OpenFlow Protocol

EE555 Final Project

Jie Luo

Yang Meng

Rouqi Wang

Abstract

OpenFlow is an open interface for remotely controlling the forwarding tables in network switches, routers, and access points. Upon this low-level primitive, researchers can build networks with new high-level properties. For example, OpenFlow enables more secure default-off networks, wireless networks with smooth handoffs, scalable data center networks, host mobility, more energy-efficient networks and new wide-area networks – to name a few. This project is to gain hands-on experience with the platforms and debugging tools most useful for developing network control applications on OpenFlow.

The tutorial turns the provided hub controller into a controller-based learning switch, then a flow-accelerated learning switch, and extend this from a single-switch network to a multiple-switch multiple-host network. Along the way, we learn the full suite of OpenFlow debugging tools.

Our implementation and report includes the following partitions: part 1, create a Learning Switch and Router Exercise; part 2, Advanced Topology and create firewall; part 3, Create our own topology.

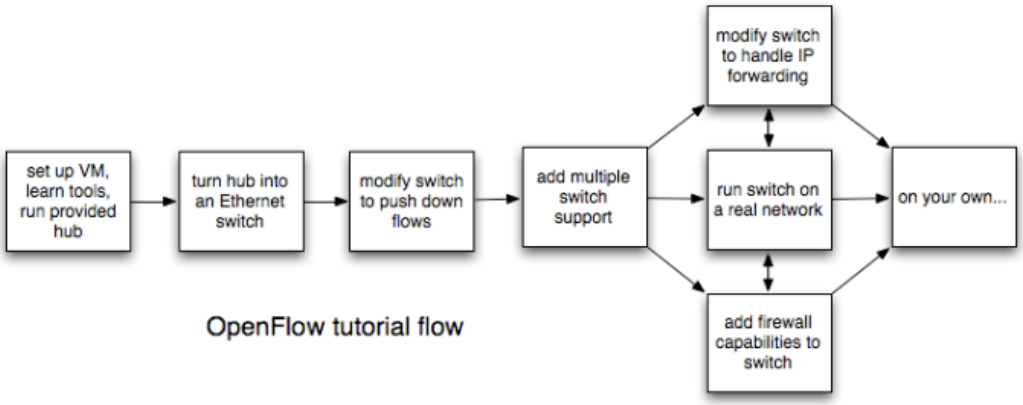

Fig.1 OpenFlow tutorial flow

I. Part 1

1.1 Create a Learning Switch

Our controller is POX and all the logic is written in Python. To run a testing topology, firstly kill all the running controller program from the other SSH window:

```
$ sudo killall controller
```

Run sudo mn -c and restart Mininet to make sure that everything is "clean" and using the faster kernel switch. From Mininet console:

```
mininet> exit

$ sudo mn -c

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

Now run a basic hub example:

```
$. /pox.py log. level --DEBUG misc. of_project
```

This tells POX to enable verbose logging and to start the of_project component which currently acts like a hub.

The switches may take a little bit of time to connect. When an OpenFlow switch loses its connection to a controller, it will generally increase the period between which it attempts to contact the controller, up to a maximum of 15 seconds. Since the OpenFlow switch has not connected yet, this delay may be anything between 0 and 15 seconds. If this is too long to wait, the switch can be configured to wait no more than N seconds using the --max-backoff parameter. Alternately, we exit Mininet to remove the switch(es), start the controller, and then start Mininet to immediately connect.

Wait until the application indicates that the OpenFlow switch has connected. When the switch connects, POX will print something like this:

```
INFO: openflow.of_01:[Con 1/1] Connected to 00-00-00-00-00-01

DEBUG: misc.of_project:Controlling [00-00-00-00-00-01 1]
```

The first line is from the portion of POX that handles OpenFlow connections. The second is from the project component itself.

1.1.1 Verify Hub Behavior with tcpdump

Now we verify that hosts can ping each other, and that all hosts see the exact same traffic - the behavior of a hub. To do this, we'll create xterms for each host and view the traffic in each. In the Mininet console, start up three xterms:

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

In the xterms for h2 and h3, run tcpdump, a utility to print the packets seen by a host:

```
# tcpdump -XX -n -i h2-eth0
```

and respectively:

```
# tcpdump -XX -n -i h3-eth0
```

In the xterm for h1, send a ping:

```
# ping -c1 10.0.0.2
```

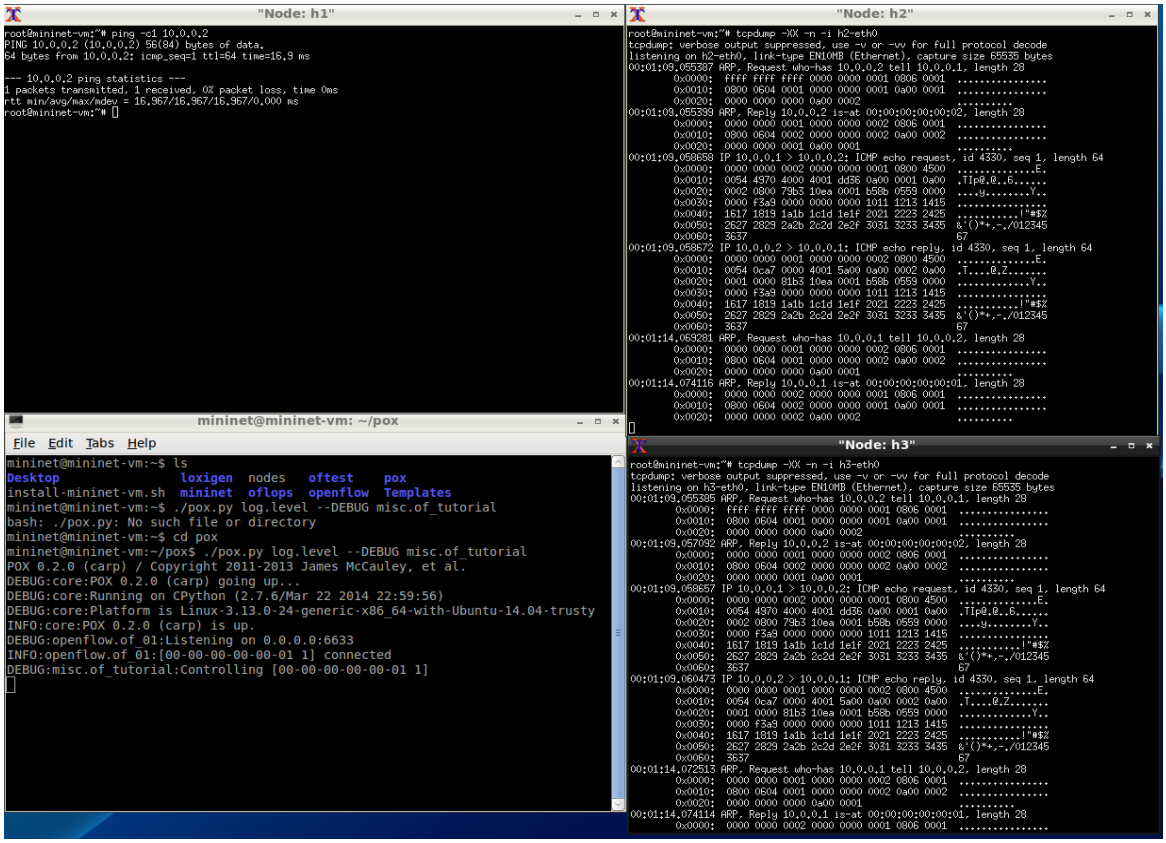

Fig.2 h1 pings a known IP

The ping packets are now going up to the controller, which then floods them out all interfaces except the sending one. We should see identical ARP and ICMP packets corresponding to the ping in both xterms running tcpdump. This is how a hub works: it sends all packets to every port on the network.

Now, see what happens when a non-existent host doesn't reply. From h1 xterm:

```
# ping -c1 10.0.0.5
```

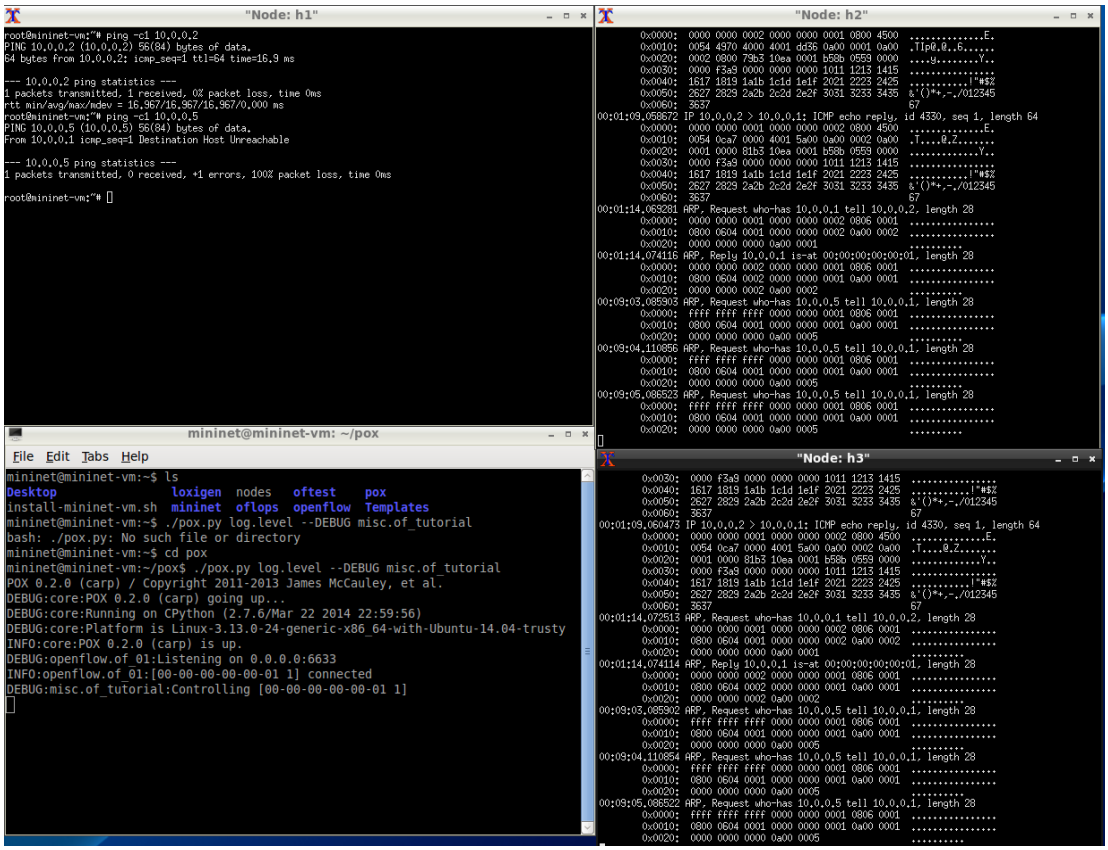

Fig.3 h1 pings an unknown IP

We see unanswered ARP requests in the tcpdump xterms.

1.1.2 Benchmark Hub Controller with iperf

Now we benchmark the provided of_tutorial hub. First, verify reachability. In the Mininet console, run:

```
mininet> pingall
```

This is just a sanity check for connectivity. Now, in the Mininet console, run:

```
mininet> iperf
```

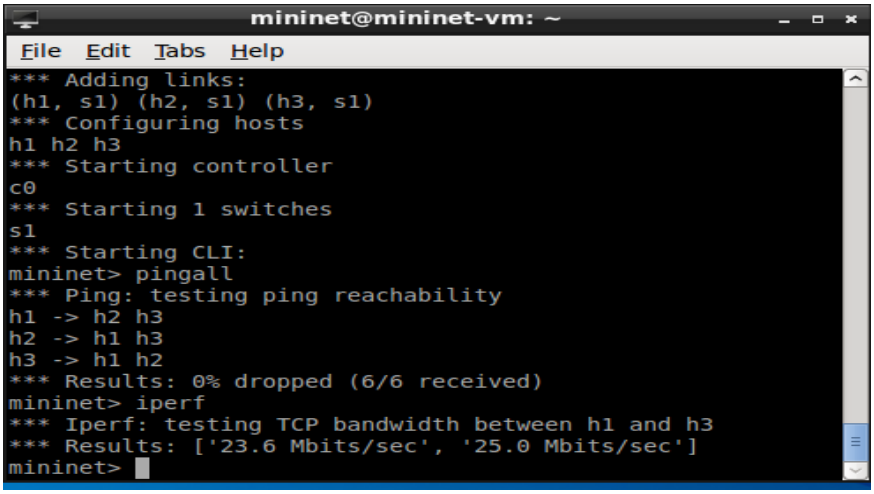

Fig.4 Hub controller

We also benchmark the reference controller. To start the reference controller, running:

```
$ controller ptcp:
```

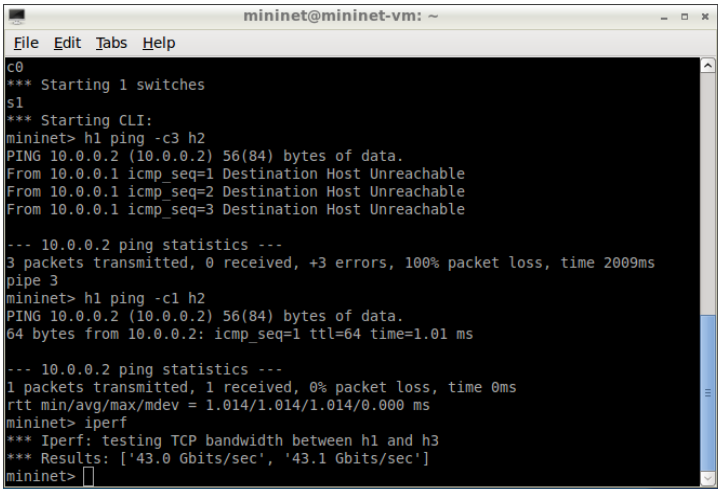

Fig.5 Reference controller

1.1.3 Implementation

The implementation of our switch is as following. In the initializer (the _init_ method), the class's instance variables are set accordingly and an empty dictionary (on that will later consist of MAC Address keys to port number values) is created.

```
self.mac_to_port = {}
```

In the method, act_like_hub, controller send received packets to all ports, using the resend_packet, which is the hub behavior.

Our act_like_switch method allows for learning switch behavior. The first thing to do is learn the port for the source MAC, which essentially means to populate the dictionary with the input packet's source MAC Address and input port.

```
self.mac_to_port[packet.src] = packet_in.in_port
```

We do not really need to check if the key/value pair is already in there, because after adding the flow mold, the controller will not see these packets. If the packet's destination port is not null, send the packet out to that port using resend_packet, and make a flow mod.

Our ofp_match object is as following:

```
msg = of.ofp_flow_mod()

msg.match.dl_dst = packet.dst

msg.idle_timeout = 1000

out_action = of.ofp_action_output(port = self.mac_to_port[packet.dst])

msg.actions.append(out_action)

self.connection.send(msg)
```

Otherwise, send the packet to all ports (“Flood” behavior) by using the port “of.OFPP_ALL”.

1.1.4 Testing Controller

To test controller-based Ethernet switch, we verify this behavior by running iperf in Mininet. Compared with Fig.4 and 5, the Reported iperf bandwidth is much higher as well, and match the number we got when using the reference learning switch controller earlier.

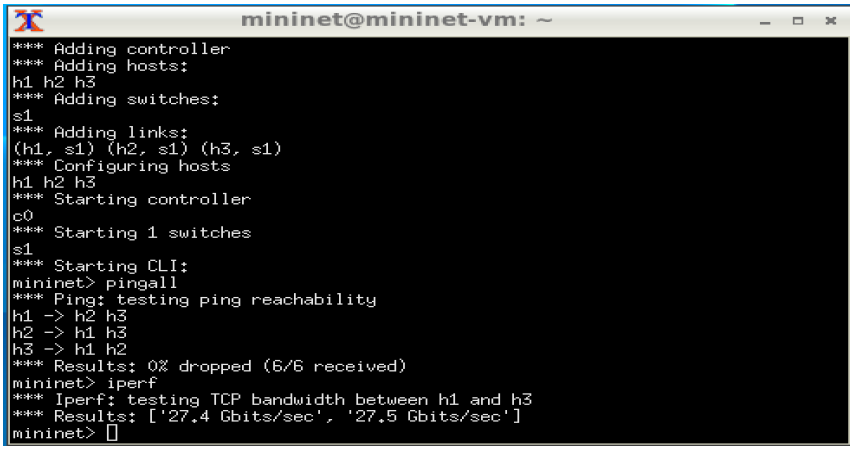

Fig.6 Our learning controller

1.1.5 Support Multiple Switches

When using POX, code is already able to pass the following tests because it creates one instance per switch and each instance maintains its own MAC table.

Start mininet with a different topology. In the Mininet console:

```
mininet> exit $ sudo mn --topo linear --switch ovsk --controller remote
```

This will create a 2-switch topology where each switch has a single connected host and the created topology looks like this:

Note: for Mininet 2.0, the hosts are h1/10.1 and h2/10.2

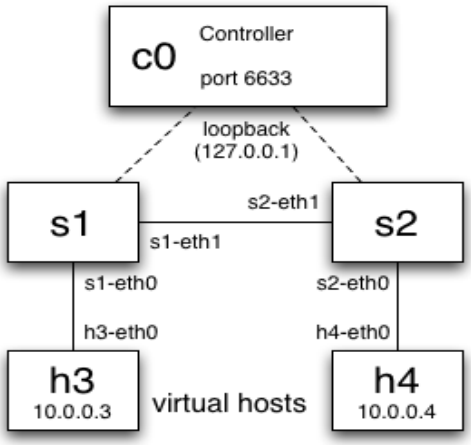

Fig.7 2-switch topology

We modify our switch so that it stores a MAC-to-port table for each DPID. This strategy only works on spanning tree networks, and the delay for setting up new paths between far-away hosts is proportional to the number of switches between them.

```
if dpid not in self.mac_to_port:
        log.debug("Add switch %s", str(dpid))
        self.mac_to_port[dpid] = {}
self.mac_to_port[dpid][packet.src] = packet_in.in_port
```

The result is as following:

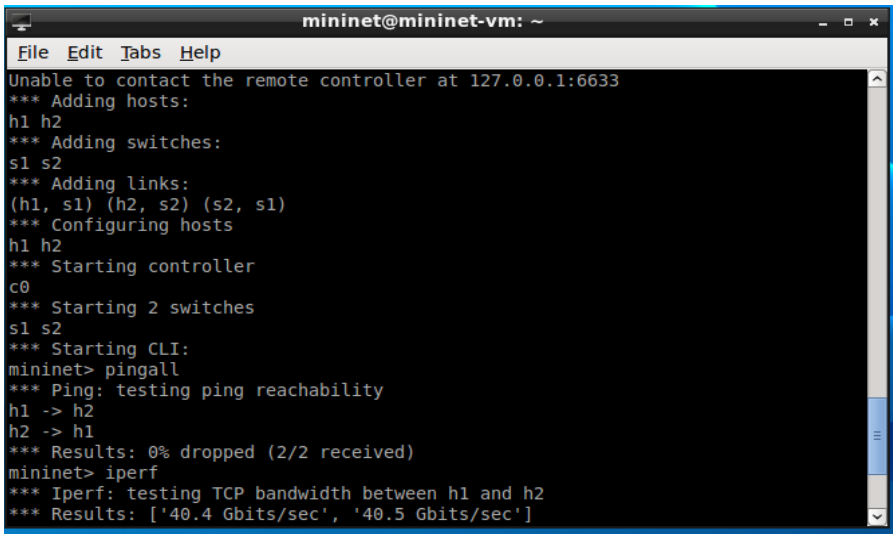

Fig.8 Learning controller for multi-switch topology

1.2 Router Exercise

In this section, we make a static layer-3 forwarder/switch. It's not exactly a router, because it won't decrement the IP TTL and recompute the checksum at each hop (so traceroute won't work). However, it will match on masked IP prefix ranges, just like a real router. Our router is completely static without BGP or OSPF, and there is no need 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.

1.2.1 Create Topology

We modify the topology file to match the picture. To run a custom topology, pass Mininet the custom file and pass in the custom topology:

```
$ sudo mn --custom mytopo.py --topo mytopo --mac
```

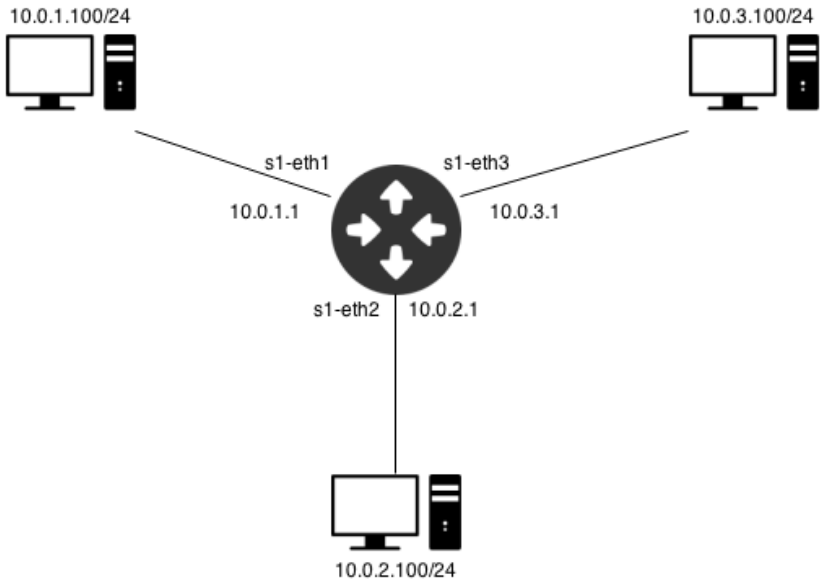

Fig.9 Topology for router exercise

1.2.2 Set up hosts

Set up IP configuration on each virtual host to force each one to send to the gateway for destination IPs that are outside of their configured subnet.

We need to configure each host with a subnet, IP, gateway, and netmask.

We do not attempt to assign IP addresses to the interfaces belonging to switches s1 and s2. If we need to handle traffic "to" or "from" a switch, we do so using OpenFlow.

We can do this directly from the custom topology that we created with mininet. Edit “mytopo.py" to include the information above. Helpful Code:

```
host1 = self.addHost( 'h1', ip="10.0.1.100/24", defaultRoute = "via 10.0.1.1" )
```

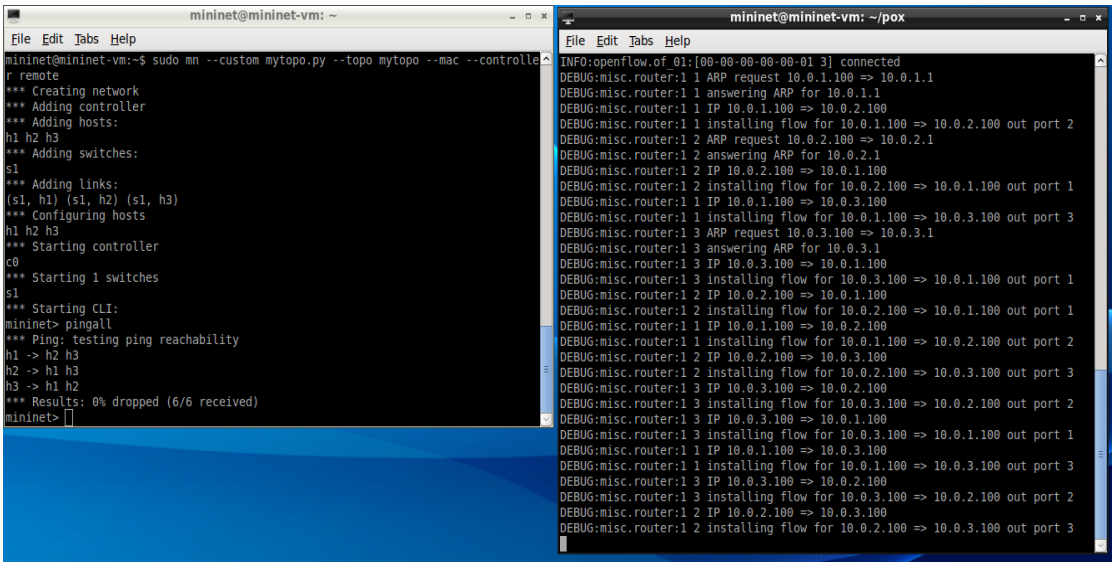

Fig.10 Verifying reachability

1.2.3 Implementation

A router generally has to respond to ARP requests. The ethernet broadcasts will be forwarded to the controller. Our controller constructs ARP replies and forward them out the appropriate ports.

Needed structures:

```
* arp cache
* routing table (create a structure with all of the information statically assigned)
* ip to port dictionary
* message queue (while the router waits for an ARP reply)
```

Our implementation is as following. When receiving an ARP request, the switch firstly see if it already has this ARP entry. If it does, then reply it; if not, it will flood an arp request for it and send this arp request into its arp cache with an expiration. After it has expired, this arp request will be abandoned and the switch will reply the host with an ICMP unreachable message. This will solve the unreachable ICMP message from hosts.

In case that the host will arp for the switch, the switch need a fake mac address. Also, the fake ip address of this switch need to be pre-configured. We use the switch's dpid to make up a fake mac address:

```
def dpid_to_mac (dpid):
    return EthAddr("%012x" % ((dpid*16*16*16 + dpid) & 0xffFFffFFffFF,))
```

And when the switch is firstly connected to the controller, this fake mac address and the fake ip address will be added into the arp table:

```
if dpid not in self.arpTable:
    log.debug("Add switch %s", str(dpid))
    self.arpTable[dpid] = {}
    for fake in self.fakeways[dpid-1]:
      self.arpTable[dpid][IPAddr(fake)] = Entry(of.OFPP_NONE,
       dpid_to_mac(dpid))
```

For static routing, we can add IP prefix as an arp table entry so that it will forward packet to the remote subnet.

The only change in the packet should be the source and destination MAC addresses. The mac addresses changing is added as an action of the ofp_flow_mod object:

```
actions.append(of.ofp_action_dl_addr.set_dst(mac))
```

Additionally, controller may receive ICMP echo (ping) requests for the router, which it should respond to. If the packet is ICMP type and the destination IP address is the switch itself, an ICMP reply message will be constructed as following:

```
packet.payload.payload.type = 0 # ICMP reachable

packet.payload.dstip, packet.payload.srcip = packet.payload.srcip, packet.payload.dstip

packet.src, packet.dst = packet.dst, packet.src
```

NOTE: the controller only accepts 128-byte packet_ins by default. If messages are getting truncated, this could be why. As a workaround, add another component at the end of call to the controller. it should look like this:

```
$ ./pox.py log.level --DEBUG ( controller) misc.full_payload
```

1.2.4 Testing router

If the router works properly:

A. Attempts to send from a host to an unknown address range like 10.99.0.1 should yield an ICMP destination unreachable message.
B. Packets sent to hosts on a known address range should have their MAC dst field changed to that of the next-hop router.
C. The router should be pingable, and should generate an ICMP echo reply in response to an ICMP echo request.

Now run iperf to test tcp and udp traffic. In mininet, open up xterm in host 1 and host 3 with:

```
mininet>xterm h1 h3
```

Host 3 will be the iperf server, so in its own terminal run the command

```
$ iperf -s
```

Host 1 will be the client, so in its own terminal run the command

```
$ iperf -c (ipaddress of iperf server)
```

We also test udp traffic by adding a -u option to the end of both commands. The results are shown in Fig. 11 and 12:

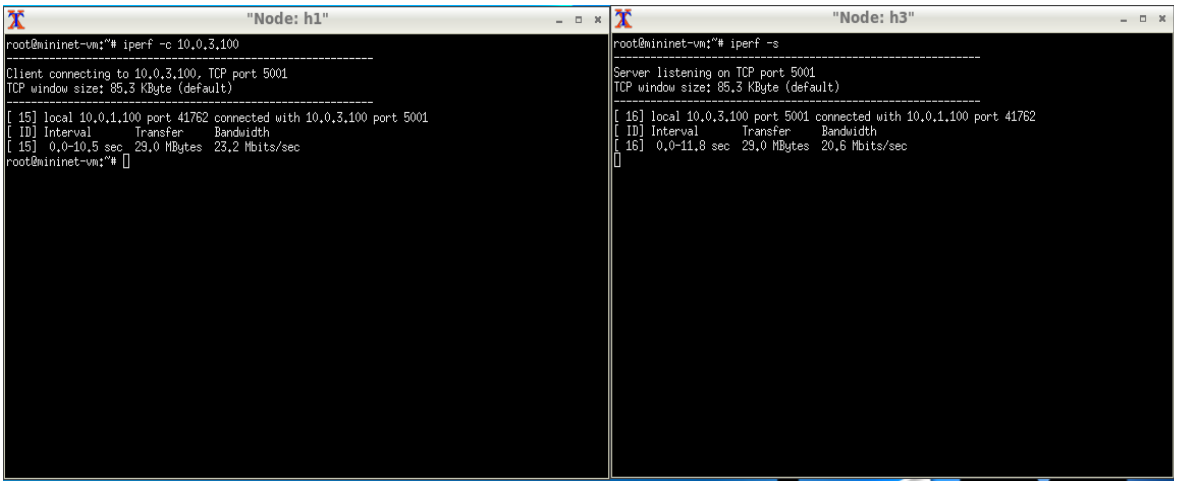

Fig.11 TCP traffic without flow mods

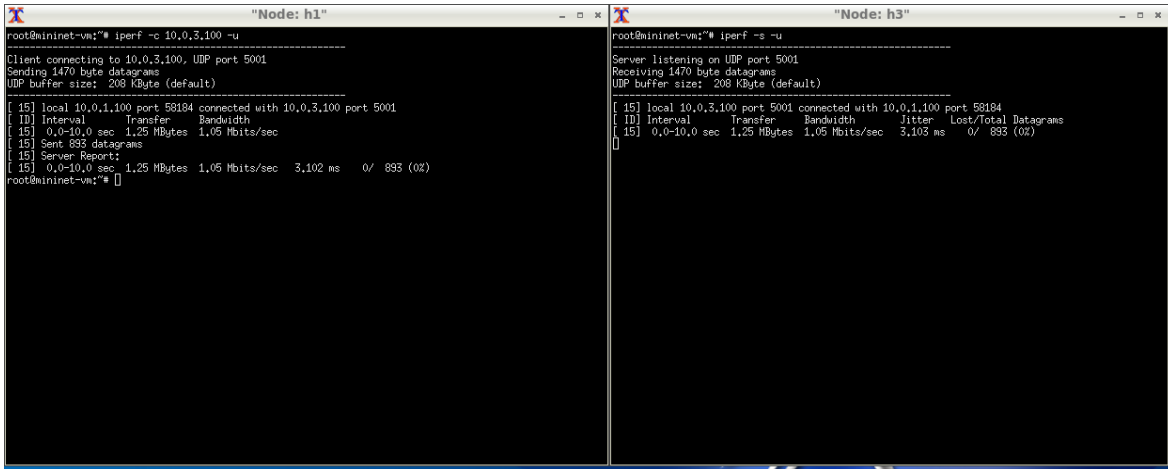

Fig.12 UDP traffic without flow mods

Note the results for this test. In code's current state, the router sends all of the traffic to the controller via packet_ins, which makes a routing decision and sends a packet_out to the router.

A. Attempts to send from a host to an unknown address range should yield an ICMP destination unreachable message.

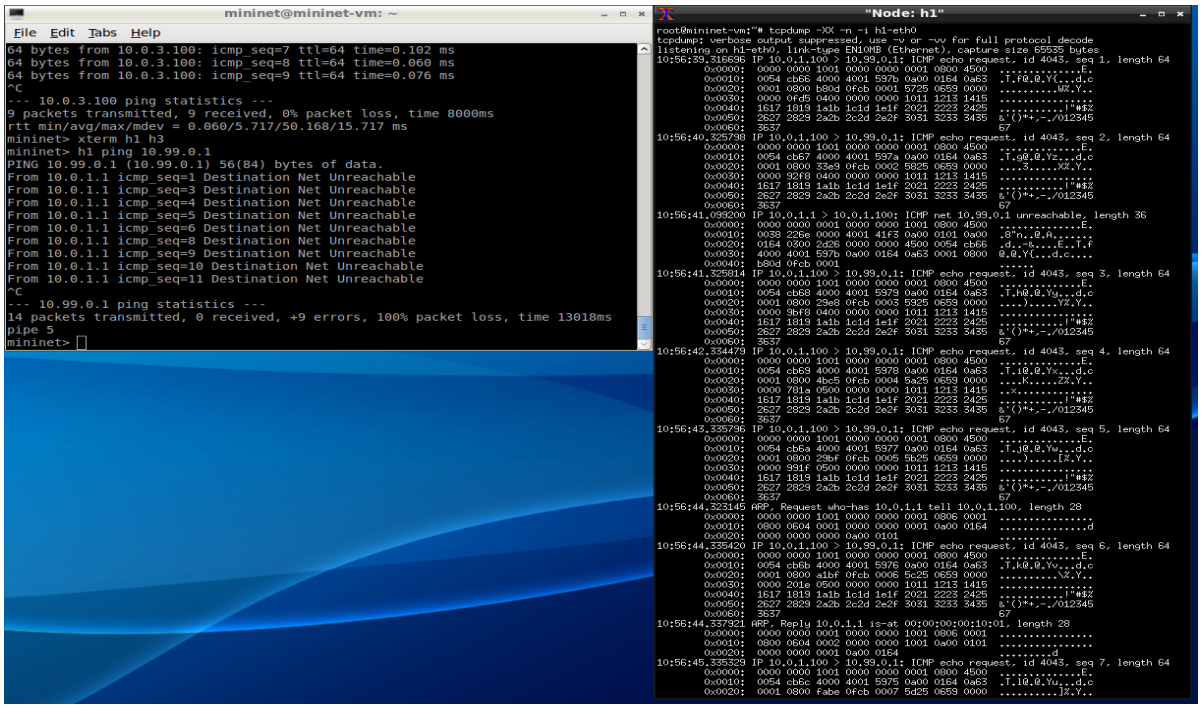

Fig.13 h1 pings an unknown IP

B. Packets sent to hosts on a known address range should have their MAC dst field changed to that of the next-hop router.

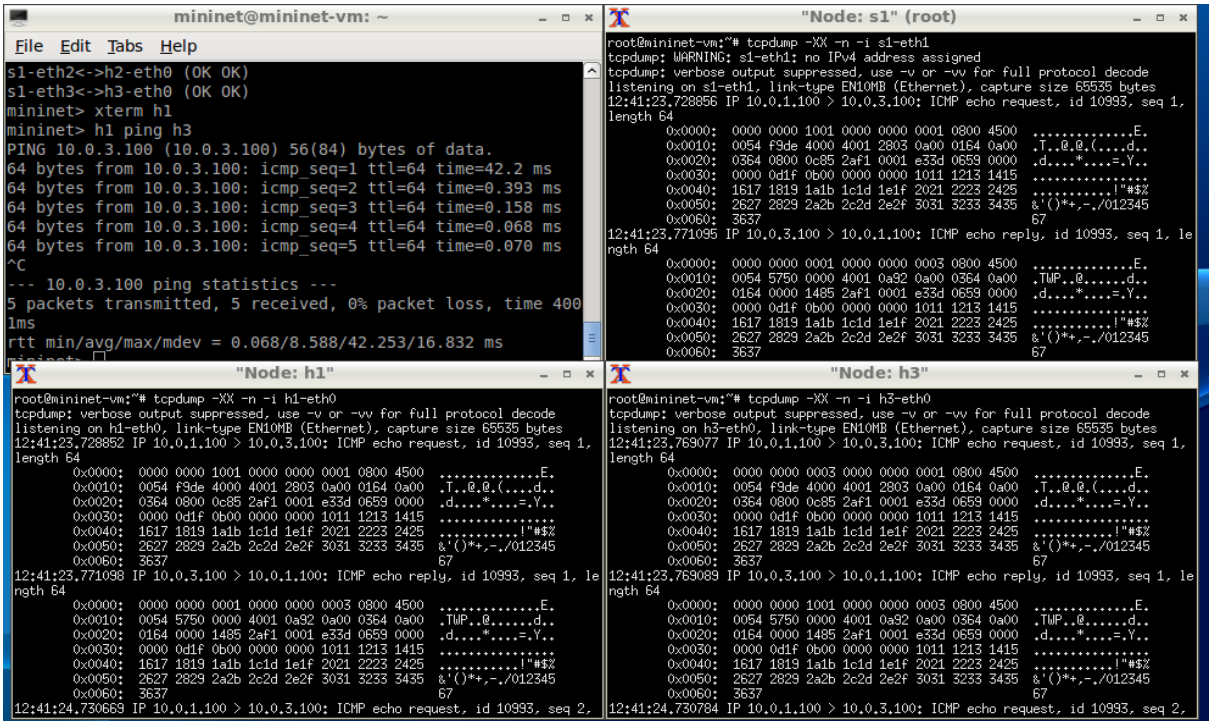

Fig.14 h1 pings h3

C. The router should be pingable, and should generate an ICMP echo reply in response to an ICMP echo request.

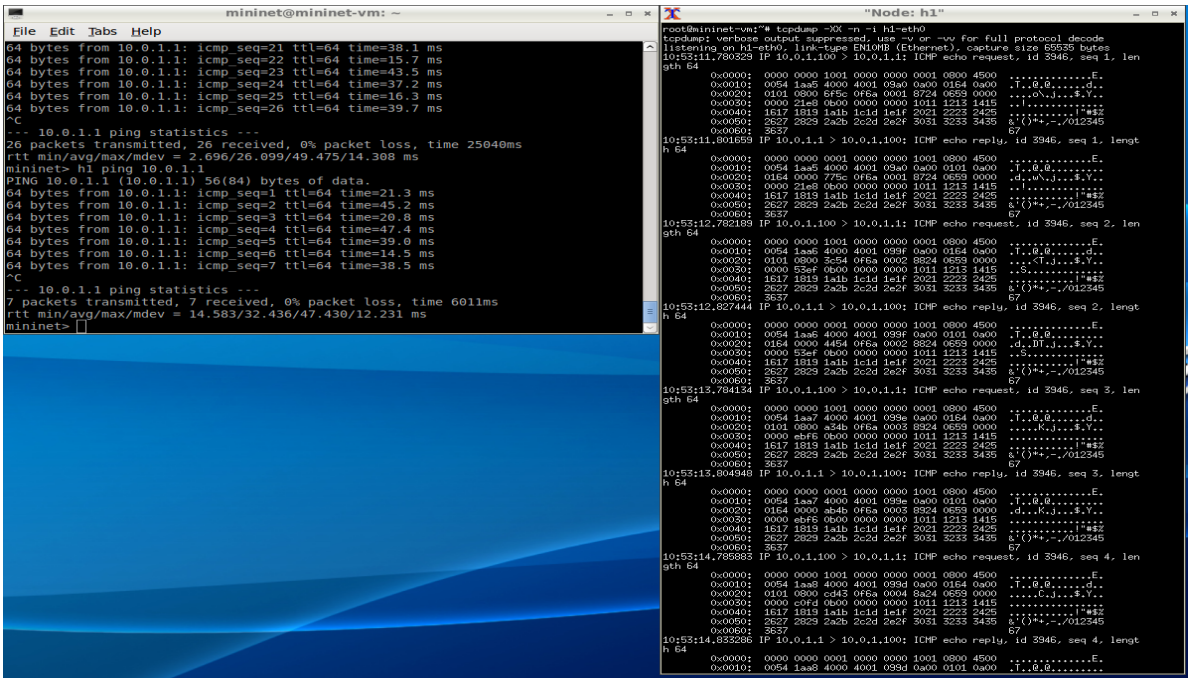

Fig.15 h1 pings the router

D. All hosts must be connected to each other. This can be verified using 'pingall'.

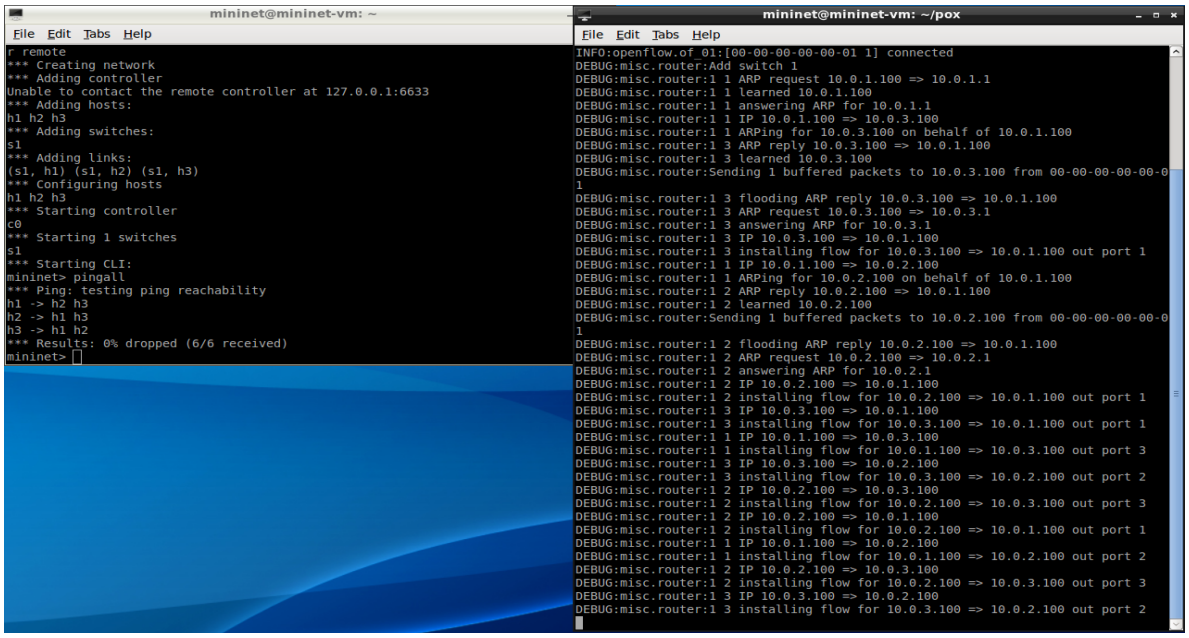

Fig.16 Verifying reachability of the testing router

Fig 17 and 18 show that the controller with flow mods yields better performance with iperf compared with Fig.11 and 12.

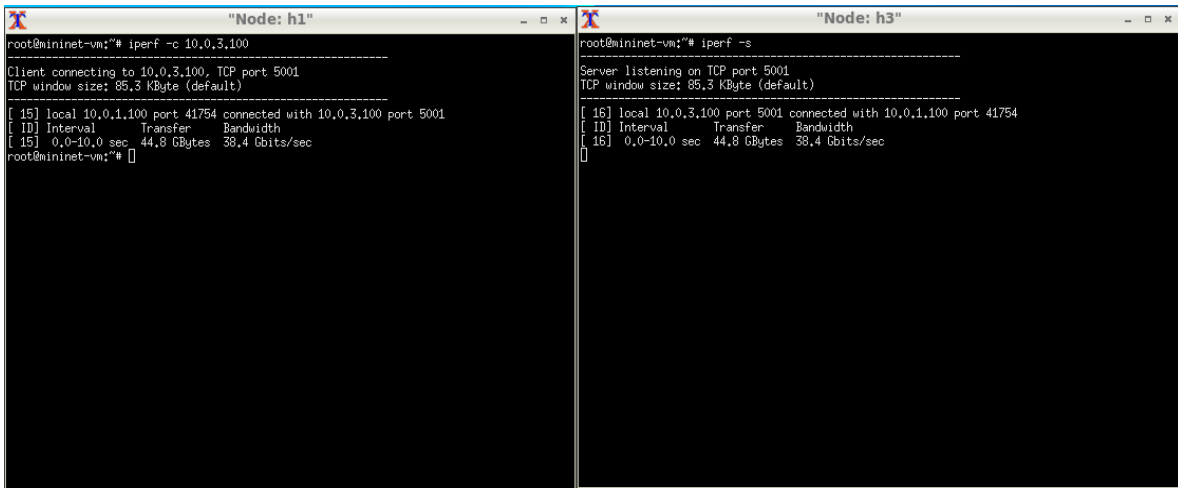

Fig.17 TCP traffic with flow mods

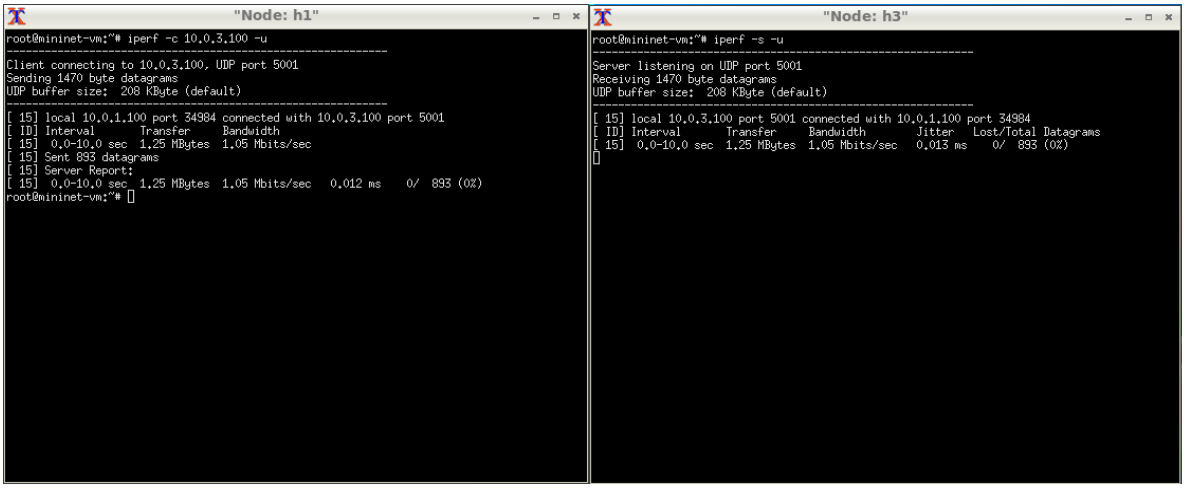

Fig.18 UDP traffic without flow mods

II. Part 2

2.1 Advanced Topology

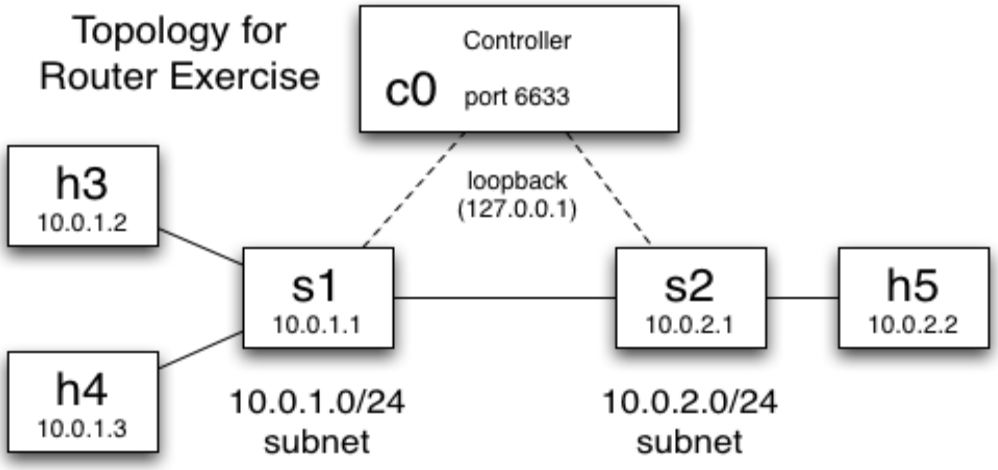

Fig.19 Advanced topology

The topology includes two routers controlling different subnets. The multi router case is much similar to the multi switches case. We use dpid to differentiate the arp table for each switch and the message sent from the controller:

```
core.openflow.sendToDPID(dpid, msg)
```

A. Attempts to send from a host to an unknown address range should yield an ICMP destination unreachable message.

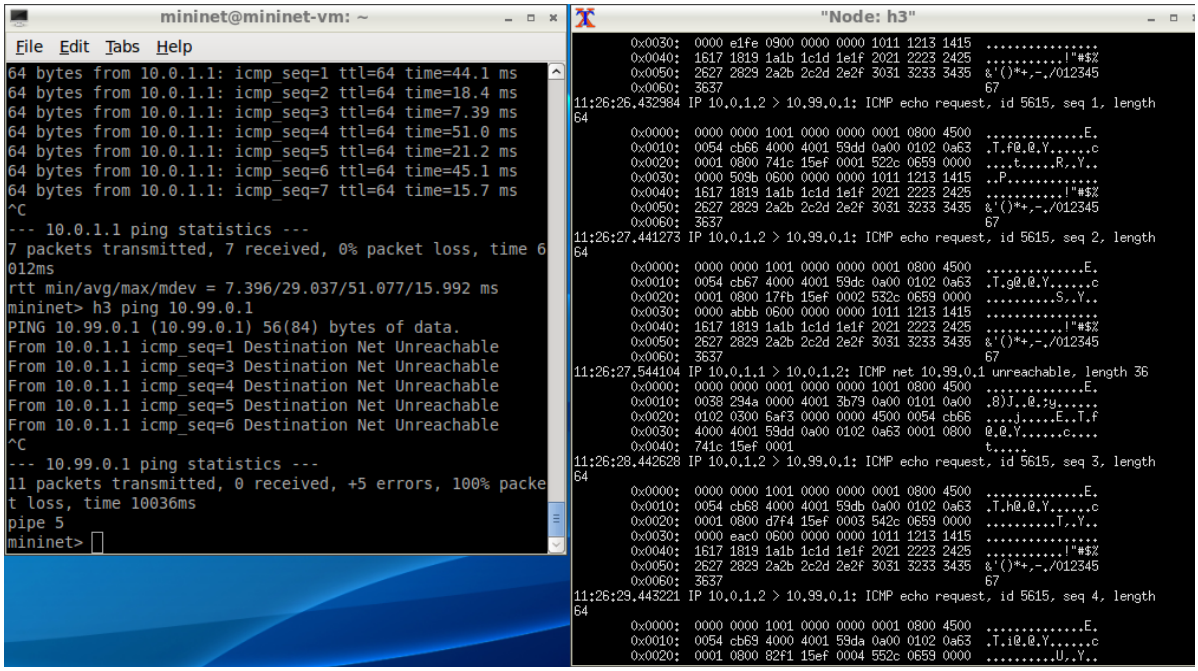

Fig.20 h3 pings an unknown IP

B. Packets sent to hosts on a known address range should have their MAC dst field changed to that of the next-hop router.

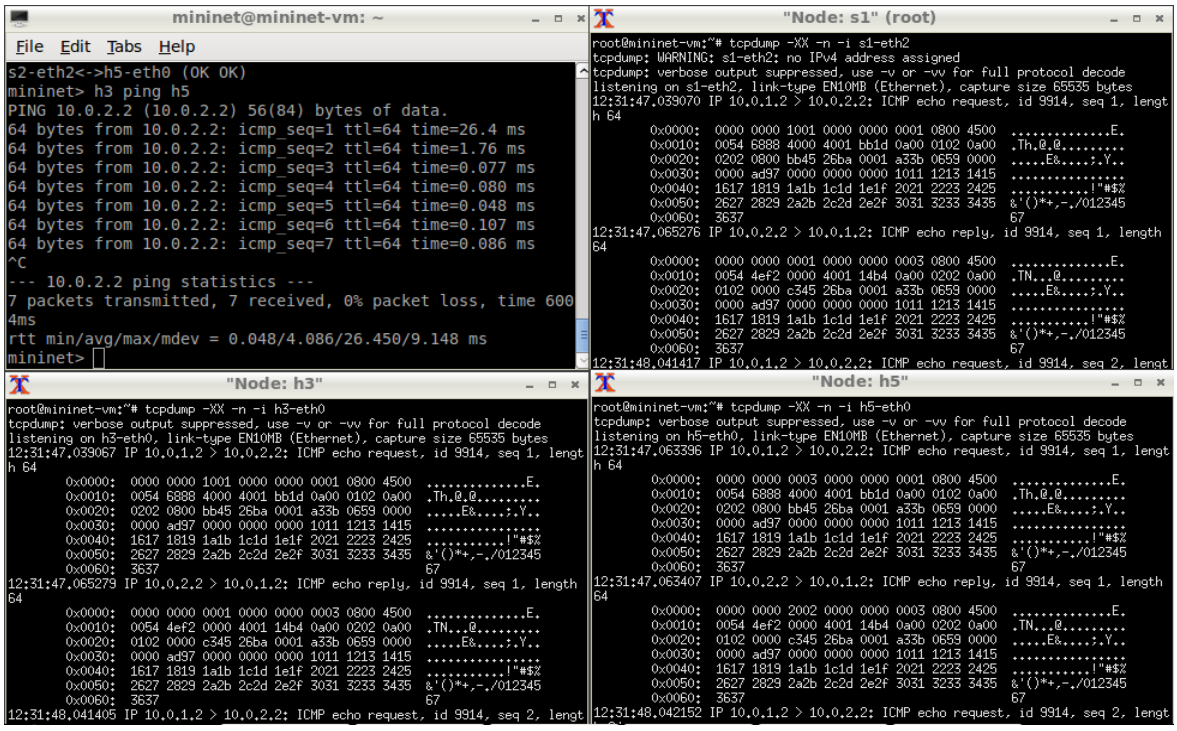

Fig.21 h3 pings h5

As can be seen from Fig.21, s1 has changed the MAC dst field from 00:00:00:00:00:01 to 00:00:00:00:00:03, which is the MAC of s2.

C. The router should be pingable, and should generate an ICMP echo reply in response to an ICMP echo request.

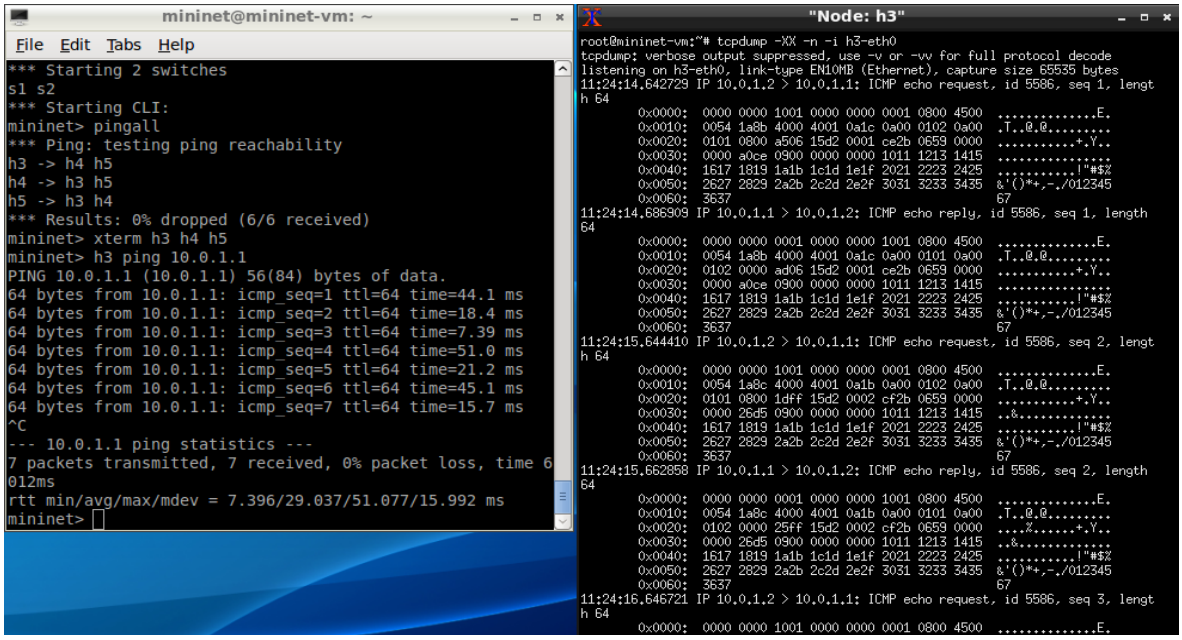

Fig.22 h3 pings s1

D. All hosts must be connected to each other. This can be verified using 'pingall'.

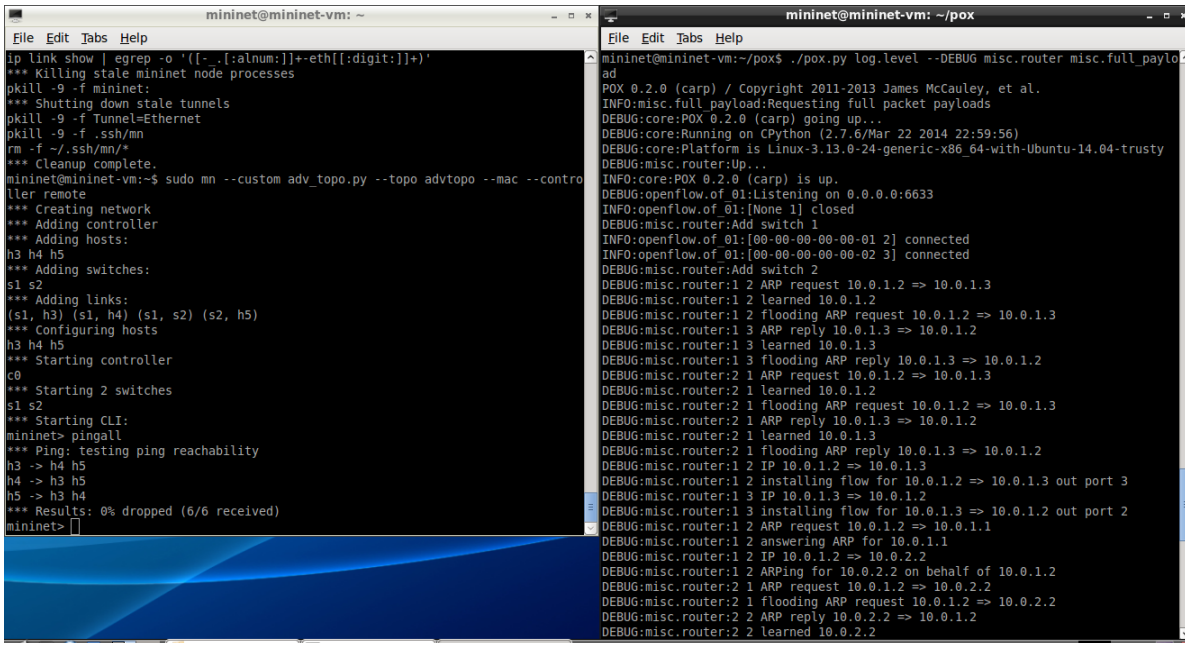

Fig.23 Verifying reachability

2.2 Create Firewall

In this exercise, we modify switch to reject connection attempts to specific ports, just like a firewall. The implementation is simple: just add an ofp_flow_mod with empty action to drop the matched packets.

To run iperf on xterms, on the xterm for h3, start up a server:

```
$ iperf -s
```

Then on the xterm for h4 start up a client:

```
$ iperf -c (IP of h3)
```

Task is to prevent this from happening, and to block all flow entries with this particular port. We change the port used by iperf with the -p option. Both server and client will need this option specified. The result is as following:

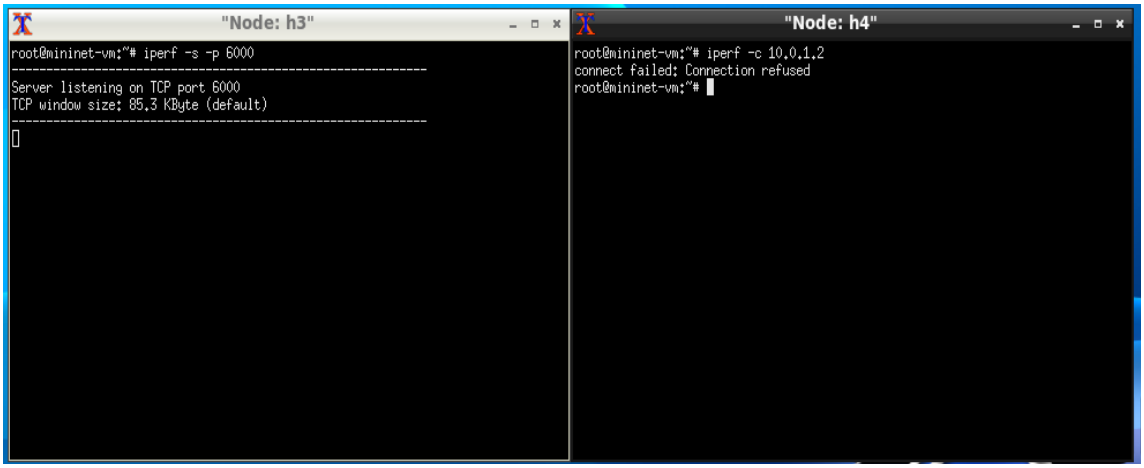

Fig.24 Port 6000 is blocked when h3 uses

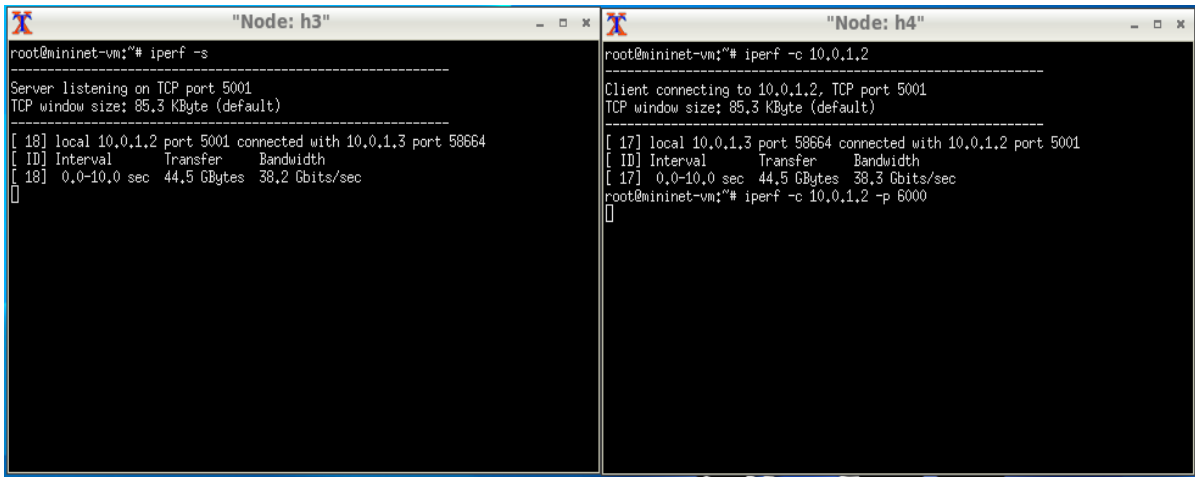

Fig.25 Port 6000 is blocked when h4 uses

According to Fig.24 and 25, the TCP traffic via port 6000 has been blocked both in the server and client.

III. Bonus

3.1 Topology of our own

In this exercise, Repeat the Advanced Topology section, but this time using a topology of our own. Specifically, we need to use a topology that satisfies the following requirements:

- Our topology has at least 3 routers.
- Each router has at least 1 host attached.

Our topology has 6 hosts and three switches in total as following:

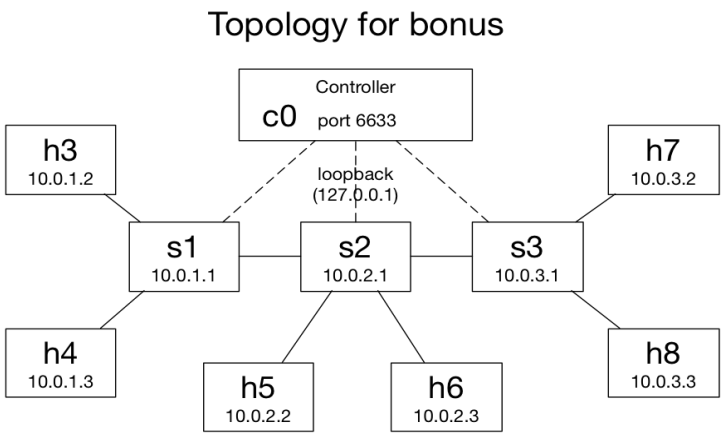

Fig.26 Topology for bonus

A. Attempts to send from a host to an unknown address range should yield an ICMP destination unreachable message.

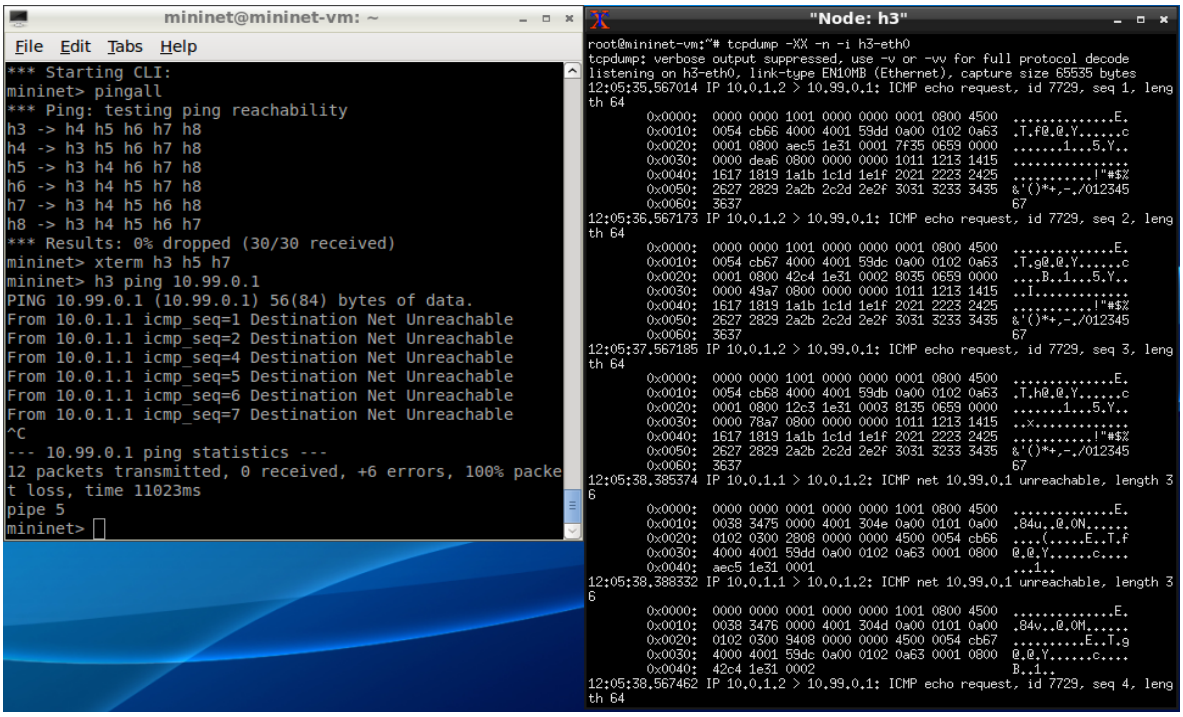

Fig.27 h3 pings an unknown IP

B. Packets sent to hosts on a known address range should have their MAC dst field changed to that of the next-hop router.

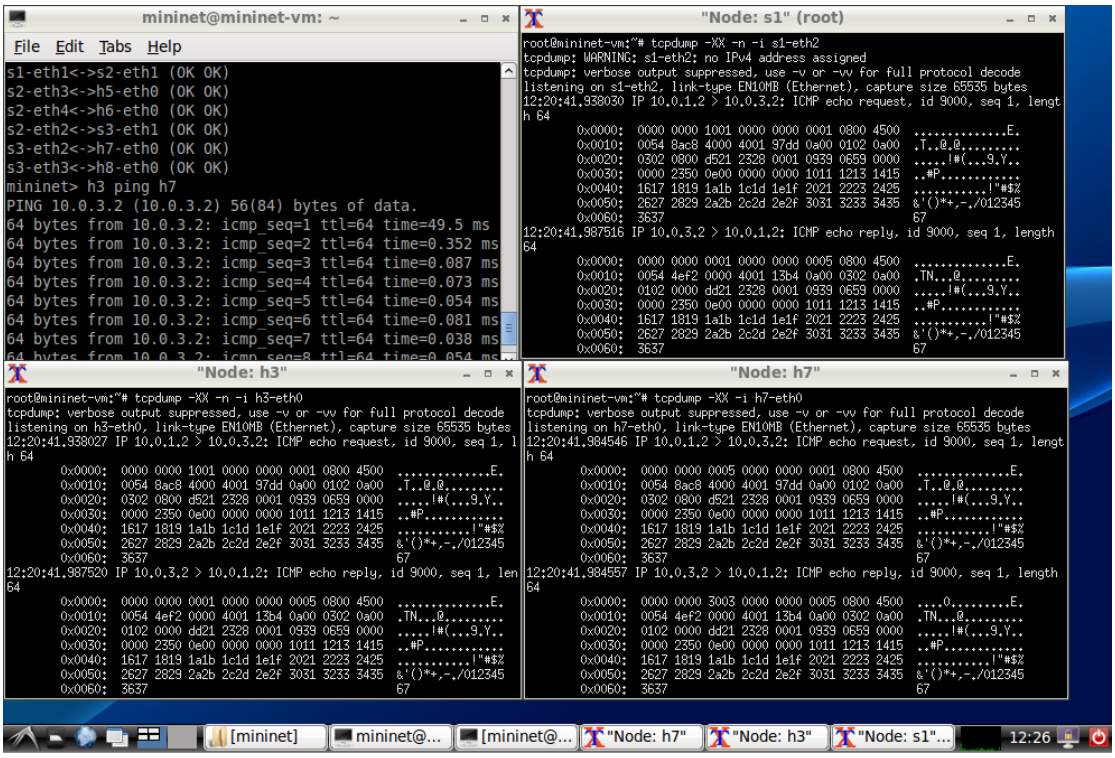

Fig.28 h3 pings h7

C. The router should be pingable, and should generate an ICMP echo reply in response to an ICMP echo request.

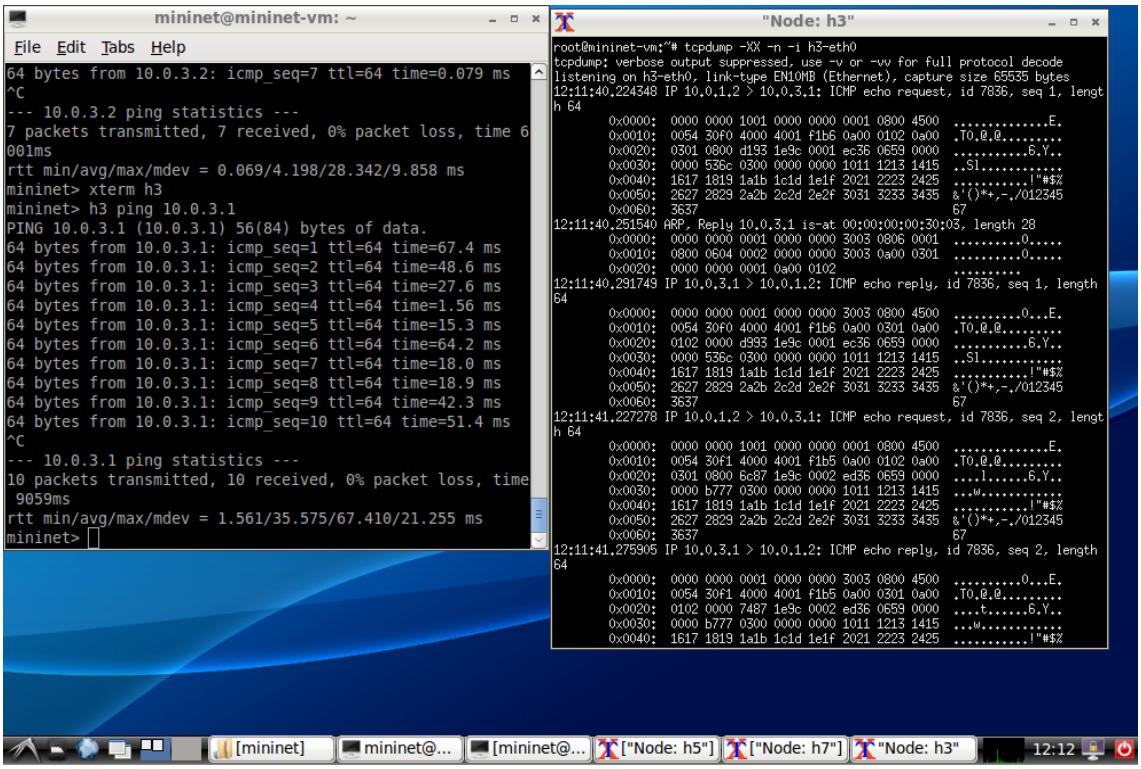

Fig.29 h3 pings s3

D. All hosts must be connected to each other. This can be verified as using 'pingall'.

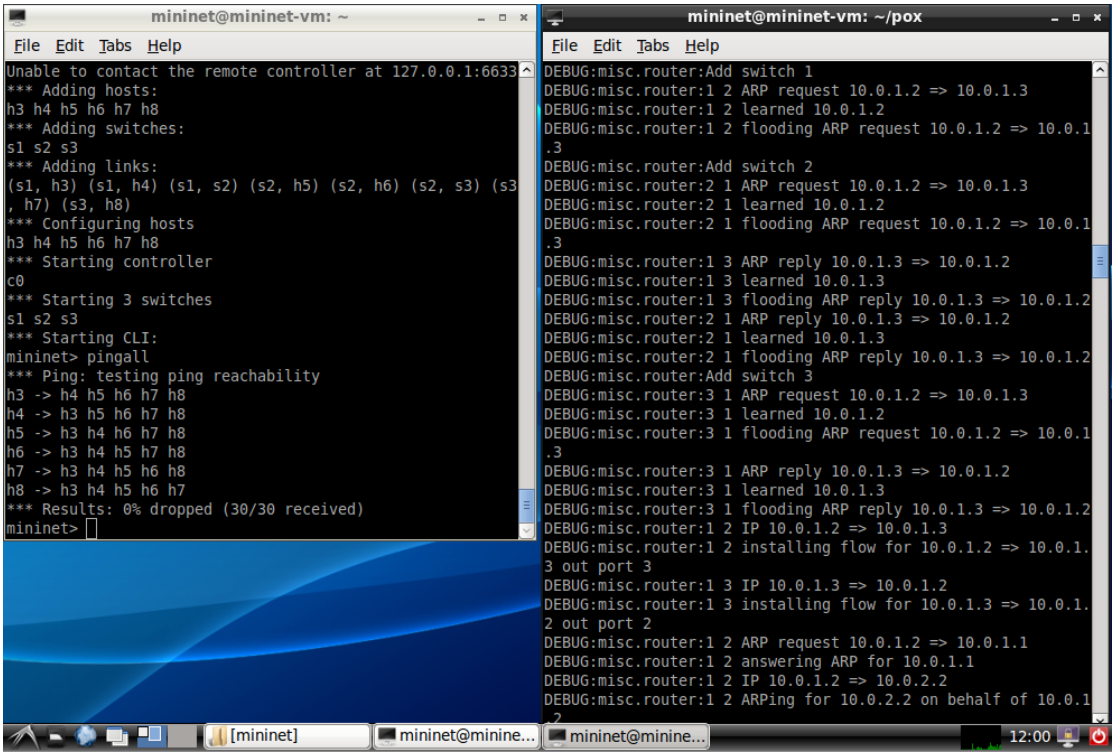

Fig.30 Verifying reachability

IV. Conclusion

This report states the steps from the creation of virtual machine to the implementation of an openflow router with Openflow and Mininet. First, A learning switch is created and has better performance than the example hub. In addition, the 1-router topology and the advanced topology with 2 routers are implemented and verified, and the function of firewall is added. Finally, a more complex advanced topology is created and tested, which contains 3 routers and 6 hosts.

V. References

[1] https://github.com/mininet/openflow-tutorial/wiki