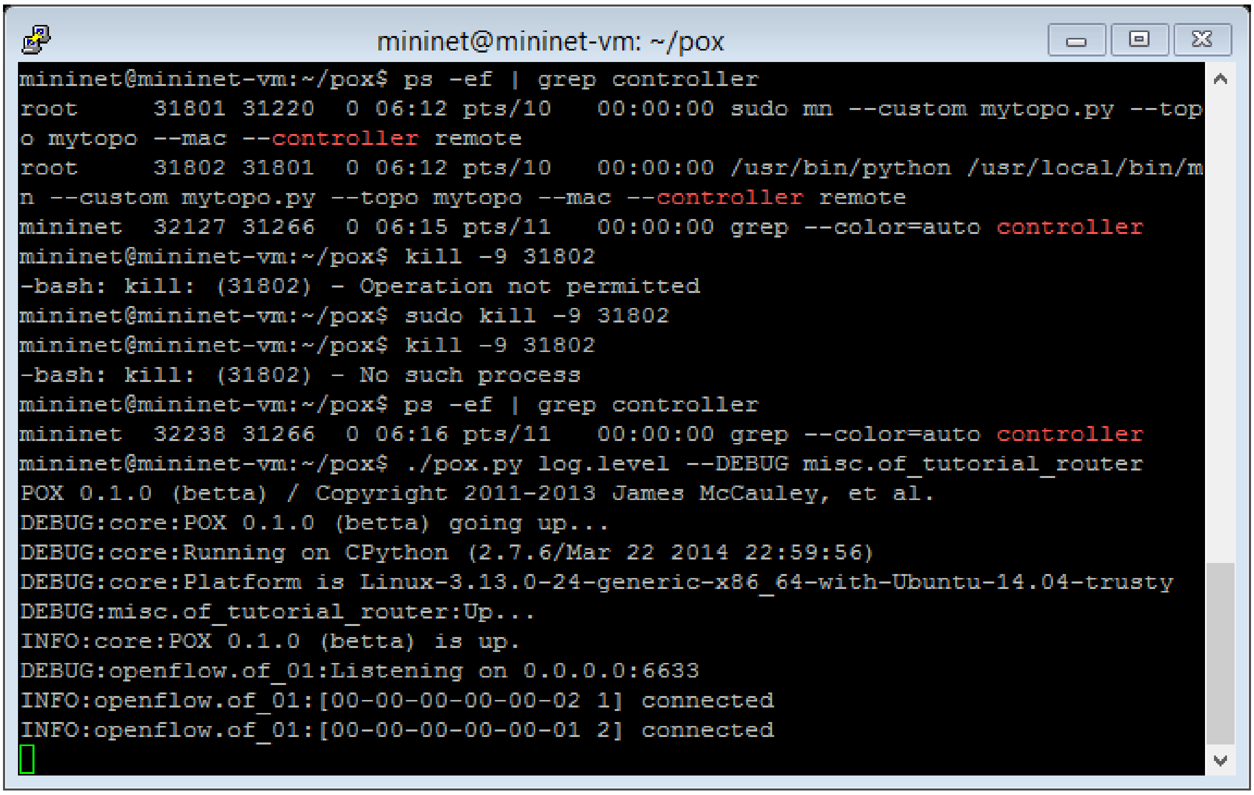
This code snippet explains how to add links and hosts:

```
# Add hosts and switches
        leftHost = self.addHost( 'h1',ip="10.0.1.100",defaultRoute="via 10.0.1.1" )
        rightHost = self.addHost( 'h2',ip="10.0.2.100",defaultRoute="via 10.0.2.1" )
        bottomHost = self.addHost( 'h3',ip="10.0.3.100",defaultRoute="via 10.0.3.1" )
        switch = self.addSwitch( 's1' )

# Add links
        self.addLink( leftHost, switch )
        self.addLink( rightHost, switch )
        self.addLink( bottomHost, switch )
```

sudo mn --custom mytopo1.py --topo mytopo --mac --switch ovsk --controller remote

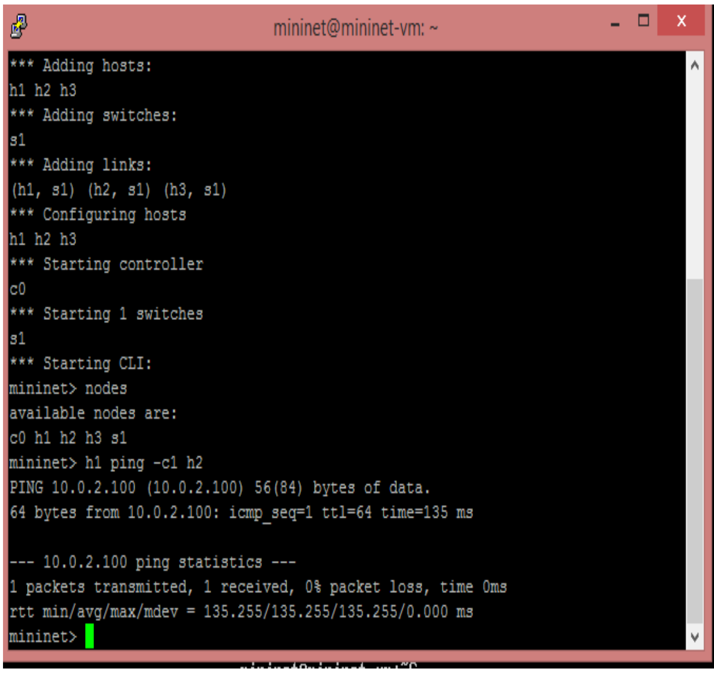
Startting controller:

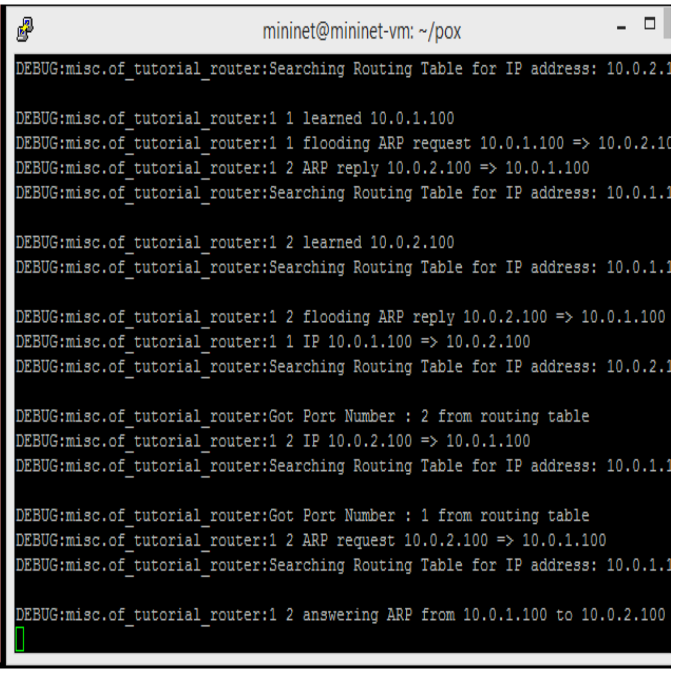


This topology is placed in home/mininet and the excecutable is run in the misc folder by calling:

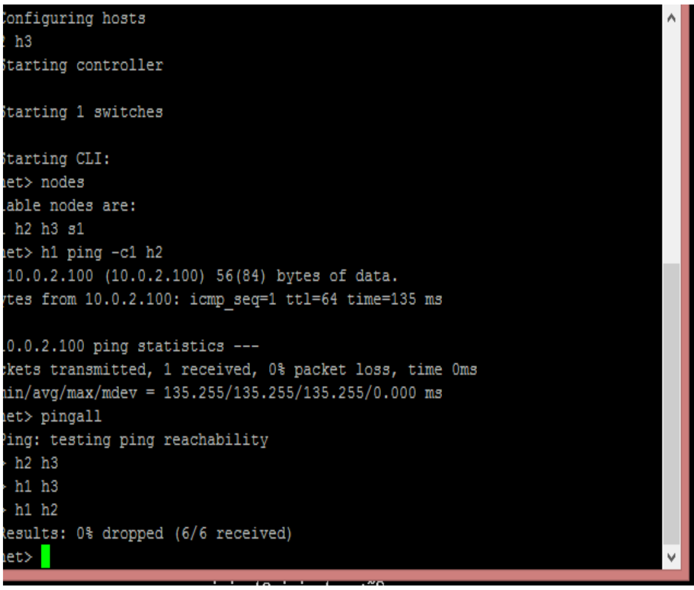
$ ./pox.py log.level --DEBUG misc.of_router_a misc.full_payload

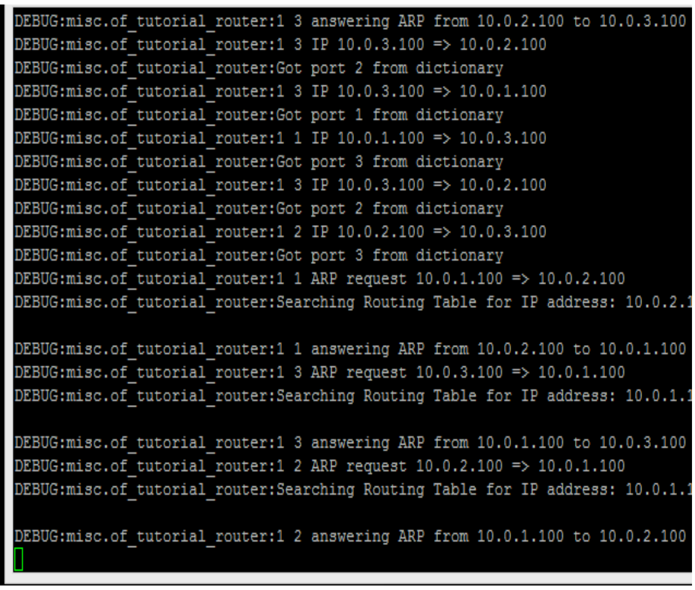
Ping h2 from h1 :



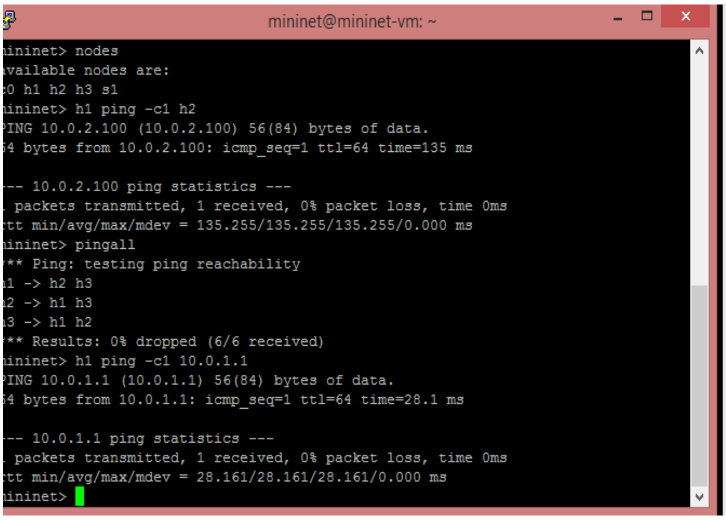


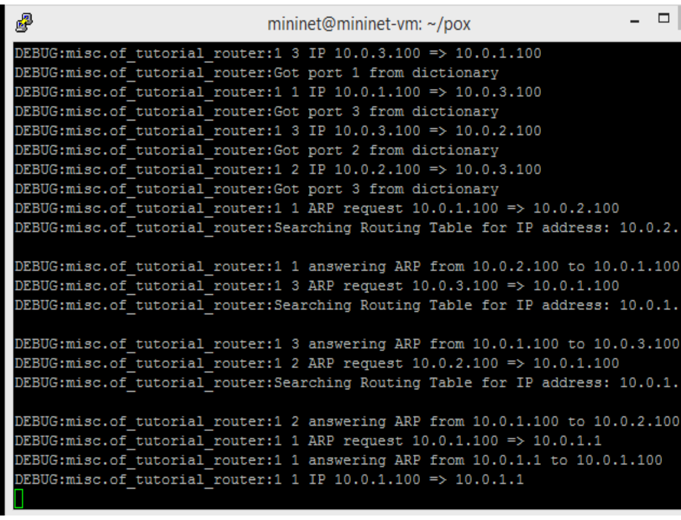
Pingall :





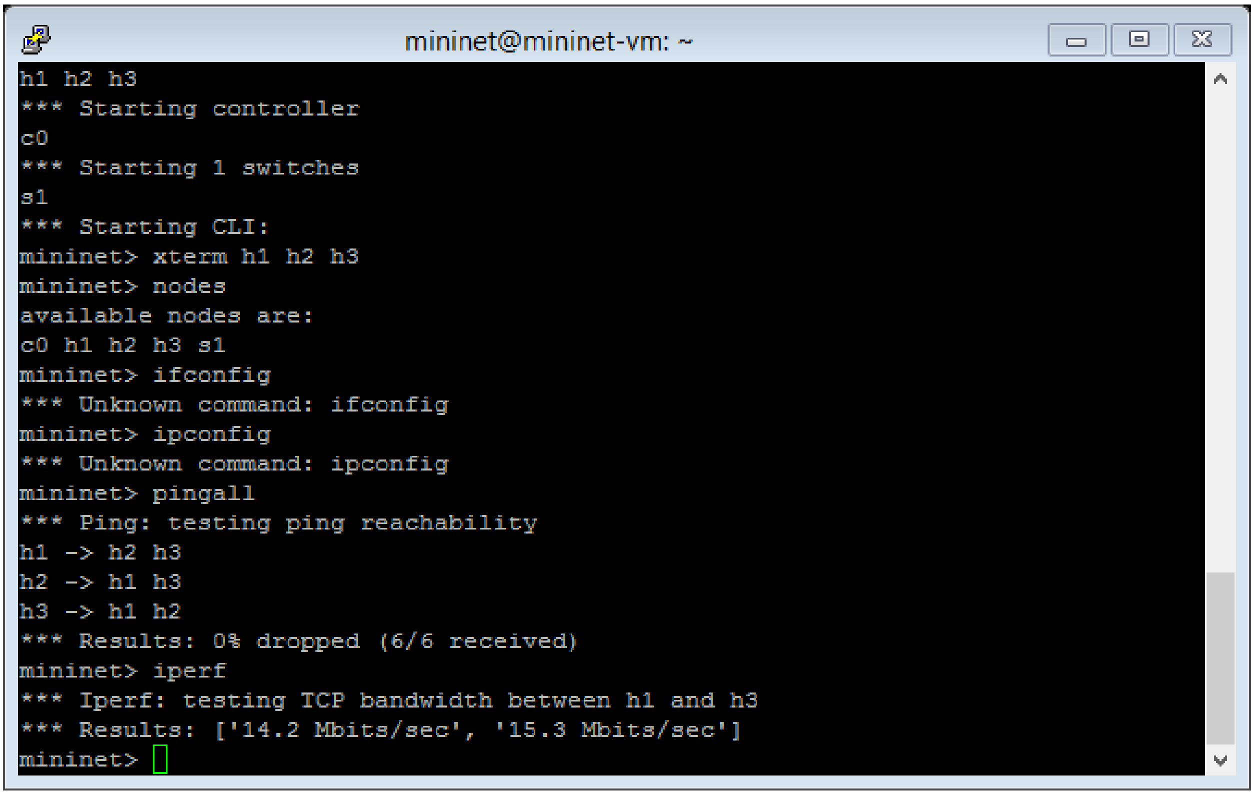
Reachable host:



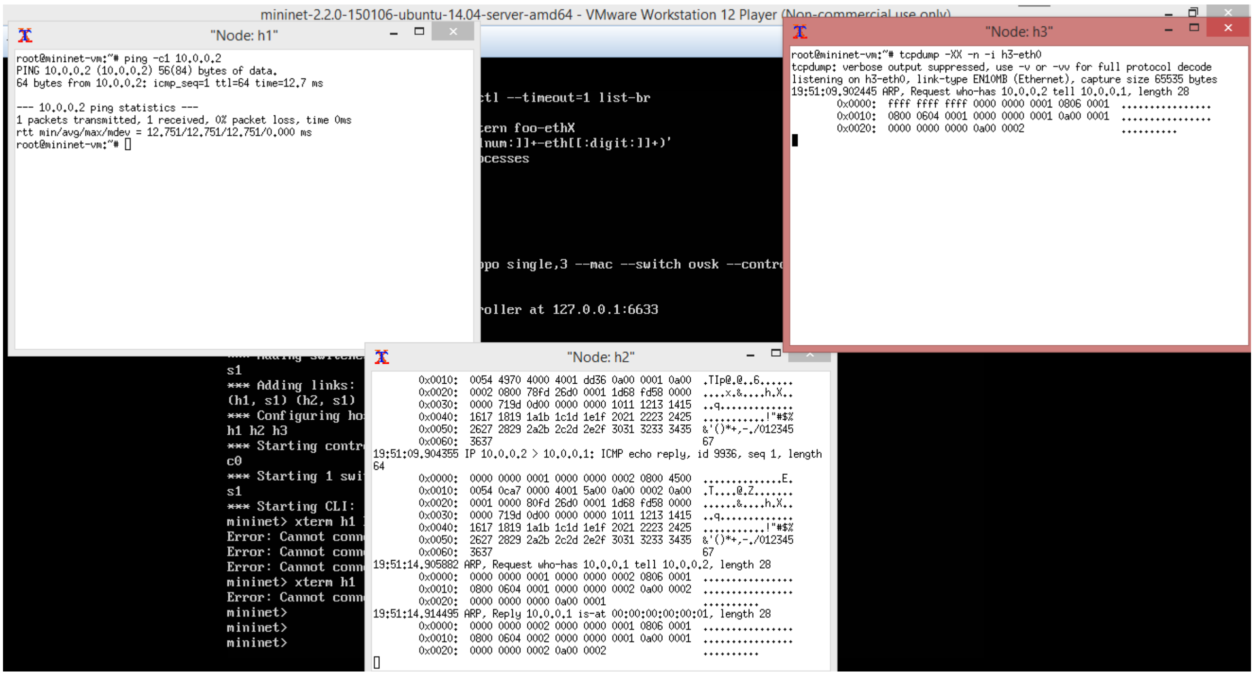


Whenever a packet arrives, the packet is parsed to see if it is either of ARP or IP type.
It processes the packet(see the attached code) to categorize it and send the required reply back.

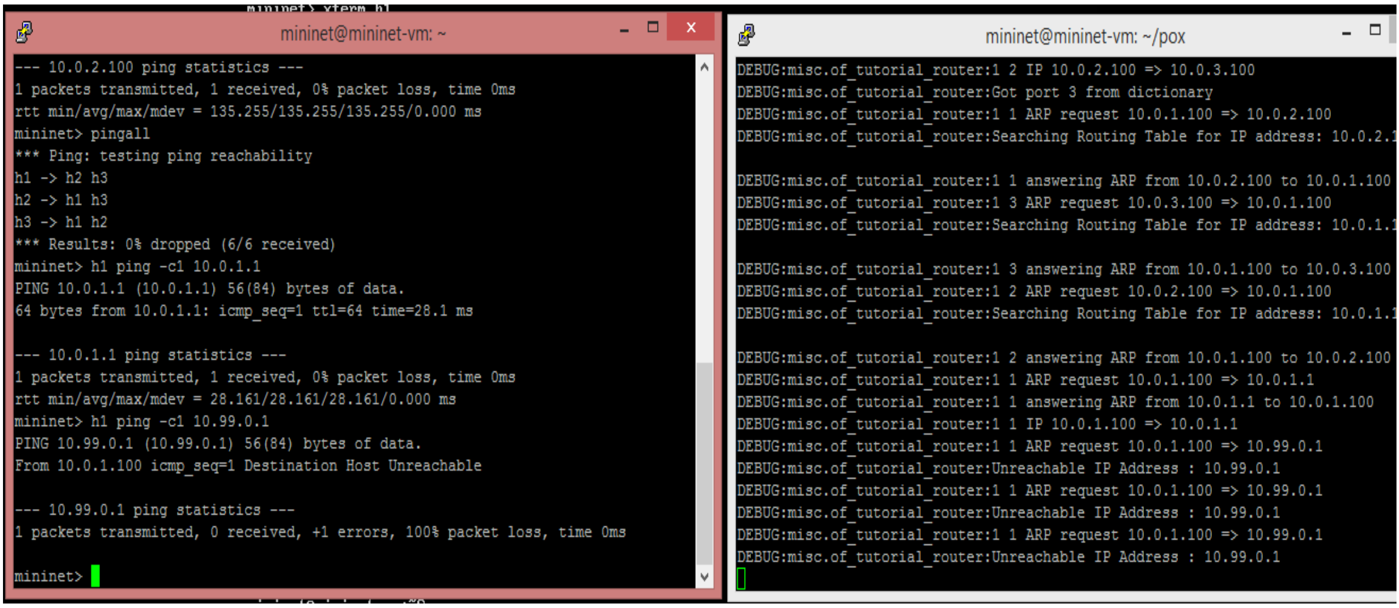
A pingall and iperf test was performed to see the working of this router:

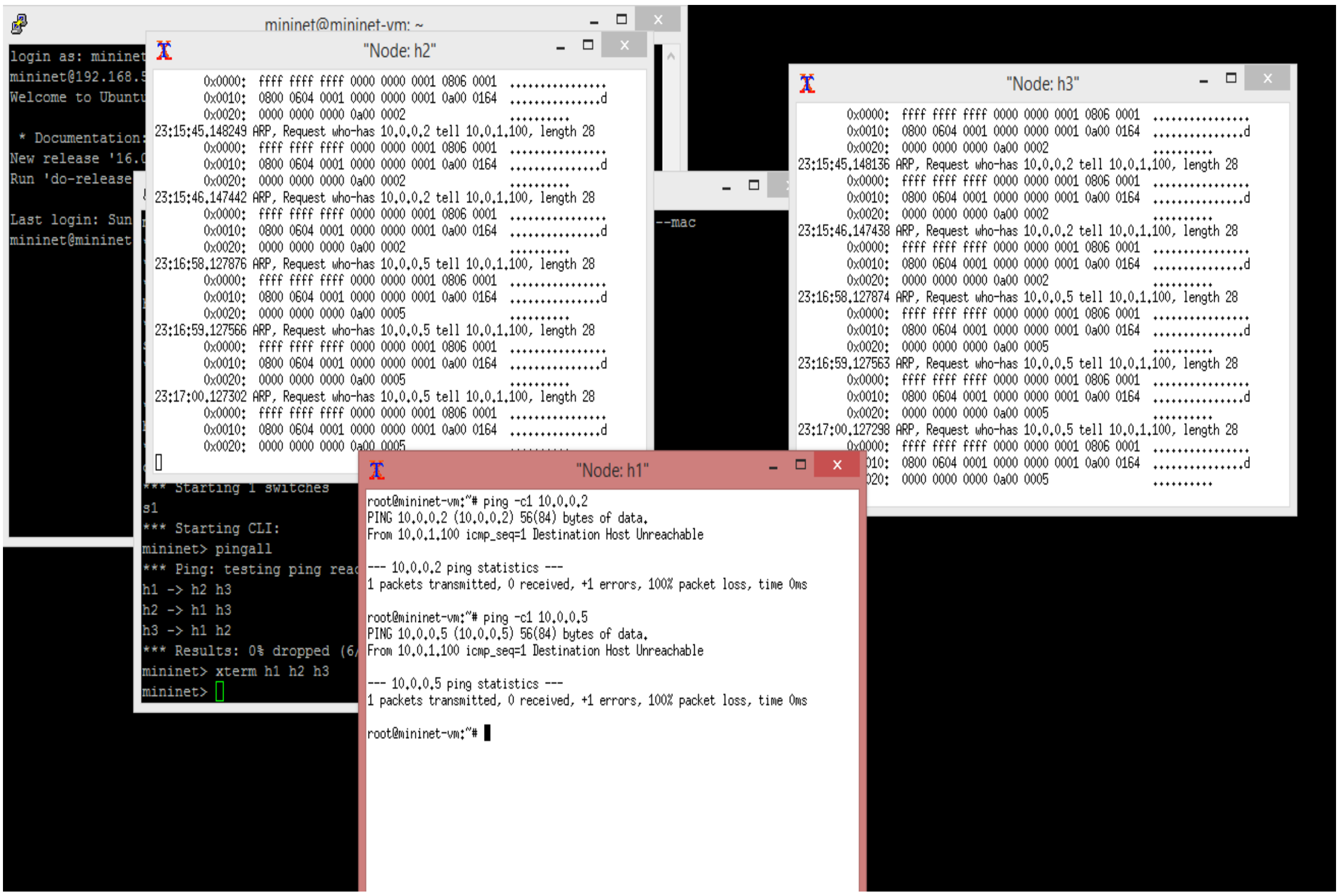


A pingall and iperf test was performed to see the working of this router:



Unreachable host:



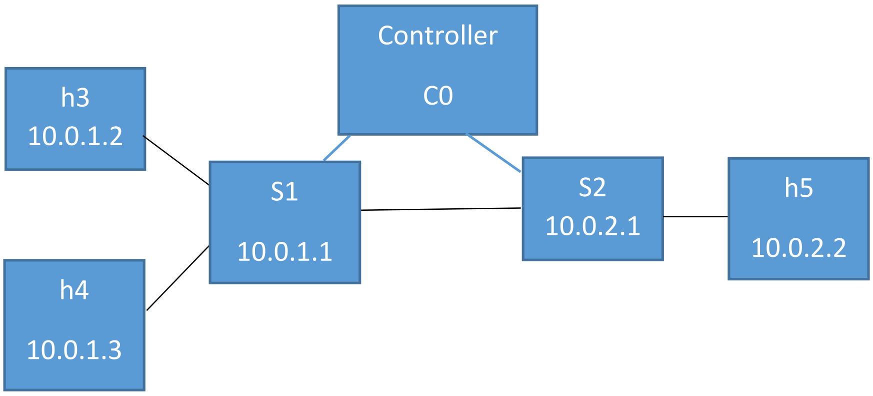


PART 2 IMPLEMENTATION:

This part requires us to make the topology give to us and perform the actions.

The topology given has :

- 3 hosts
- 2 switches

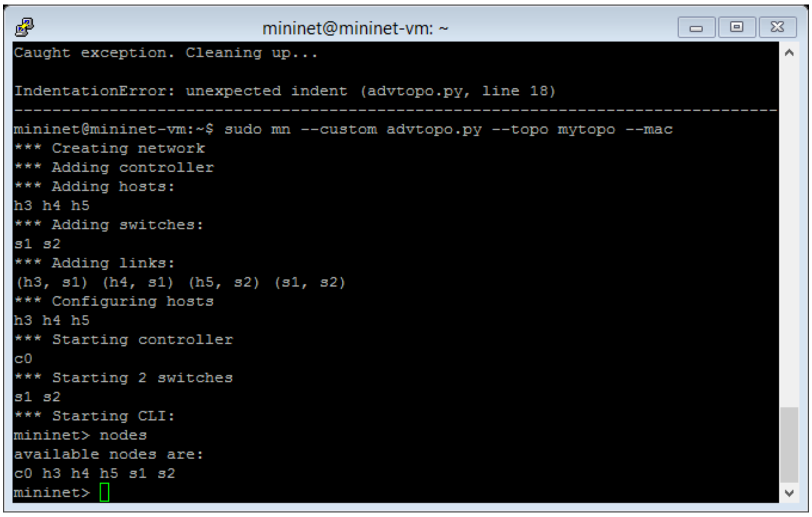


Similar to part 1 implementation, same steps are followed except for a few changes are made in the topology and code. First we need to create the above mentioned topology by editing the topology file by adding the needed components and assigning IP addresses and establishing links.

The controller performs the similar actions in this stage too with ARP and ICMP. Now here each network node will have a configured subnet. If a packet is destined for a host within that subnet, the node acts as a switch and forwards the packet with no changes, to a known port or broadcast, just like in the previous part. If a packet is destined for some IP address for which the router knows the next hop, it should modify the layer-2 destination and forward the packet to the correct port.

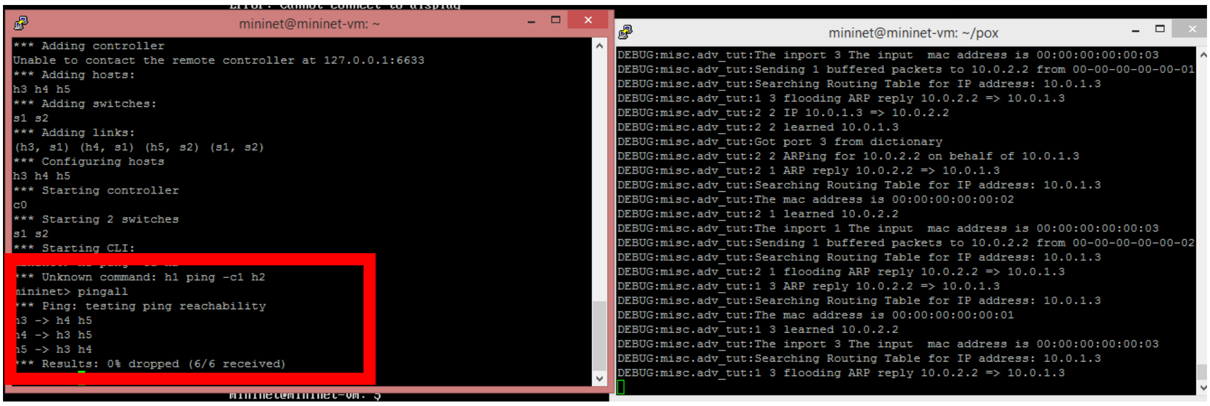
Creating the topology :

When we create the topology, we see this on the console and finally get the “ mininet > “



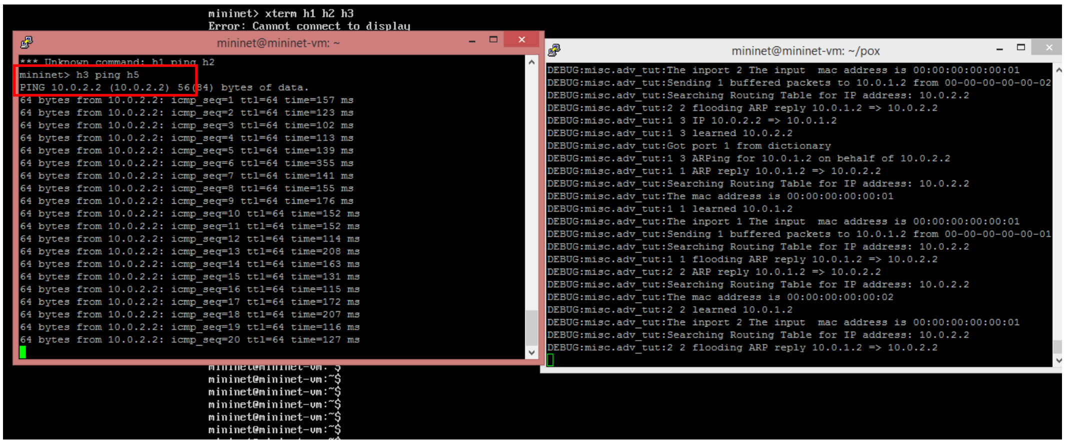
We need to modify the controller and add the necessary code and run the controller. We can see the controller and mininet screens side by side in the screenhots attached for reference.

Now the **ping all** test yield viz.



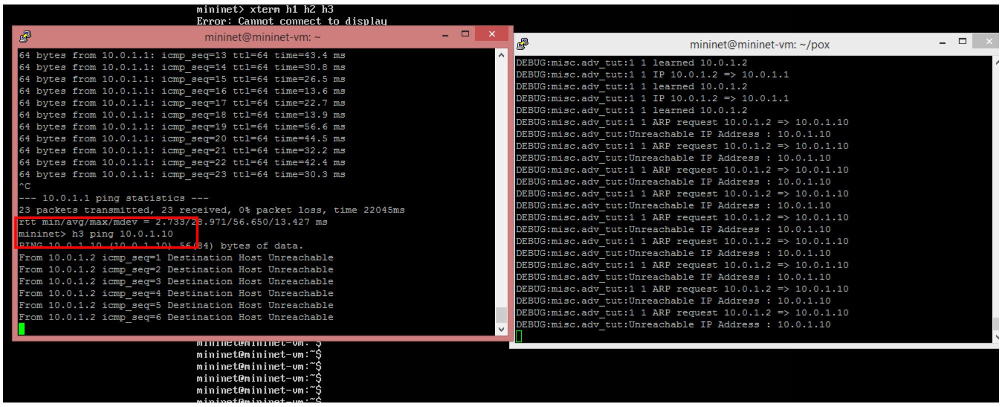
Pinging a known host:

When the controller receives ICMP echo (ping) requests for the known, respond with the following messages:

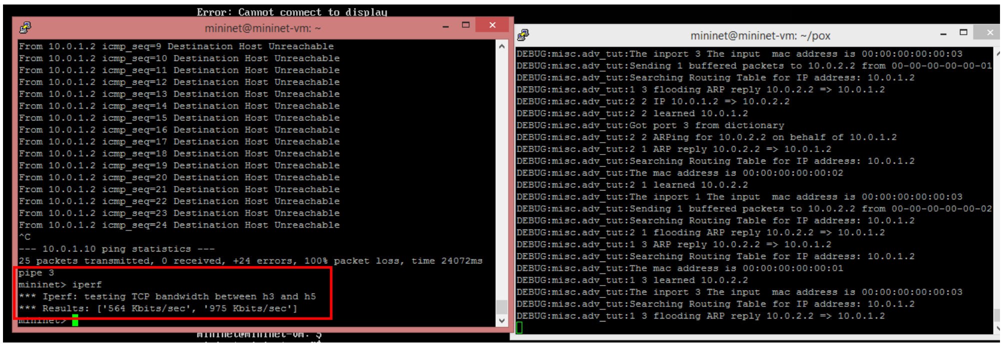


Pinging an unknown host :

For packets for unreachable subnets (IP;’s that don’t exits) it responds with ICMP network unreachable messages: (Eg: trying to ping 10.0.1.10- non-existent)

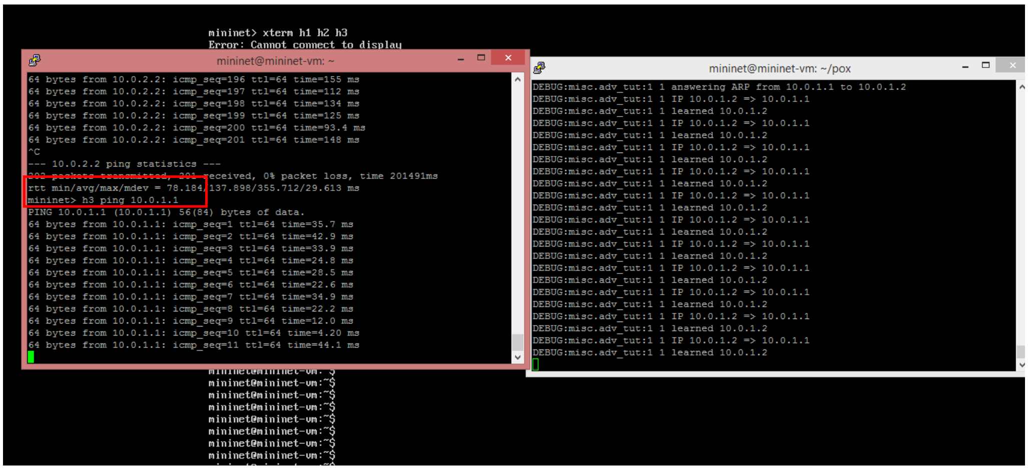


Iperf to check the bandwidth :



Pinging the switch from host :

The switch is should pingable, and generates an ICMP echo reply in response to an ICMP echo request.

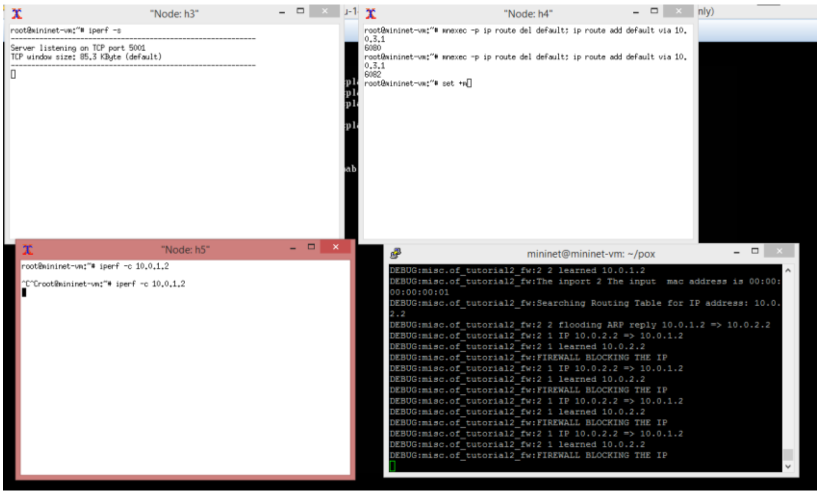


Firewall Part:

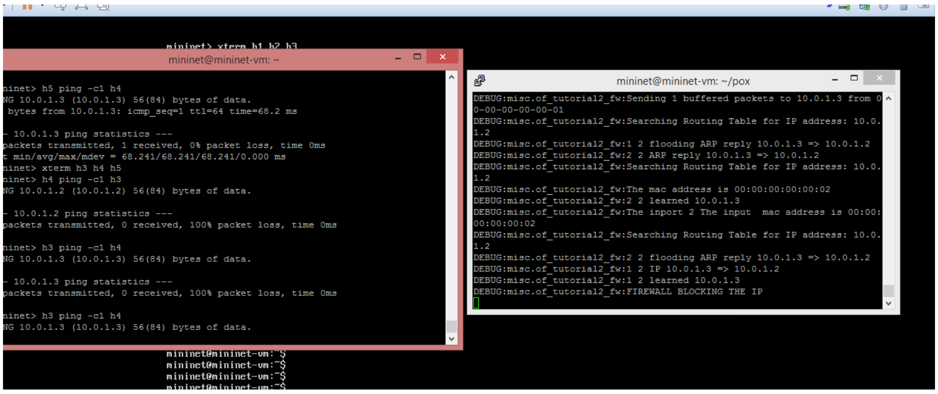
We modified the switch to reject connection attempts to specific ports, just like a firewall.

The modification helps show how OpenFlow can even do basic layer-4 tasks. The modification blocks all flow entries with this particular port - 10.0.1.2 (h3).

Firewall blocking the iperf request:



Ping gets blocked due to our firewall :



CONCLUSION:

OpenFlow and Mininet tools were used to successfully create switches as a router and hub and the respective configurations.
We learnt to create different topologies and try the commands that ensure all the hosts are connected. The messages for successful and unsuccessful flows were studied.
Implementation of firewall was done and tested and studied that show how OpenFlow can even do basic layer-4 tasks

REFERENCES:
https://openflow.stanford.edu/display/ONL/POX+Wiki
http://mininet.org/walkthrough/
https://github.com/mininet/openflow-tutorial/wiki