Hand-on Experience with SDN Tools and Applications

ELG 7187B – Software Defined Networking and Cloud
Winter 2019
School of Electrical Engineering and Computer Science
University of Ottawa

This report was prepared for professor Ahmed Karmouch in partial fulfillment of the requirements for the course ELG7187B

Submitted By:
Isaac Ampratwum
Joshua Okosun

Problem 1: Discover Mininet

1. The default topology contains 4 nodes.
2. a. The default topology contains one (1) switch

   b. Yes there is a controller connected to the switch

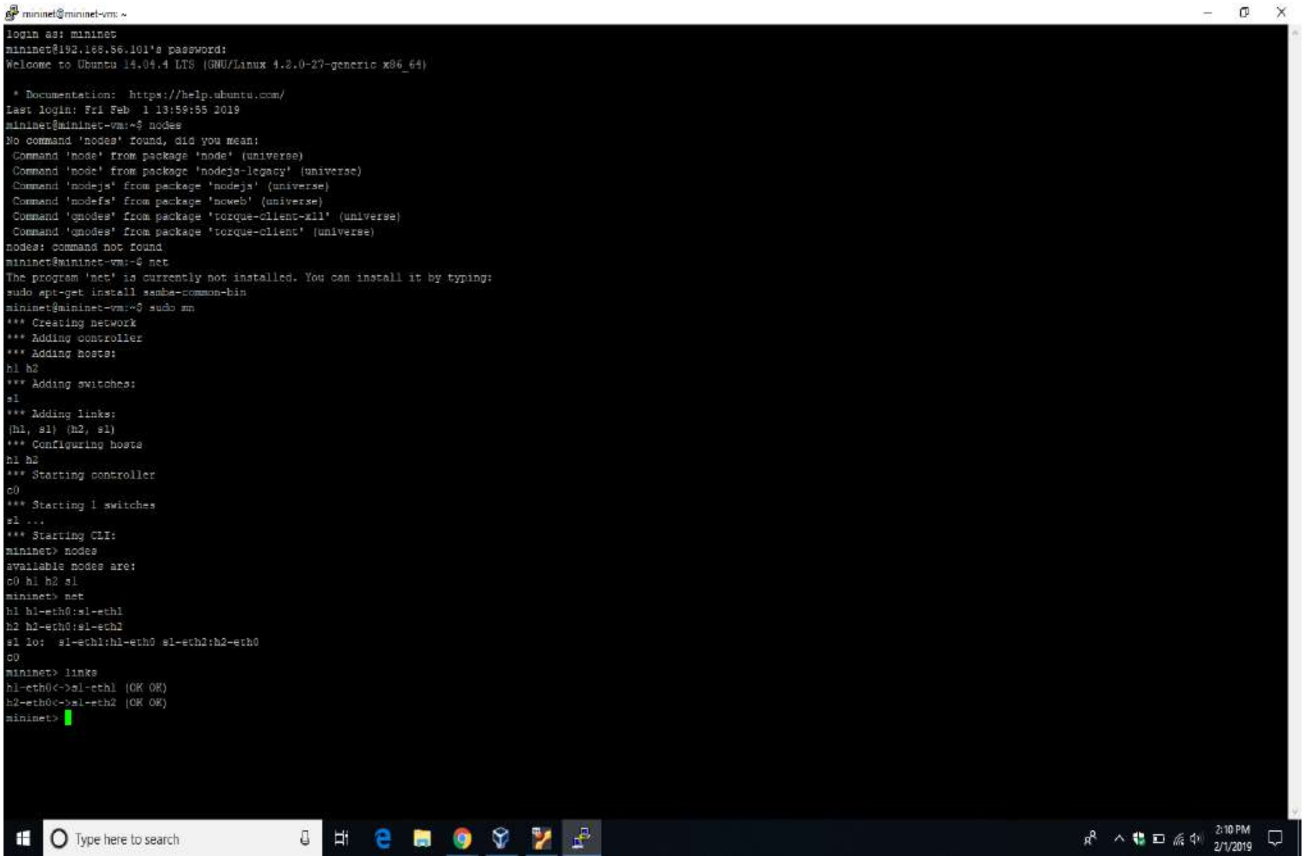
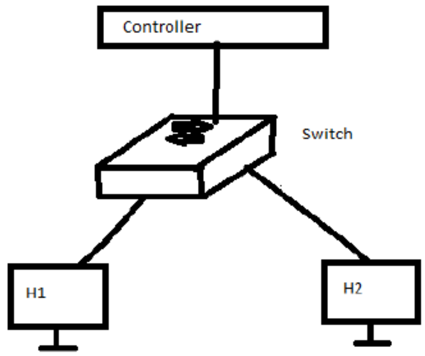


Fig 1. Screenshot of default topology created in mininet

3. The network consists of a controller, a switch and 2 hosts as indicated below



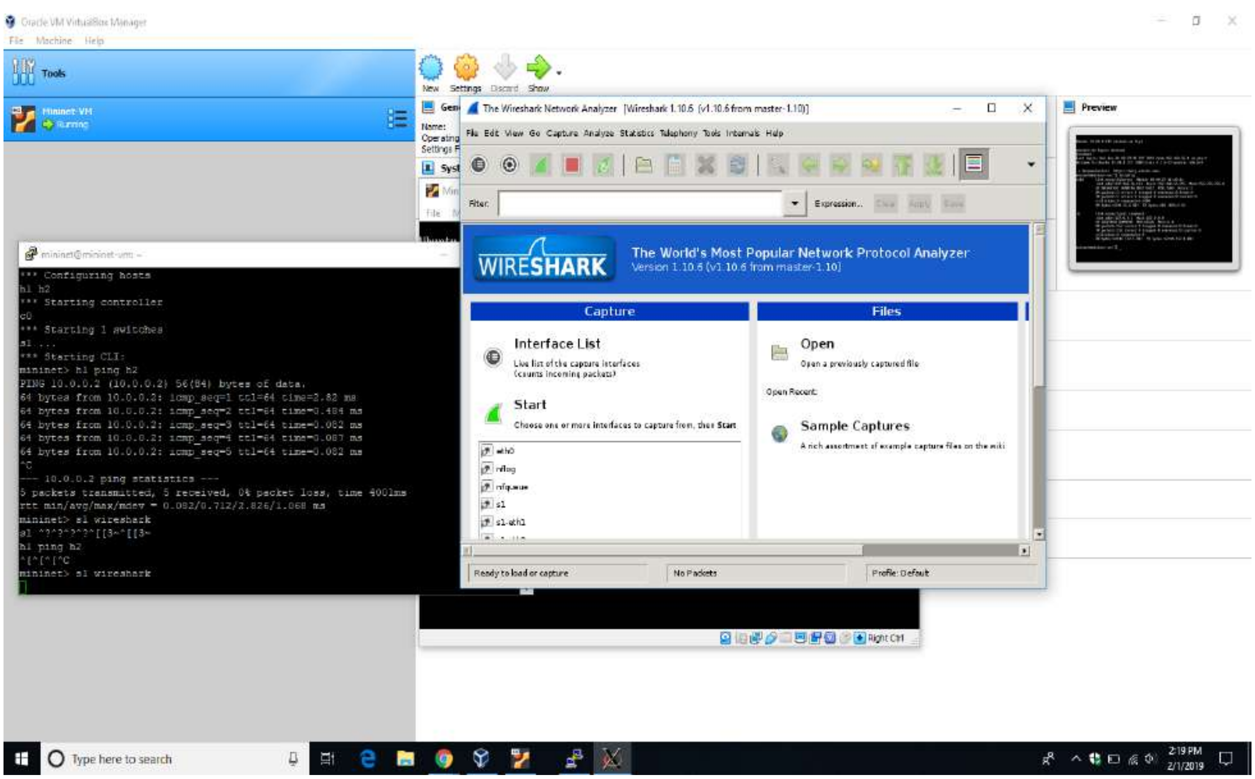
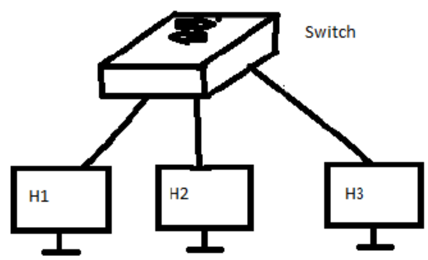


Fig 2. Screenshot of Wireshark launched in Mininet using s1 wireshark & command

Problem 2: Manual Configuration of the switches

4. Sketch topology—The new network has no controller and consists of 1 switch and 3 hosts as shown below



5. Performing a ping from h1 to h2 does not work because there is no controller connected by default and flow rules have not been installed manually as well.

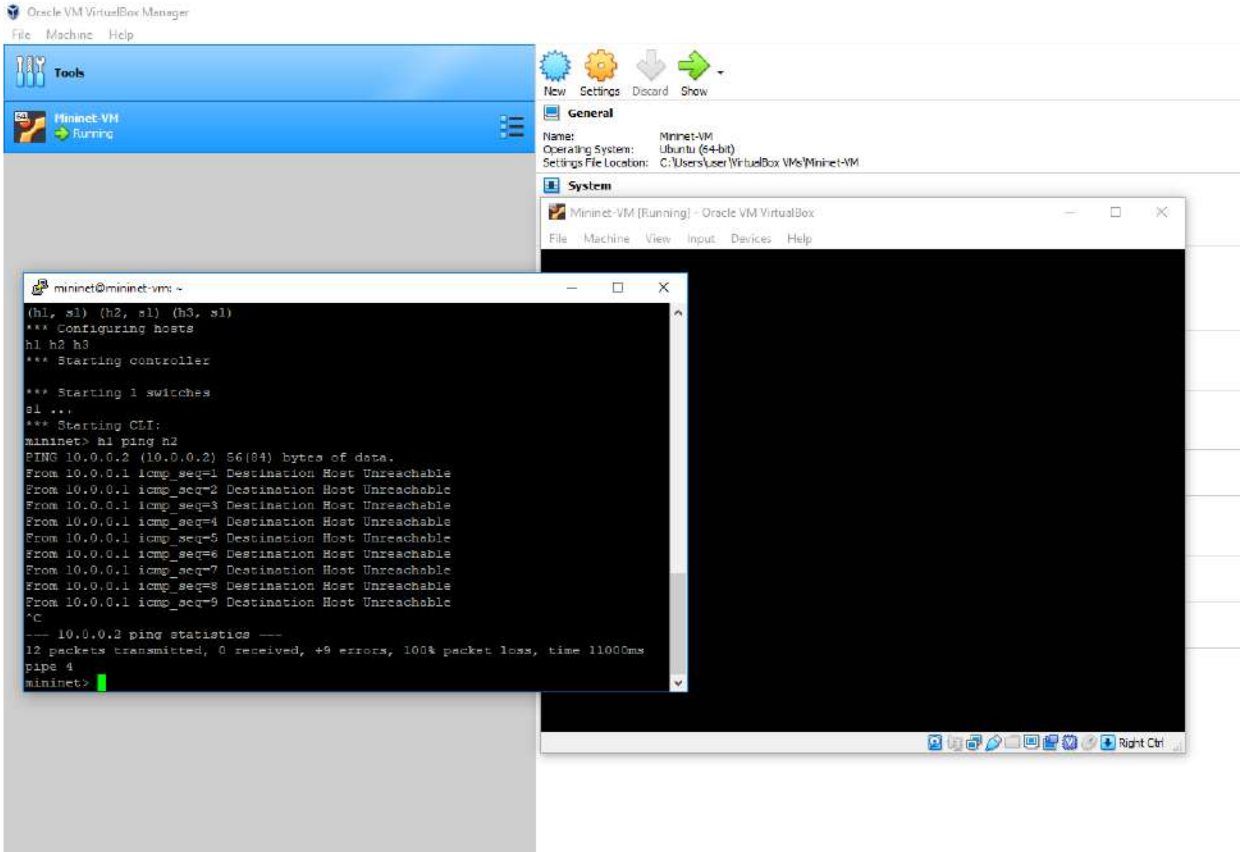


Fig 3: Screenshot of h1 pinging h2 when flow tables have not been installed.

6. The flow table of S1 is empty

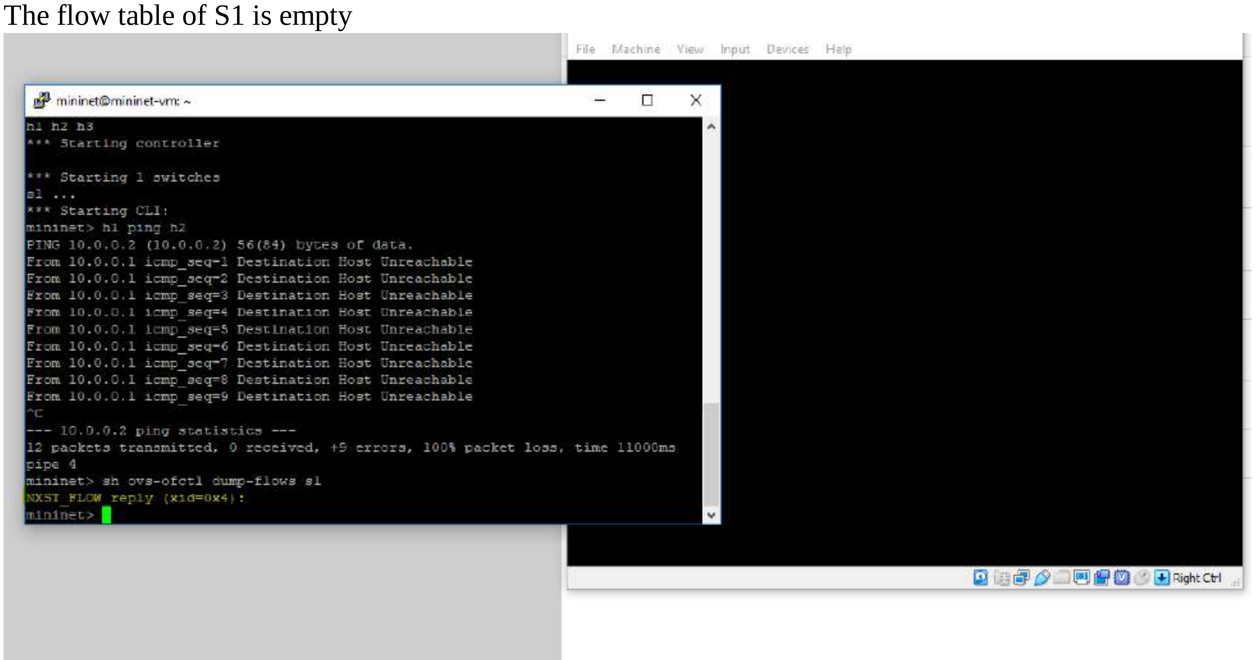


Fig 4: Screenshot of empty flow table

7. The matching rules for the table are either installed by a controller or configured manually. However, in this instance, there is no controller connected to the switch and flow tables haven't been installed manually, that is why it is empty.

8. a. Ethernet source and destination MAC address (Ethernet protocol)
   b. Source and destination IPv4 address (IPv4 protocol)
   c. UDP or TCP source and destination ports (user datagram or transport control protocol)

9. It implies that packets that come in through port 1 on the switch should be forwarded out through port 2 and vice versa.

10. Yes, it works

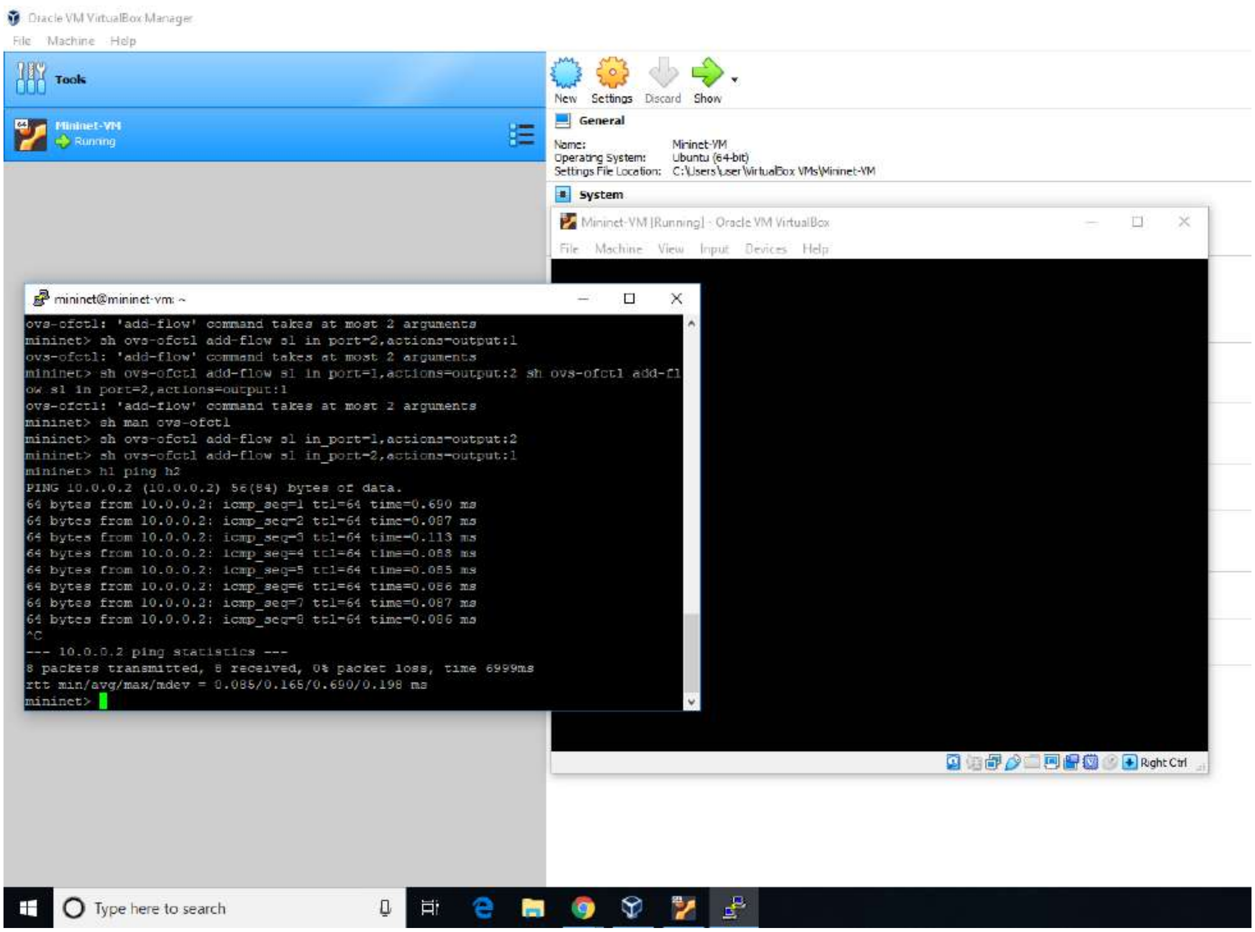


Fig 5: Screenshot of h1 pinging h2 after installing a matching rule.

Problem 3: Manual Configuration of a layer 2 forwarding

Commands for Layer 2 forwarding

We first use the ***ifconfig*** commands for **h1,h2,h3** to learn the MAC address of the hosts.

Also we use the ***sh ovs-ofctl show s1*** command to determine the switch ports the hosts are connected to

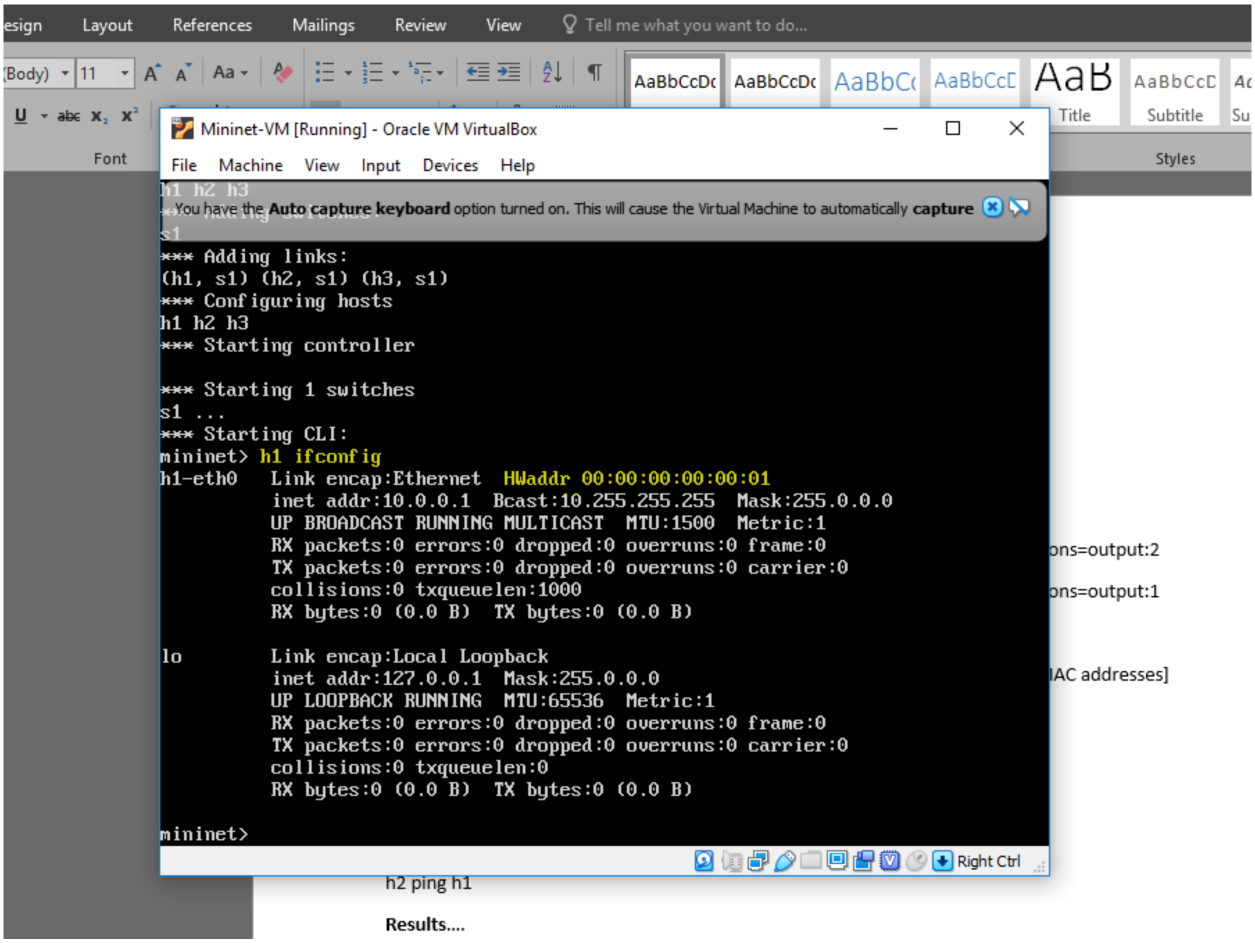


Fig 6: Screenshot of using the ifconfig command to learn the MAC address for host 1

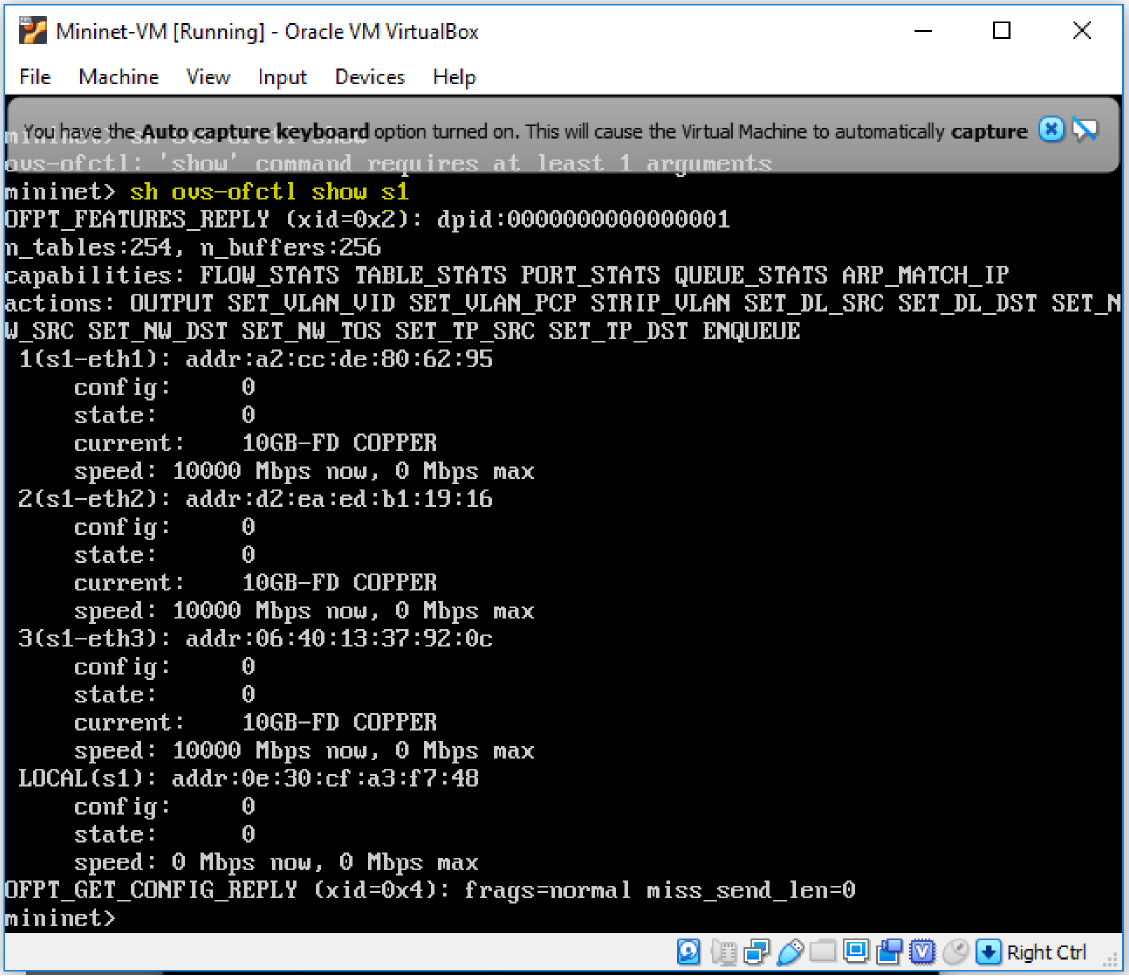


Fig 7: display of the switch ports of the hosts

Next step is to install the flow stables to implement layer 2 forwarding on Switch 1 using the **sh ovs-ofctl add-flow** command. Below is the list of commands used to map the layer 2 addresses of all the hosts in the network topology

sh ovs-ofctl add-flow s1 dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,actions=output:2
sh ovs-ofctl add-flow s1 dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:01,actions=output:1
sh ovs-ofctl add-flow s1 dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:03,actions=output:3
sh ovs-ofctl add-flow s1 dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:01,actions=output:1
sh ovs-ofctl add-flow s1 dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:03,actions=output:3
sh ovs-ofctl add-flow s1 dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:02,actions=output:2

We then add the command below to **implement the Address Resolution protocol** so that the hosts will be

sh ovs-ofctl add-flow s1 dl_type=0x806,nw_proto=1,actions=flood

[0x806 is the ethertype value for ARP and nw_proto=1 implies an ARP request]

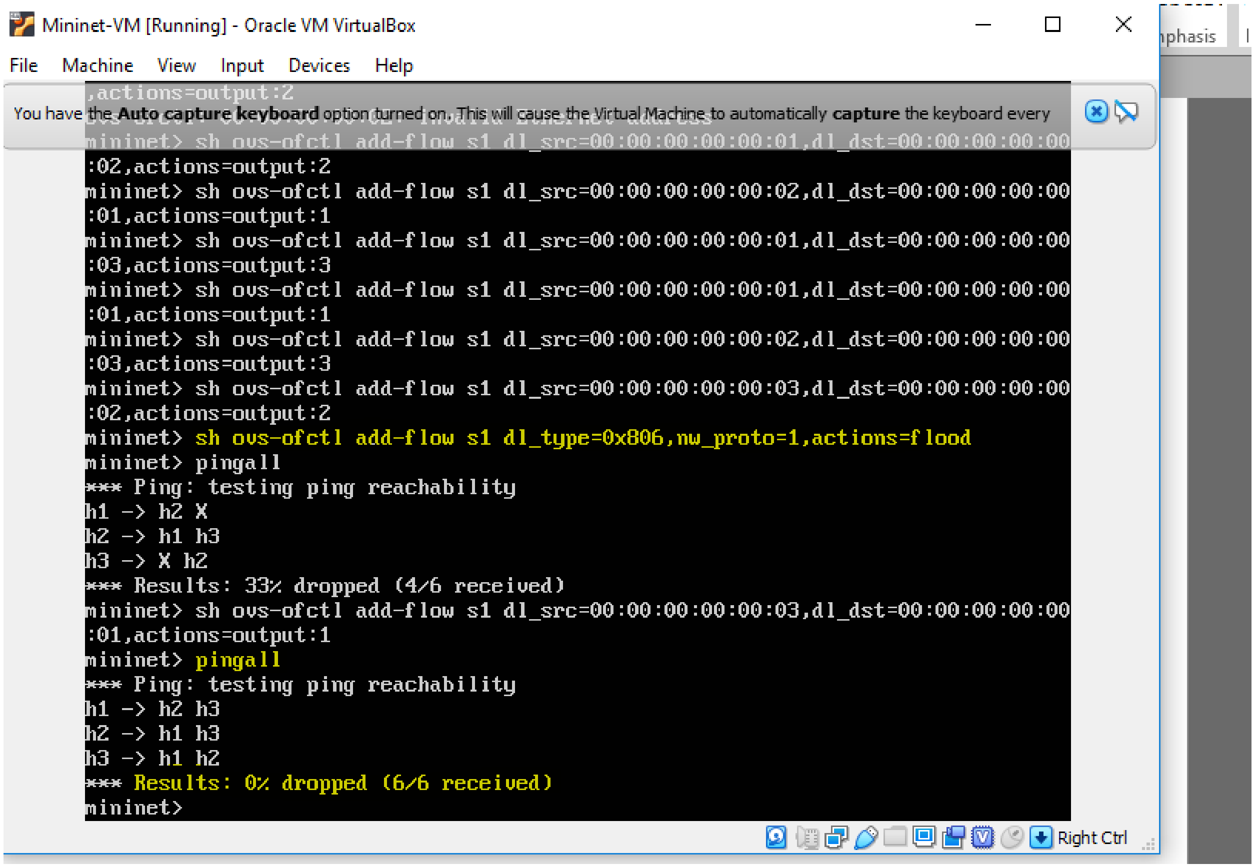


Fig 8: Flow table installation and test methodology

Test

Pingall command

Results

100% reachability

Success

Problem 4: Manual Configuration of a layer 3 forwarding

Commands for layer 3 forwarding

We first use the ***ifconfig*** command to learn the IP addresses for **h1,h2 and h3**

sh ovs-ofctl add-flow s1 dl_type=0x800, nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24,actions=normal

This command is used to install flows for IPv4 forwarding. The source and destination address are generalized as any address within the /24 subnet mask in the network. The 0x800 is the ethertype value for IP

sh ovs-ofctl add-flow s1 dl_type=0x806,nw_dst=10.0.0.1,actions=output:1
sh ovs-ofctl add-flow s1 dl_type=0x806,nw_dst=10.0.0.2,actions=output:2
sh ovs-ofctl add-flow s1 dl_type=0x806,nw_dst=10.0.0.2,actions=output:3

The above commands are used to implement ARP requests. This time, the ARP requests are not flooded in the network but rather specific for each IP address. The specific port for each ARP request based on the IP addresses have been given.

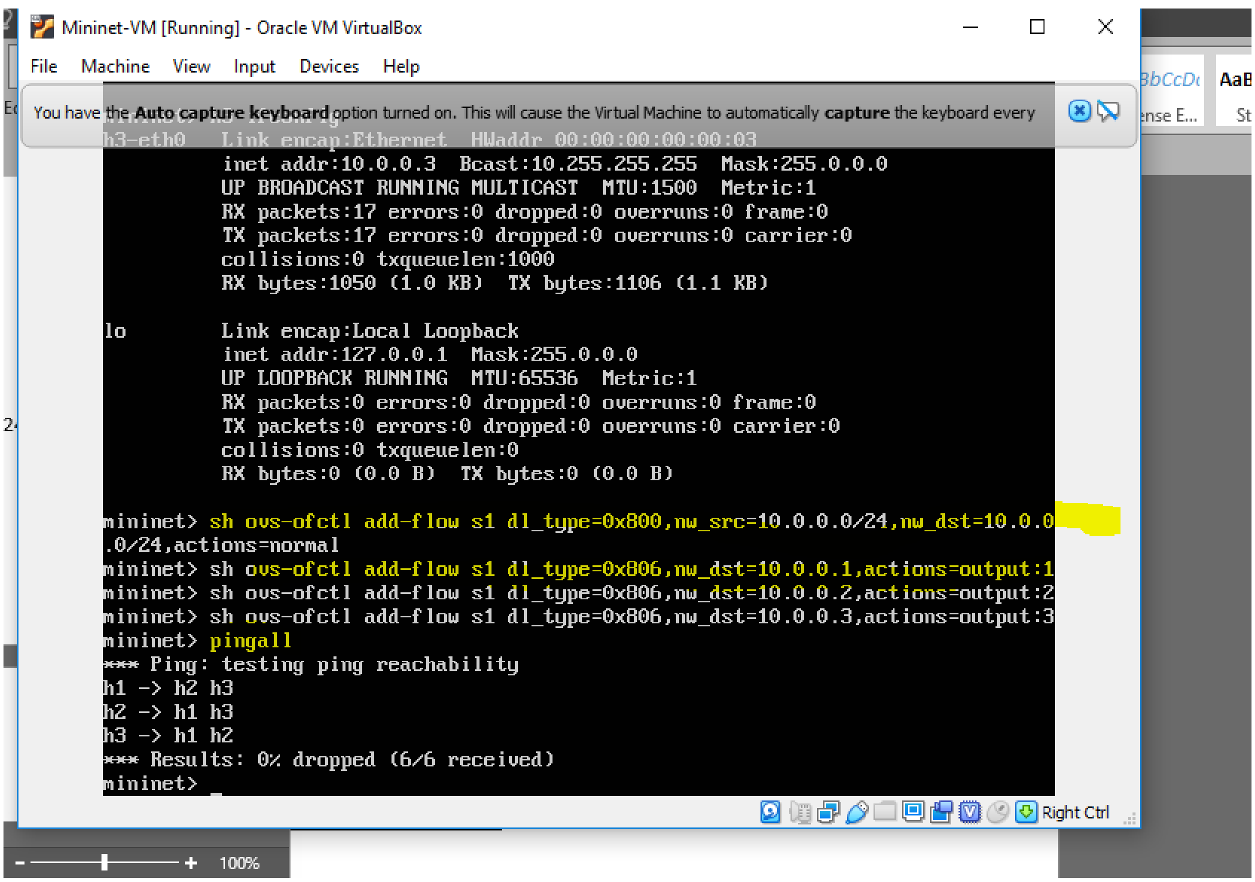


Fig 9: Adding flow tables, ARP requests and test methodology

Test

Pingall

Results

100% reachability

Success

<u>Problem 5: Manual Configuration of a layer 4 forwarding</u>

Commands for layer 4 forwarding

h3 python –m SimpleHTTPServer 80 &

Used to start a webserver on h3

sh ovs-ofctl add-flow s1 dl_type=0x806,actions=normal

The above command is used to implement ARP

sh ovs-ofctl add-flow s1 dl_type=0x800,nw_proto=6,tp_dst=80,actions=output:3

The command implies every HTTP traffic coming to the switch is routed to the server(h3) through port 3. The nw_proto=6 is for TCP and tp_dst=80 means transport layer destination port 80.

sh ovs-ofctl add-flow s1 dl_type=0x800,nw_src=10.0.0.3,actions=normal

The command routes the return traffic from the server to the other 2 hosts.

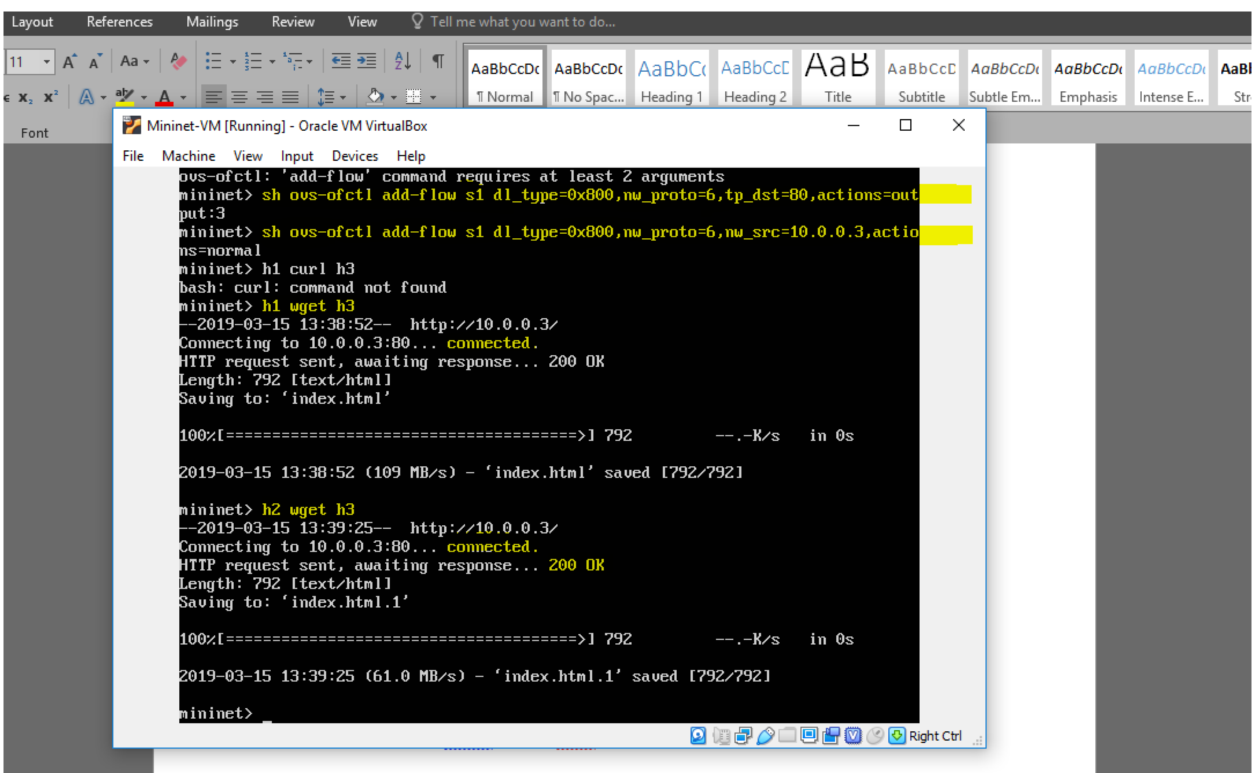


Fig 10: Commands and test methodology for layer 4 forwarding

Test

h1 wget h3

h2 wget h3

Results

Connected, HTTP request sent!.

Problem 6: Manual Configuration of a layer 3 forwarding with a firewall

We first use the ***ifconfig*** command to learn the IP addresses for **h1,h2 and h3**

sh ovs-ofctl add-flow s1 dl_type=0x800,nw_src=10.0.0.1,nw_dst=10.0.0.2,actions=normal
sh ovs-ofctl add-flow s1 dl_type=0x800,nw_src=10.0.0.2,nw_dst=10.0.0.1,actions=normal
sh ovs-ofctl add-flow s1 dl_type=0x800,nw_src=10.0.0.1,nw_dst=10.0.0.3,actions=drop
sh ovs-ofctl add-flow s1 dl_type=0x800,nw_src=10.0.0.3,nw_dst=10.0.0.1,actions=normal
sh ovs-ofctl add-flow s1 dl_type=0x800,nw_src=10.0.0.2,nw_dst=10.0.0.3,actions=drop
sh ovs-ofctl add-flow s1 dl_type=0x800,nw_src=10.0.0.3,nw_dst=10.0.0.2,actions=normal

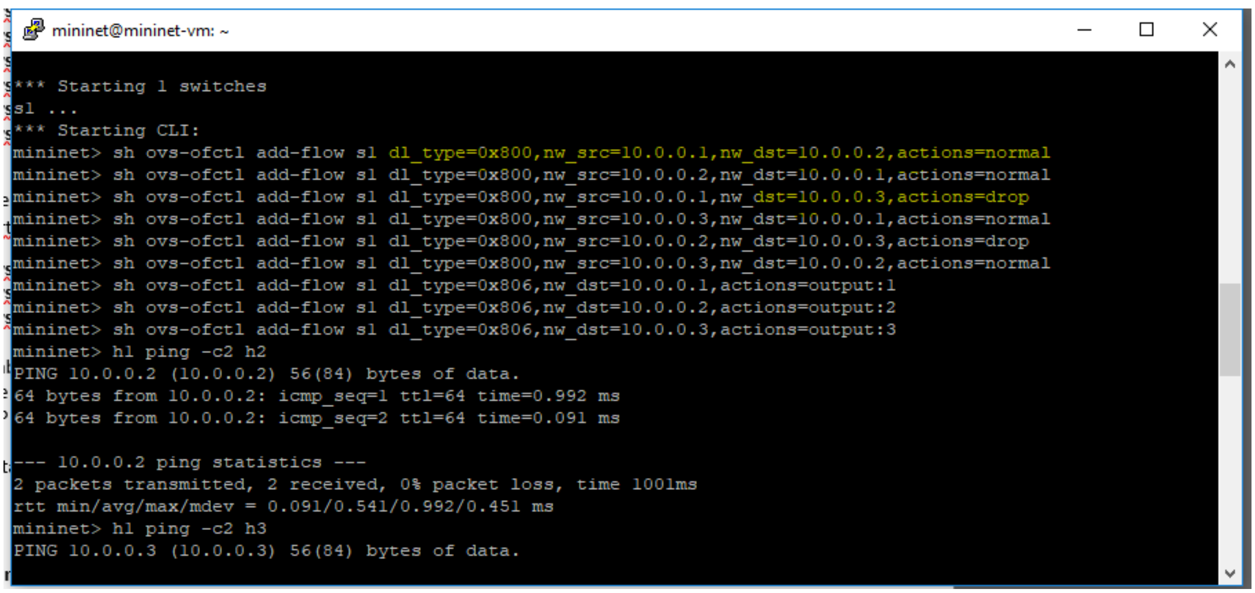


Fig: screenshot showing flow rules

These commands are used to install an IPv4 forwarding and to drop all traffic to host 3. The 0x800 is the ethertype value for IP

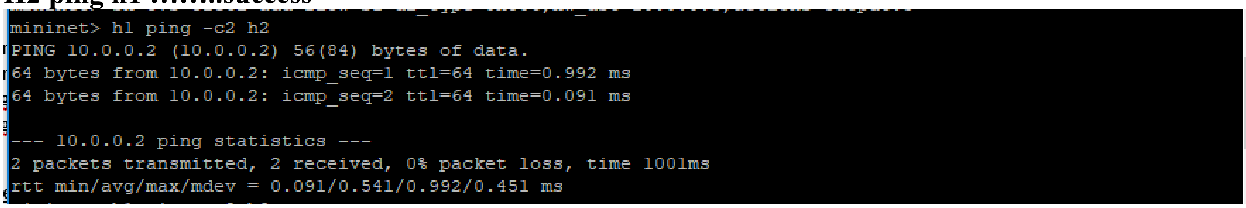
sh ovs-ofctl add-flow s1 dl_type=0x806,nw_dst=10.0.0.1,actions=output:1
sh ovs-ofctl add-flow s1 dl_type=0x806,nw_dst=10.0.0.2,actions=output:2
sh ovs-ofctl add-flow s1 dl_type=0x806,nw_dst=10.0.0.2,actions=output:3

The above commands are used to implement ARP requests. This time, the ARP requests are not flooded in the network but rather specific for each IP address. The specific port for each ARP request based on the IP addresses have been given.

We start a web browser on h3 using **h3 python –m SimpleHTTPServer 80 &** command

Test

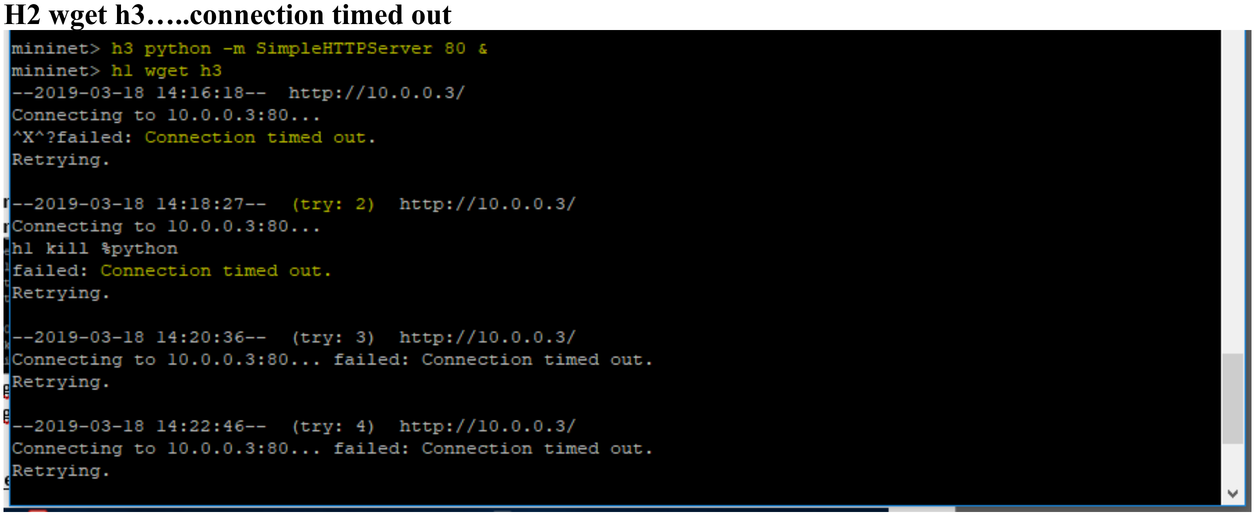
H1 ping h2 ……..success
H2 ping h1 ……..success



```
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_seq=1 ttl=64 time=0.992 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.091 ms

--- 10.0.0.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.091/0.541/0.992/0.451 ms
```

H1 wget h3…….connection timed out
H2 wget h3…..connection timed out



```
mininet> h3 python -m SimpleHTTPServer 80 &
mininet> h1 wget h3
--2019-03-18 14:16:18--  http://10.0.0.3/
Connecting to 10.0.0.3:80...
^X^?failed: Connection timed out.
Retrying.

--2019-03-18 14:18:27--  (try: 2)  http://10.0.0.3/
Connecting to 10.0.0.3:80...
h1 kill %python
failed: Connection timed out.
Retrying.

--2019-03-18 14:20:36--  (try: 3)  http://10.0.0.3/
Connecting to 10.0.0.3:80... failed: Connection timed out.
Retrying.

--2019-03-18 14:22:46--  (try: 4)  http://10.0.0.3/
Connecting to 10.0.0.3:80... failed: Connection timed out.
Retrying.
```

Problem 7: Using a remote controller

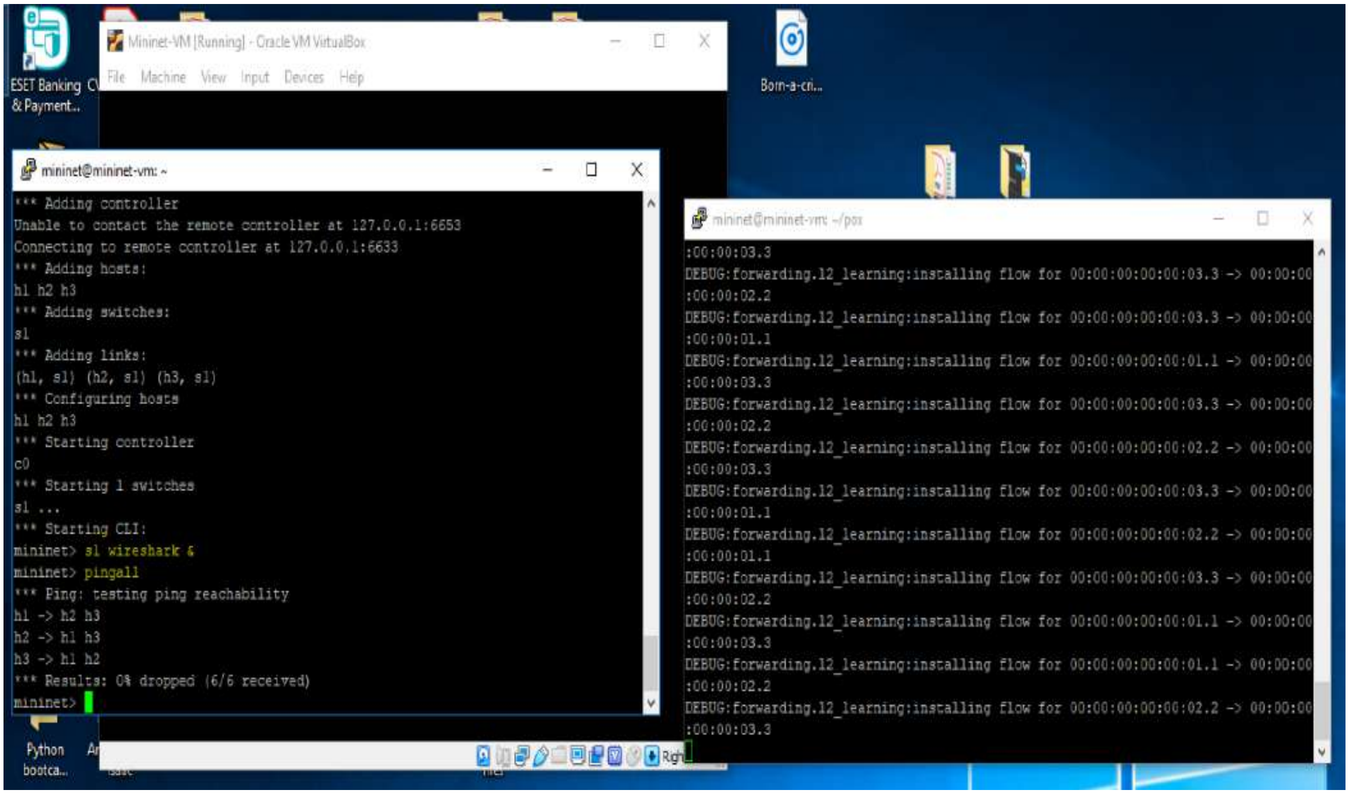


Fig : Screenshot of topology terminal and remote controller(pox) terminal

The openflow packet is an ARP packet. It is a broadcast packet flooded in the network in order to match a particular IP address to the physical address of the destination.

The packet consists of the

- Source address field
- The destination address field which is a broadcast address
- Protocol type which is ARP

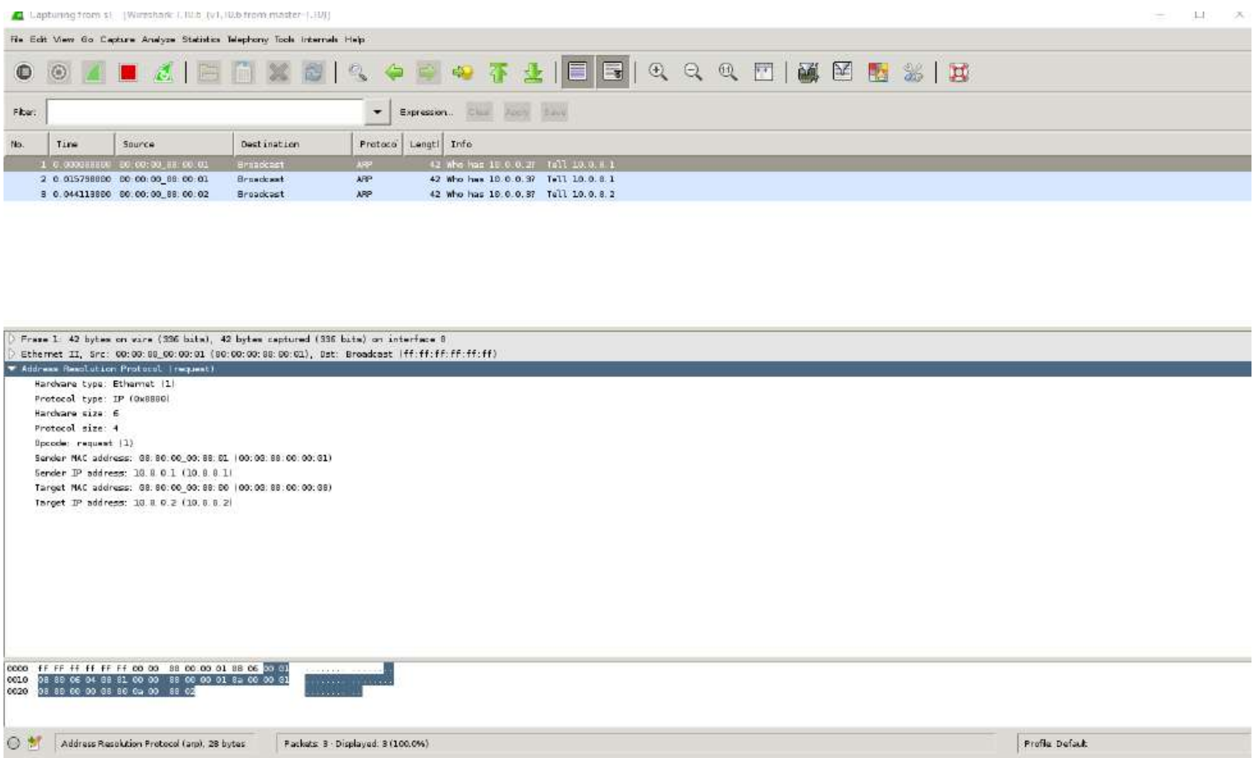


Fig : Wireshark capture of the ARP packet

How the l2_learning algorithm works

For each packet that comes to the switch, it examines the source MAC address of the packet and determine the output ports corresponding to packets sent from that MAC address

- Use source address and switch port to update address/port table
- Drop certain kinds of packets like packets with bridge filtered destination addresses
- Check if destination is multicast and flood if so
- Check if destination address is in address/port table, if not, flood packet as a hub
- If output port = input port, drop packet to avoid loops
- Install appropriate flow entry into switch based on source MAC address and port to send packet out the mapped output port

The network behavior it implements is **Routing**

Problem 8: Create a custom topology

Python Script for the diamond topology displayed

```
#To import modules
from mininet.topo import Topo
class FirstTopo(Topo):
        def build(self):

    # Create templates
    abouthost = {'inNamespace':True}
    bw_one = {'bw': 1}
    bw_two = {'bw': 10}

    # To Create host nodes (h1, h2, h3, h4)
    for i in range(4):
       self.addHost('h%d' % (i+1), **abouthost)

    # To Create switch nodes
    self.addSwitch('s1',dpid='0000000000000001')
    self.addSwitch('s2',dpid='0000000000000002')
    self.addSwitch('s3',dpid='0000000000000003')
    self.addSwitch('s4',dpid='0000000000000004')


    # To Add switch links
    self.addLink('s1', 's2', **bw_one)
    self.addLink('s2', 's4', **bw_one)
    self.addLink('s1', 's3', **bw_two)
    self.addLink('s3', 's4', **bw_two)

    # To Add host links
    self.addLink('h1', 's1')
    self.addLink('h2', 's1')
    self.addLink('h3', 's4')
    self.addLink('h4', 's4')

topos = { 'mytopo': ( lambda: FirstTopo())}
```

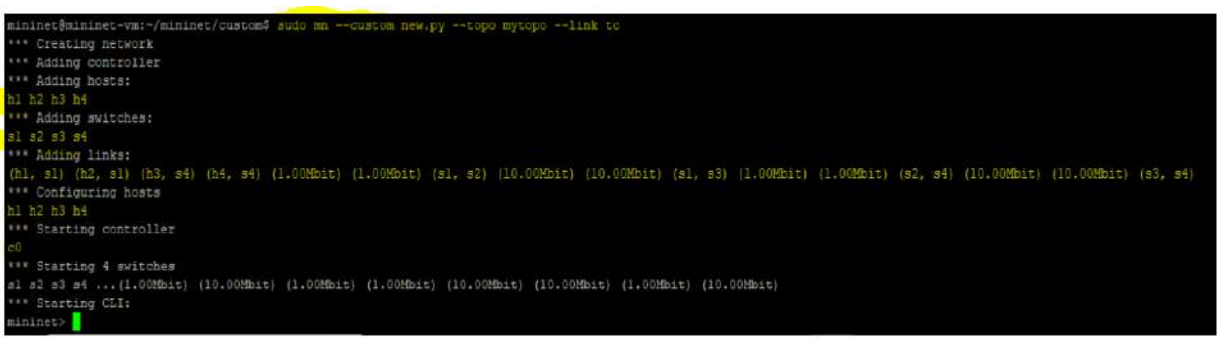


***Fig : Screenshot of the custom topology implemented in Mininet*.**

The command ***sudo mn –custom script_name.py –topo mytopo –link tc*** is used to load the topology and set up the link capacities.

Problem 9: Network slicing using flowvisor

Use ***fvctl -f /dev/null list-slices*** to list slices which outputs just one slice: the fvadmin slice

```
mininet@mininet-vm: ~
    "slice-name": "fvadmin"
  },
  "flow-stats-cache": 30,
  "flowmod-limit": {
    "fvadmin": {
      "00:00:00:00:00:00:00:01": -1,
      "00:00:00:00:00:00:00:02": -1,
      "00:00:00:00:00:00:00:03": -1,
      "00:00:00:00:00:00:00:04": -1,
      "any": null
    }
  },
  "host": "localhost",
  "log_facility": "LOG_LOCAL7",
  "log_ident": "flowvisor",
  "logging": "NOTE",
  "stats-desc": false,
  "track-flows": false,
  "version": "flowvisor-1.4.0"
}
mininet@mininet-vm:~$ fvctl -f /dev/null list-slices
Configured slices:
fvadmin          --> enabled
mininet@mininet-vm:~$
```

Use ***fvctl –f /dev/null list-flowspace*** to view flow entries

```
}
mininet@mininet-vm:~$ fvctl -f /dev/null list-slices
Configured slices:
fvadmin          --> enabled
mininet@mininet-vm:~$ fvctl -f /dev/null list-flowspace
Configured Flow entries:
  None
mininet@mininet-vm:~$
```

Use fvctl –f /dev/null list-datapaths to ensure and display connected switches

```
mininet@mininet-vm:~$ fvctl -f /dev/null list-datapaths
Connected switches:
  1 : 00:00:00:00:00:00:00:01
  2 : 00:00:00:00:00:00:00:02
  3 : 00:00:00:00:00:00:00:03
  4 : 00:00:00:00:00:00:00:04
mininet@mininet-vm:~$
```

We list the links with the ***fvctl –f /dev/null list-links command***

mininet@mininet-vm: ~

```
mininet@mininet-vm:~$ fvctl -f /dev/null list-links
[
  {
    "dstDPID": "00:00:00:00:00:00:00:02",
    "dstPort": "1",
    "srcDPID": "00:00:00:00:00:00:00:01",
    "srcPort": "1"
  },
  {
    "dstDPID": "00:00:00:00:00:00:00:01",
    "dstPort": "1",
    "srcDPID": "00:00:00:00:00:00:00:02",
    "srcPort": "1"
  },
  {
    "dstDPID": "00:00:00:00:00:00:00:04",
    "dstPort": "1",
    "srcDPID": "00:00:00:00:00:00:00:02",
    "srcPort": "2"
  },
  {
```

Creating the network slices

We add slices using the syntax

Fvctl add-slice [options]<slice name><controller-url><admin-email> and use ***fvctl -f /dev/null list-slices*** to confirm that the slices have been added

```
mininet@mininet-vm:~$ fvctl -f /dev/null add-slice upper tcp:localhost:10001 admin@upperslice
Slice password:
Slice upper was successfully created
mininet@mininet-vm:~$ fvctl -f /dev/null add-slice lower tcp:localhost:10002 admin@lowerslice
Slice password:
Slice lower was successfully created
mininet@mininet-vm:~$ fvctl -f /dev/null list-slices
Configured slices:
fvadmin         --> enabled
upper           --> enabled
lower           --> enabled
mininet@mininet-vm:~$
```

After creating the slices, we go ahead to create the **flow entries(space)**

Using the commands

fvctl -f /dev/null add-flowspace dpid1-port1 1 1 in_port=1 upper=7
fvctl -f /dev/null add-flowspace dpid1-port3 1 1 in_port=3 upper=7
fvctl -f /dev/null add-flowspace dpid2 2 1 any upper=7
fvctl -f /dev/null add-flowspace dpid4-port1 4 1 in_port=1 upper=7
fvctl -f /dev/null add-flowspace dpid4-port3 4 1 in_port=3 upper=7

We added ports 1 and 3 of switches 1 and 4 as well as all ports of switch 2 to the Upper Slice.

We confirmed that entries have been added with ***fvctl –f /dev/null list-flowspace***

```
mininet@mininet-vm:~$ fvctl -f /dev/null add-flowspace dpid1-port3 1 1 in_port=3 upper=7
FlowSpace dpid1-port3 was added with request id 7.
mininet@mininet-vm:~$ fvctl -f /dev/null add-flowspace dpid2 2 1 any upper=7
FlowSpace dpid2 was added with request id 8.
mininet@mininet-vm:~$ fvctl -f /dev/null add-flowspace dpid4-port3 4 1 in_port=3 upper=7
FlowSpace dpid4-port3 was added with request id 9.
mininet@mininet-vm:~$ fvctl -f /dev/null add-flowspace dpid4-port1 4 1 in_port=1 upper=7
FlowSpace dpid4-port1 was added with request id 10.
mininet@mininet-vm:~$ fvctl -f /dev/null list-flowspace
Configured Flow entries:
{"force-enqueue": -1, "name": "dpid1-port1", "slice-action": [{"slice-name": "upper", "permission": 7}], "queues"
: [], "priority": 1, "dpid": "00:00:00:00:00:00:00:01", "id": 7, "match": {"wildcards": 4194302, "in_port": 1}}
{"force-enqueue": -1, "name": "dpid1-port3", "slice-action": [{"slice-name": "upper", "permission": 7}], "queues"
: [], "priority": 1, "dpid": "00:00:00:00:00:00:00:01", "id": 8, "match": {"wildcards": 4194302, "in_port": 3}}
{"force-enqueue": -1, "name": "dpid2", "slice-action": [{"slice-name": "upper", "permission": 7}], "queues": [],
"priority": 1, "dpid": "00:00:00:00:00:00:00:02", "id": 9, "match": {"wildcards": 4194303}}
{"force-enqueue": -1, "name": "dpid4-port3", "slice-action": [{"slice-name": "upper", "permission": 7}], "queues"
: [], "priority": 1, "dpid": "00:00:00:00:00:00:00:04", "id": 10, "match": {"wildcards": 4194302, "in_port": 3}}
{"force-enqueue": -1, "name": "dpid4-port1", "slice-action": [{"slice-name": "upper", "permission": 7}], "queues"
: [], "priority": 1, "dpid": "00:00:00:00:00:00:00:04", "id": 11, "match": {"wildcards": 4194302, "in_port": 1}}
mininet@mininet-vm:~$
```

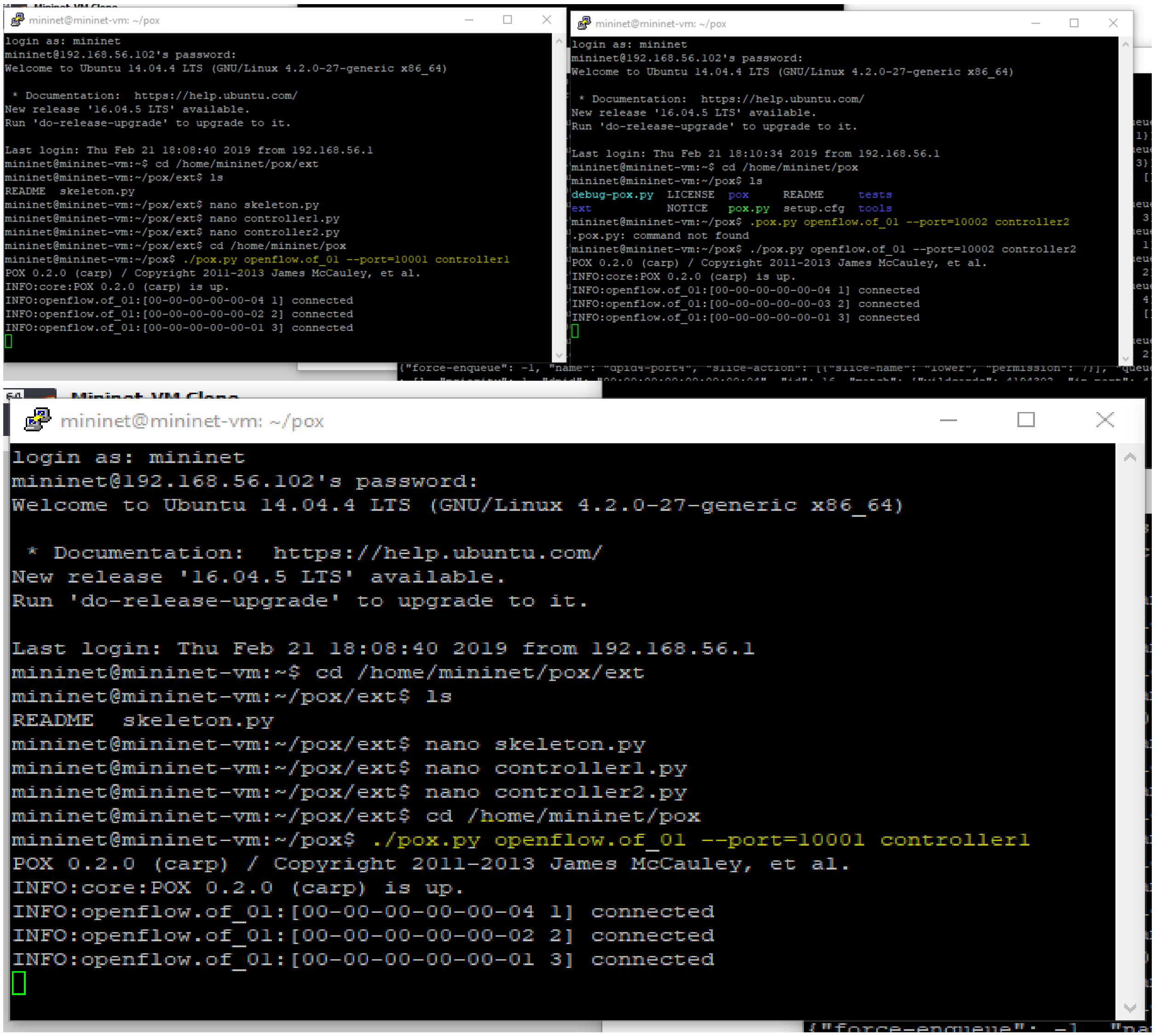
We then repeated the same approach to add ports 2 and 4 of switches 1 and 4 as well as all ports of switch 3 to the lower slice

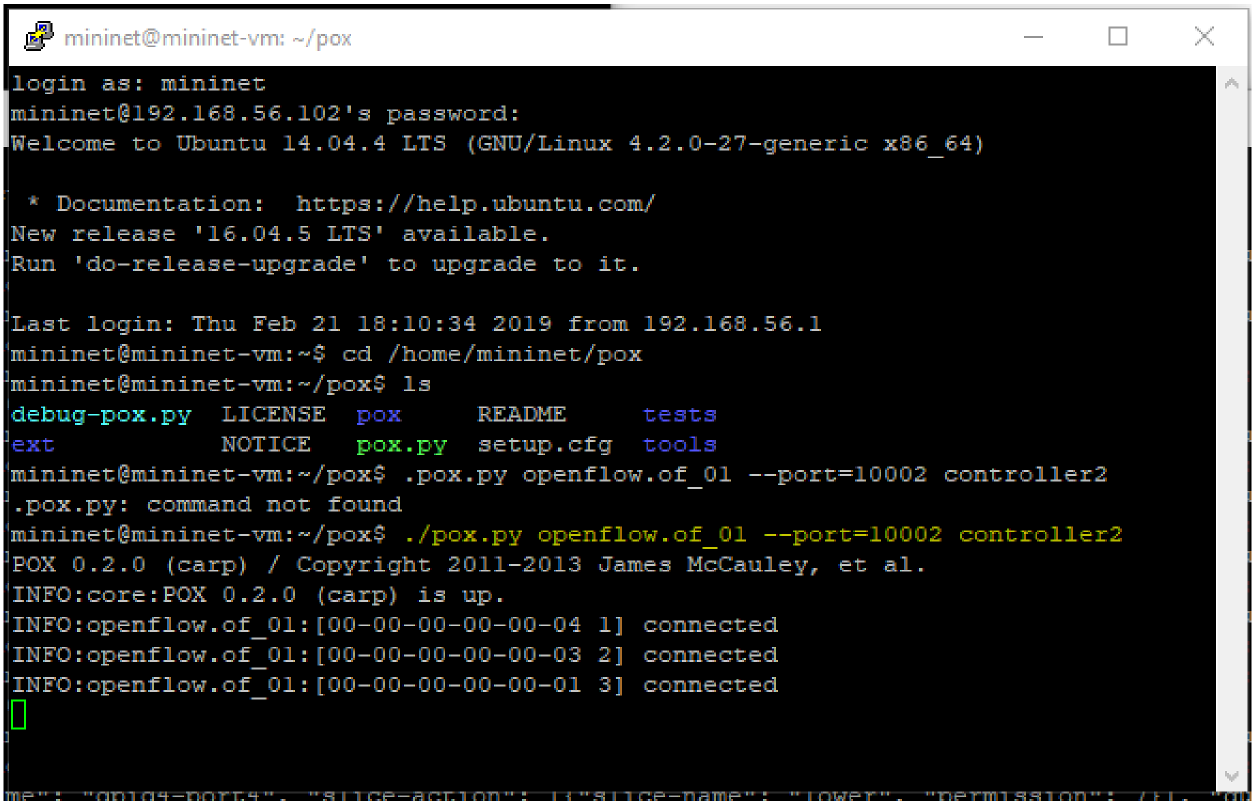
fvctl -f /dev/null add-flowspace dpid1-port2 1 1 in_port=2 lower=7
fvctl -f /dev/null add-flowspace dpid1-port4 1 1 in_port=4 lower=7
fvctl -f /dev/null add-flowspace dpid3 3 1 any lower=7
fvctl -f /dev/null add-flowspace dpid4-port2 4 1 in_port=2 lower=7
fvctl -f /dev/null add-flowspace dpid4-port4 4 1 in_port=4 lower=7

Next we open 2 new terminals to implement 2 POX controllers
cd /home/mininet/pox/ext
nano controller1.py to create a controller based on the l2_learning python script and save
nano controller2.py to create the second controller based on the l2_learning python script

In Terminal 1, we enter
cd /home/mininet/pox
./pox.py openflow.of_01 –port=10001 contoller1 to connect controller1 to the upper slice and

In Terminal 2,
cd /home/mininet/pox
./pox.py openflow.of_01 –port=10002 contoller2 to connect controller2 to the lower slice

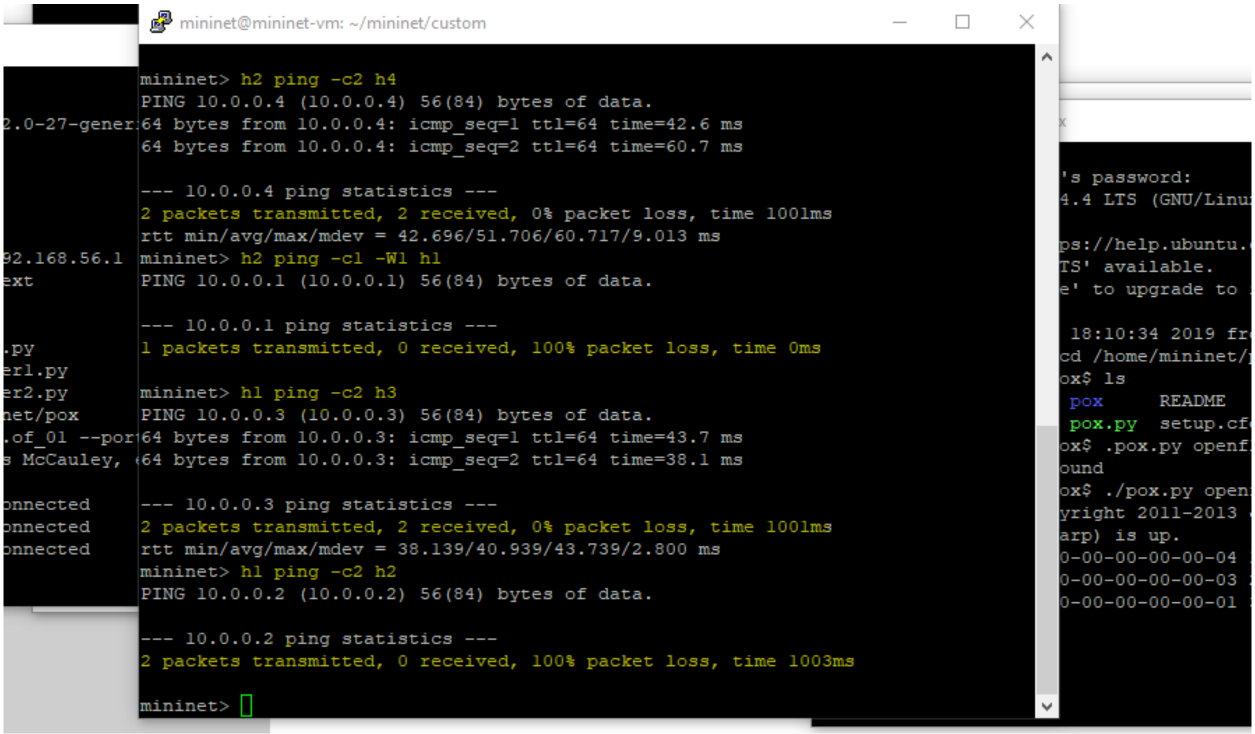

mininet@mininet-vm: ~/pox
login as: mininet
mininet@192.168.56.102's password:
Welcome to Ubuntu 14.04.4 LTS (GNU/Linux 4.2.0-27-generic x86_64)
* Documentation: https://help.ubuntu.com/
New release '16.04.5 LTS' available.
Run 'do-release-upgrade' to upgrade to it.
Last login: Thu Feb 21 18:08:40 2019 from 192.168.56.1
mininet@mininet-vm:~$ cd /home/mininet/pox/ext
mininet@mininet-vm:~/pox/ext$ ls
README skeleton.py
mininet@mininet-vm:~/pox/ext$ nano skeleton.py
mininet@mininet-vm:~/pox/ext$ nano controller1.py
mininet@mininet-vm:~/pox/ext$ nano controller2.py
mininet@mininet-vm:~/pox/ext$ cd /home/mininet/pox
mininet@mininet-vm:~/pox$ ./pox.py openflow.of_01 --port=10001 controller1
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
INFO:core:POX 0.2.0 (carp) is up.
INFO:openflow.of_01:[00-00-00-00-00-04 1] connected
INFO:openflow.of_01:[00-00-00-00-00-02 2] connected
INFO:openflow.of_01:[00-00-00-00-00-01 3] connected
mininet@mininet-vm: ~/pox
login as: mininet
mininet@192.168.56.102's password:
Welcome to Ubuntu 14.04.4 LTS (GNU/Linux 4.2.0-27-generic x86_64)
* Documentation: https://help.ubuntu.com/
New release '16.04.5 LTS' available.
Run 'do-release-upgrade' to upgrade to it.
Last login: Thu Feb 21 18:10:34 2019 from 192.168.56.1
mininet@mininet-vm:~$ cd /home/mininet/pox
mininet@mininet-vm:~/pox$ ls
debug-pox.py LICENSE pox README tests
ext NOTICE pox.py setup.cfg tools
mininet@mininet-vm:~/pox$ .pox.py openflow.of_01 --port=10002 controller2
.pox.py: command not found
mininet@mininet-vm:~/pox$ ./pox.py openflow.of_01 --port=10002 controller2
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
INFO:core:POX 0.2.0 (carp) is up.
INFO:openflow.of_01:[00-00-00-00-00-04 1] connected
INFO:openflow.of_01:[00-00-00-00-00-03 2] connected
INFO:openflow.of_01:[00-00-00-00-00-01 3] connected
mininet@mininet-vm: ~/pox
login as: mininet
mininet@192.168.56.102's password:
Welcome to Ubuntu 14.04.4 LTS (GNU/Linux 4.2.0-27-generic x86_64)
* Documentation: https://help.ubuntu.com/
New release '16.04.5 LTS' available.
Run 'do-release-upgrade' to upgrade to it.
Last login: Thu Feb 21 18:08:40 2019 from 192.168.56.1
mininet@mininet-vm:~$ cd /home/mininet/pox/ext
mininet@mininet-vm:~/pox/ext$ ls
README skeleton.py
mininet@mininet-vm:~/pox/ext$ nano skeleton.py
mininet@mininet-vm:~/pox/ext$ nano controller1.py
mininet@mininet-vm:~/pox/ext$ nano controller2.py
mininet@mininet-vm:~/pox/ext$ cd /home/mininet/pox
mininet@mininet-vm:~/pox$ ./pox.py openflow.of_01 --port=10001 controller1
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
INFO:core:POX 0.2.0 (carp) is up.
INFO:openflow.of_01:[00-00-00-00-00-04 1] connected
INFO:openflow.of_01:[00-00-00-00-00-02 2] connected
INFO:openflow.of_01:[00-00-00-00-00-01 3] connected



We test connectivity between nodes h1 and h3; h2 and h2 using to confirm successful connection

h2 ping –c2 h4
h1 ping –c2 h3

And also verify that h2 cannot ping h1 and h3 and h1 cannot reach h2 and h4
h1 ping –c1 –W1 h2
h1 ping –c1 –W1 h4
h2 ping –c1 –W1 h1
h2 ping –c1 –W1 h3



Problem 10: Custom network behavior

Our chosen network behavior is to create a load balancer with a pox controller

The network will consist of a switch and 6 hosts. Two (2) of the hosts will serve as http servers and the rest as http clients. The switch will be controlled by a pox controller to implement a balance on how http traffic requests from the host clients will be directed to the 2 servers.

We used ***sudo mn –topo single,6 –controller=remote,port=6633*** command to set up the network topology

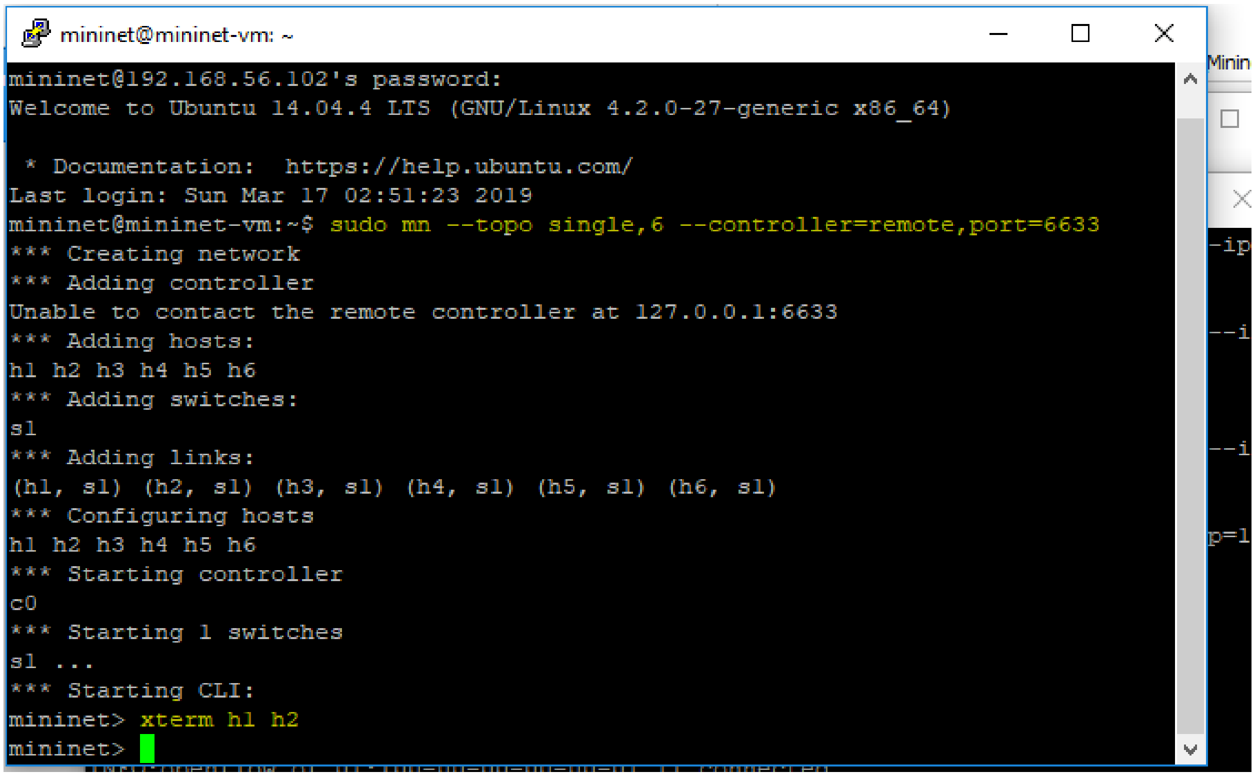


Fig: Building a simple topology in mininet

We set up h1 and h2 as http servers using the following commands

Xterm h1 h2

To open up terminals for h1 and h2

Python –m SimpleHTTPServer 80 &

To configure h1 and h2 as http servers

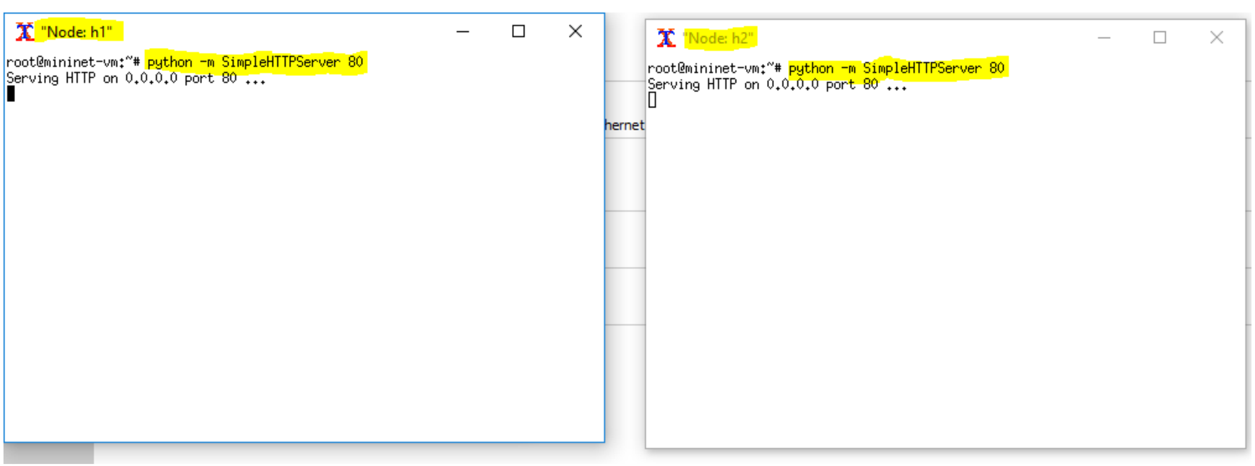


Fig: Configuring node h1 and h2 as http servers

We bring on the POX controller in a new terminal and configure it to operate as a load balancer by loading the ip_balancer script provided by pox with

cd /home/mininet/pox

./pox.py log.level –DEBUG misc.ip_loadbalancer –ip=10.0.1.1 –servers=10.0.0.1,10.0.0.2

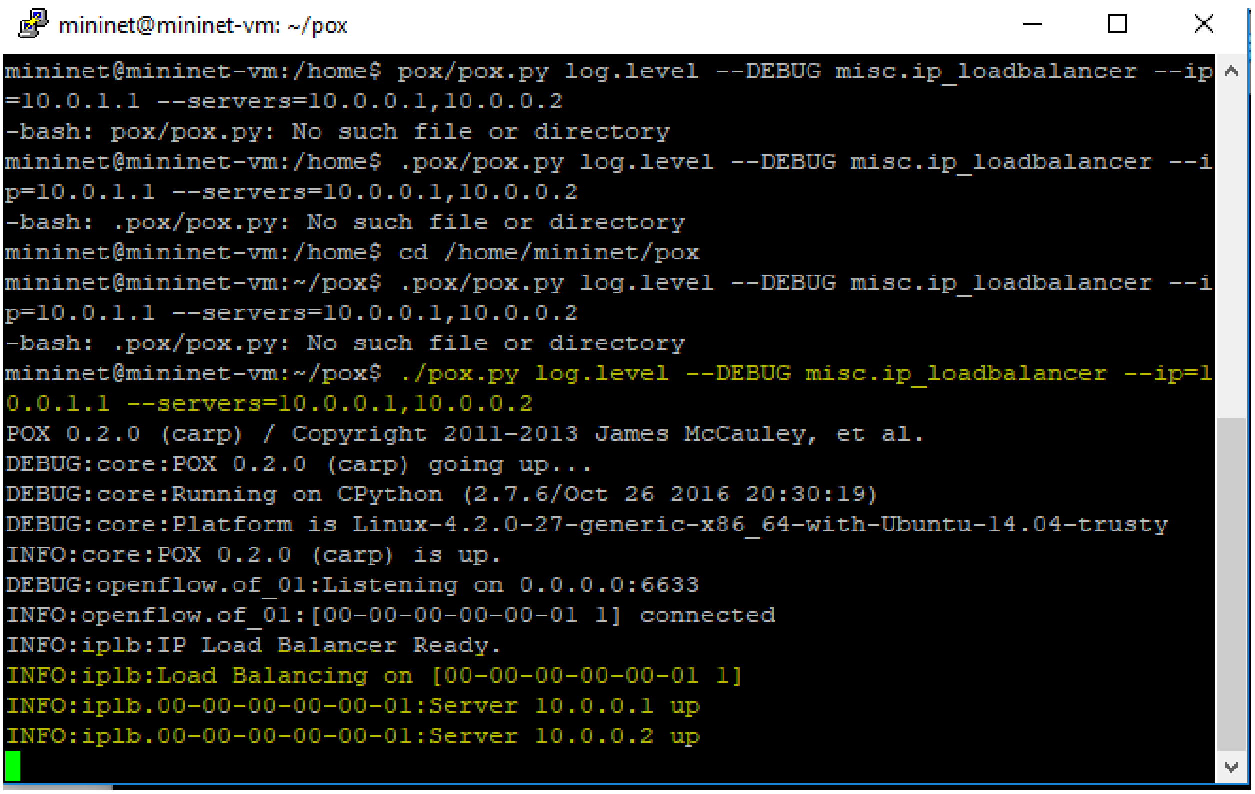


Fig: Bringing pox controller online for load balancing

Using **xterm h3 h4 h5 h6**

We open up terminals for http clients to make requests to the pox controller

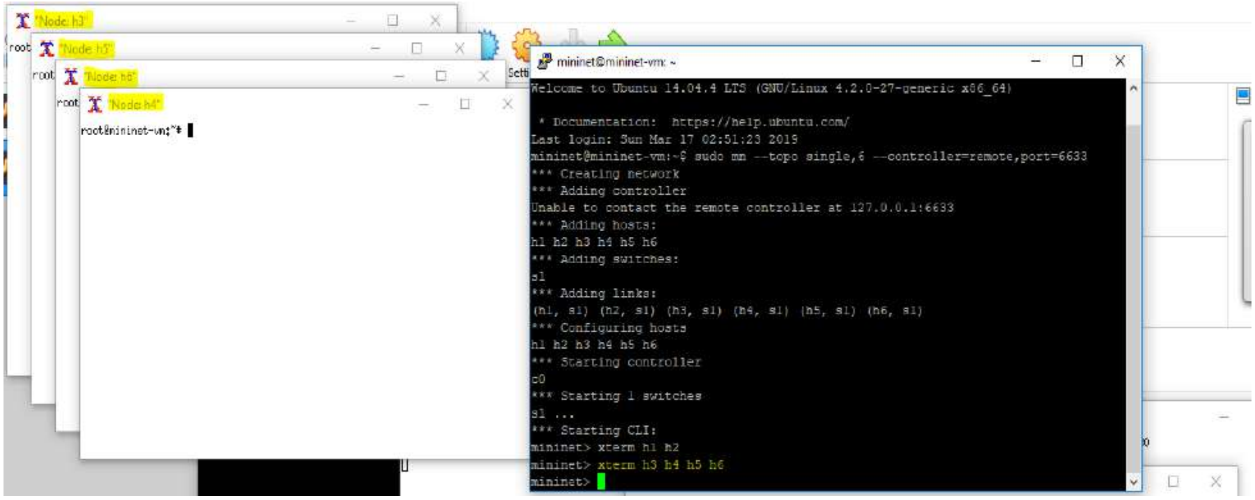


Fig: Opening up server client hosts

Make http requests from the servers through the controller and it directs the traffic to the 2 servers using a load balancing algorithm

h3 wget 10.0.1.1
h4 wget 10.0.1.1
h5 wget 10.0.1.1
h6 wget 10.0.1.1

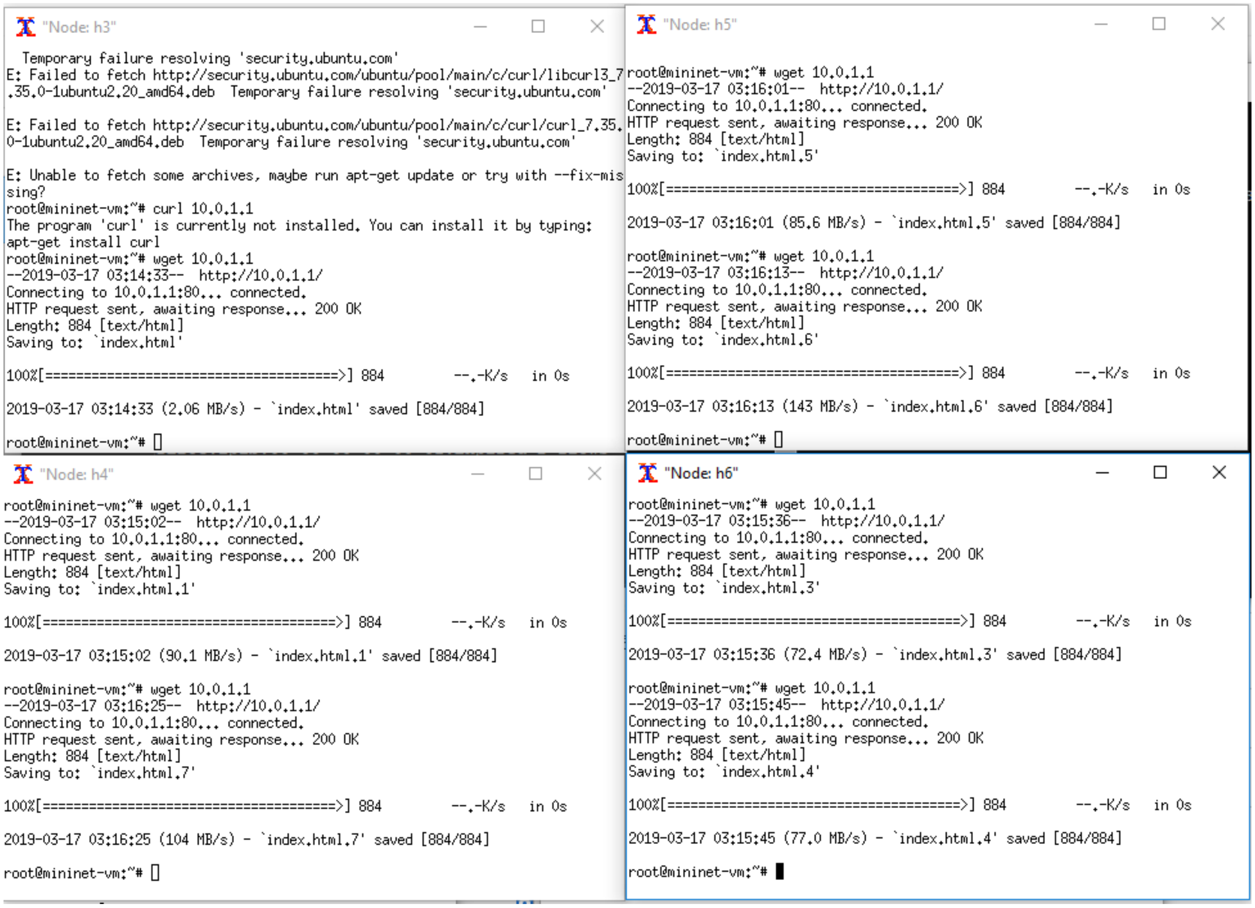


Fig: Screenshot showing multiple requests made by various nodes to the pox controller

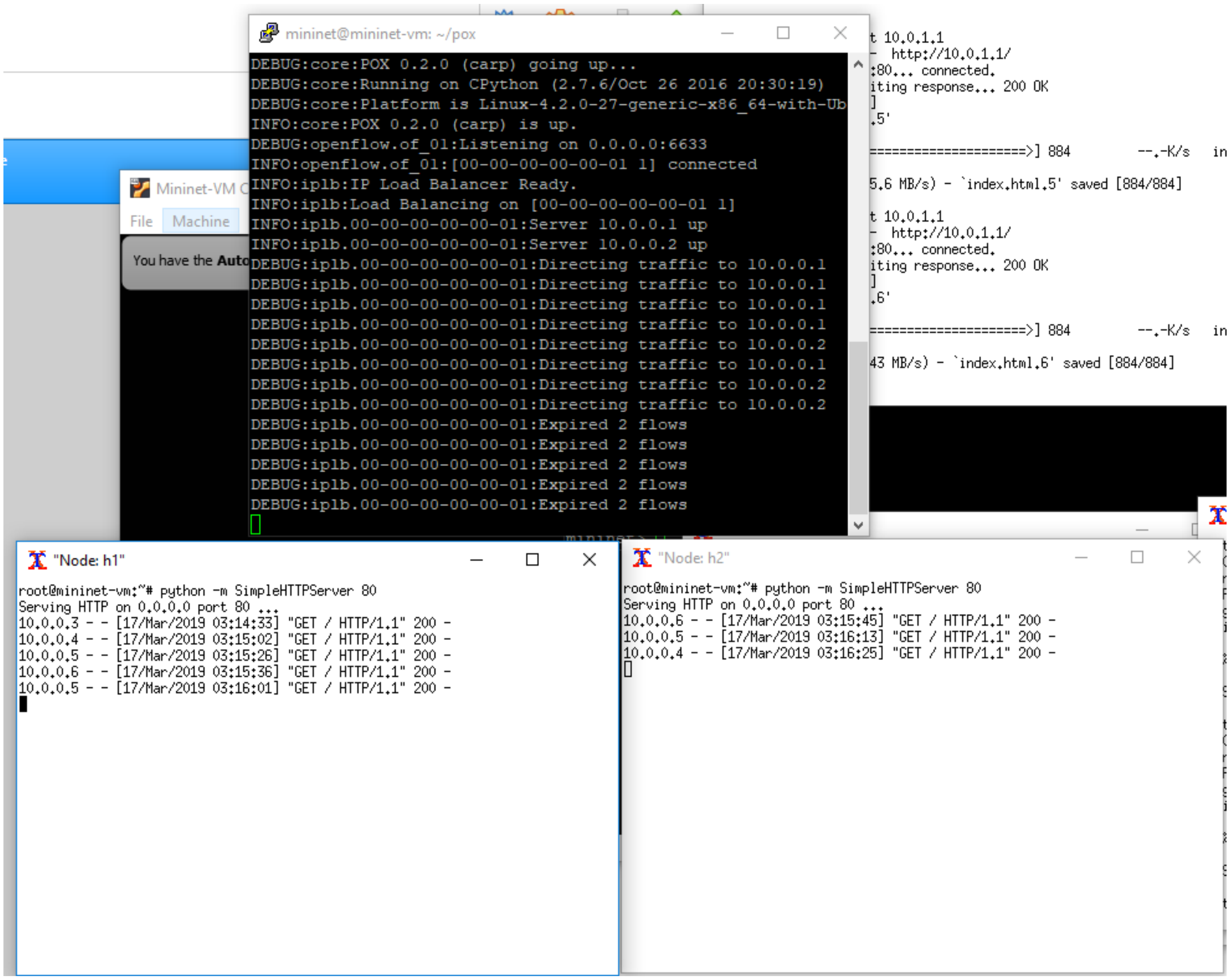


Fig: Screenshot of how the http request load is balanced between the 2 servers by the pox controller

APPENDIX A

Python script for ip load balancing for pox controller

```
from pox.core import core
import pox
log = core.getLogger("iplb")

from pox.lib.packet.ethernet import ethernet, ETHER_BROADCAST
from pox.lib.packet.ipv4 import ipv4
from pox.lib.packet.arp import arp
from pox.lib.addresses import IPAddr, EthAddr
from pox.lib.util import str_to_bool, dpid_to_str, str_to_dpid
```

```
import pox.openflow.libopenflow_01 as of

import time
import random

FLOW_IDLE_TIMEOUT = 10
FLOW_MEMORY_TIMEOUT = 60 * 5



class MemoryEntry (object):

  def __init__ (self, server, first_packet, client_port):
    self.server = server
    self.first_packet = first_packet
    self.client_port = client_port
    self.refresh()

  def refresh (self):
    self.timeout = time.time() + FLOW_MEMORY_TIMEOUT

  @property
  def is_expired (self):
    return time.time() > self.timeout

  @property
  def key1 (self):
    ethp = self.first_packet
    ipp = ethp.find('ipv4')
    tcpp = ethp.find('tcp')

    return ipp.srcip,ipp.dstip,tcpp.srcport,tcpp.dstport

  @property
  def key2 (self):
    ethp = self.first_packet
    ipp = ethp.find('ipv4')
    tcpp = ethp.find('tcp')

    return self.server,ipp.srcip,tcpp.dstport,tcpp.srcport


class iplb (object):

  def __init__ (self, connection, service_ip, servers = []):
    self.service_ip = IPAddr(service_ip)
    self.servers = [IPAddr(a) for a in servers]
    self.con = connection
    self.mac = self.con.eth_addr
    self.live_servers = {} # IP -> MAC,port
```

```
  try:
    self.log = log.getChild(dpid_to_str(self.con.dpid))
  except:
    # Be nice to Python 2.6 (ugh)
    self.log = log

  self.outstanding_probes = {} # IP -> expire_time

  # How quickly do we probe?
  self.probe_cycle_time = 5

  # How long do we wait for an ARP reply before we consider a server dead?
  self.arp_timeout = 3

  self.memory = {} # (srcip,dstip,srcport,dstport) -> MemoryEntry

  self._do_probe() # Kick off the probing

def _do_expire (self):

  t = time.time()

  for ip,expire_at in self.outstanding_probes.items():
    if t > expire_at:
      self.outstanding_probes.pop(ip, None)
      if ip in self.live_servers:
        self.log.warn("Server %s down", ip)
        del self.live_servers[ip]

  # Expire old flows
  c = len(self.memory)
  self.memory = {k:v for k,v in self.memory.items()
                 if not v.is_expired}
  if len(self.memory) != c:
    self.log.debug("Expired %i flows", c-len(self.memory))

def _do_probe (self):
  self._do_expire()

  server = self.servers.pop(0)
  self.servers.append(server)

  r = arp()
  r.hwtype = r.HW_TYPE_ETHERNET
  r.prototype = r.PROTO_TYPE_IP
  r.opcode = r.REQUEST
  r.hwdst = ETHER_BROADCAST
  r.protodst = server
  r.hwsrc = self.mac
  r.protosrc = self.service_ip
  e = ethernet(type=ethernet.ARP_TYPE, src=self.mac,
```

```
            dst=ETHER_BROADCAST)
  e.set_payload(r)
  #self.log.debug("ARPing for %s", server)
  msg = of.ofp_packet_out()
  msg.data = e.pack()
  msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
  msg.in_port = of.OFPP_NONE
  self.con.send(msg)

  self.outstanding_probes[server] = time.time() + self.arp_timeout

  core.callDelayed(self._probe_wait_time, self._do_probe)

@property
def _probe_wait_time (self):
  """
  Time to wait between probes
  """
  r = self.probe_cycle_time / float(len(self.servers))
  r = max(.25, r) # Cap it at four per second
  return r

def _pick_server (self, key, inport):
  """
  Pick a server for a (hopefully) new connection
  """
  return random.choice(self.live_servers.keys())

def _handle_PacketIn (self, event):
  inport = event.port
  packet = event.parsed

  def drop ():
    if event.ofp.buffer_id is not None:
      # Kill the buffer
      msg = of.ofp_packet_out(data = event.ofp)
      self.con.send(msg)
    return None

  tcpp = packet.find('tcp')
  if not tcpp:
    arpp = packet.find('arp')
    if arpp:
      # Handle replies to our server-liveness probes
      if arpp.opcode == arpp.REPLY:
        if arpp.protosrc in self.outstanding_probes:
          # A server is (still?) up; cool.
          del self.outstanding_probes[arpp.protosrc]
          if (self.live_servers.get(arpp.protosrc, (None,None))
              == (arpp.hwsrc,inport)):
            # Ah, nothing new here.
```

```
        pass
      else:
        # Ooh, new server.
        self.live_servers[arpp.protosrc] = arpp.hwsrc,inport
        self.log.info("Server %s up", arpp.protosrc)
    return

  # Not TCP and not ARP.  Don't know what to do with this.  Drop it.
  return drop()

# It's TCP.

ipp = packet.find('ipv4')

if ipp.srcip in self.servers:
  # It's FROM one of our balanced servers.
  # Rewrite it BACK to the client

  key = ipp.srcip,ipp.dstip,tcpp.srcport,tcpp.dstport
  entry = self.memory.get(key)

  if entry is None:
    # We either didn't install it, or we forgot about it.
    self.log.debug("No client for %s", key)
    return drop()

  # Refresh time timeout and reinstall.
  entry.refresh()

  #self.log.debug("Install reverse flow for %s", key)

  # Install reverse table entry
  mac,port = self.live_servers[entry.server]

  actions = []
  actions.append(of.ofp_action_dl_addr.set_src(self.mac))
  actions.append(of.ofp_action_nw_addr.set_src(self.service_ip))
  actions.append(of.ofp_action_output(port = entry.client_port))
  match = of.ofp_match.from_packet(packet, inport)

  msg = of.ofp_flow_mod(command=of.OFPFC_ADD,
                        idle_timeout=FLOW_IDLE_TIMEOUT,
                        hard_timeout=of.OFP_FLOW_PERMANENT,
                        data=event.ofp,
                        actions=actions,
                        match=match)
  self.con.send(msg)

elif ipp.dstip == self.service_ip:
  # Ah, it's for our service IP and needs to be load balanced
```

```
    # Do we already know this flow?
    key = ipp.srcip,ipp.dstip,tcpp.srcport,tcpp.dstport
    entry = self.memory.get(key)
    if entry is None or entry.server not in self.live_servers:
      # Don't know it (hopefully it's new!)
      if len(self.live_servers) == 0:
        self.log.warn("No servers!")
        return drop()

      # Pick a server for this flow
      server = self._pick_server(key, inport)
      self.log.debug("Directing traffic to %s", server)
      entry = MemoryEntry(server, packet, inport)
      self.memory[entry.key1] = entry
      self.memory[entry.key2] = entry

    # Update timestamp
    entry.refresh()

    # Set up table entry towards selected server
    mac,port = self.live_servers[entry.server]

    actions = []
    actions.append(of.ofp_action_dl_addr.set_dst(mac))
    actions.append(of.ofp_action_nw_addr.set_dst(entry.server))
    actions.append(of.ofp_action_output(port = port))
    match = of.ofp_match.from_packet(packet, inport)

    msg = of.ofp_flow_mod(command=of.OFPFC_ADD,
                          idle_timeout=FLOW_IDLE_TIMEOUT,
                          hard_timeout=of.OFP_FLOW_PERMANENT,
                          data=event.ofp,
                          actions=actions,
                          match=match)
    self.con.send(msg)


# Remember which DPID we're operating on (first one to connect)
_dpid = None


def launch (ip, servers, dpid = None):
  global _dpid
  if dpid is not None:
    _dpid = str_to_dpid(dpid)

  servers = servers.replace(","," ").split()
  servers = [IPAddr(x) for x in servers]
  ip = IPAddr(ip)
```

```
# We only want to enable ARP Responder *only* on the load balancer switch,
# so we do some disgusting hackery and then boot it up.
from proto.arp_responder import ARPResponder
old_pi = ARPResponder._handle_PacketIn
def new_pi (self, event):
  if event.dpid == _dpid:
    # Yes, the packet-in is on the right switch
    return old_pi(self, event)
ARPResponder._handle_PacketIn = new_pi

# Hackery done.  Now start it.
from proto.arp_responder import launch as arp_launch
arp_launch(eat_packets=False,**{str(ip):True})
import logging
logging.getLogger("proto.arp_responder").setLevel(logging.WARN)


def _handle_ConnectionUp (event):
  global _dpid
  if _dpid is None:
    _dpid = event.dpid

  if _dpid != event.dpid:
    log.warn("Ignoring switch %s", event.connection)
  else:
    if not core.hasComponent('iplb'):
      # Need to initialize first...
      core.registerNew(iplb, event.connection, IPAddr(ip), servers)
      log.info("IP Load Balancer Ready.")
    log.info("Load Balancing on %s", event.connection)

    # Gross hack
    core.iplb.con = event.connection
    event.connection.addListeners(core.iplb)


core.openflow.addListenerByName("ConnectionUp", _handle_ConnectionUp)
```