LAB 1

EXPERIMENTING WITH OPENFLOW IN MININET

AKHILA NAIR

01743358

Turn in Date: 2/12/18

Estimated Hours of Working: 7 Hours.

1) Purpose:

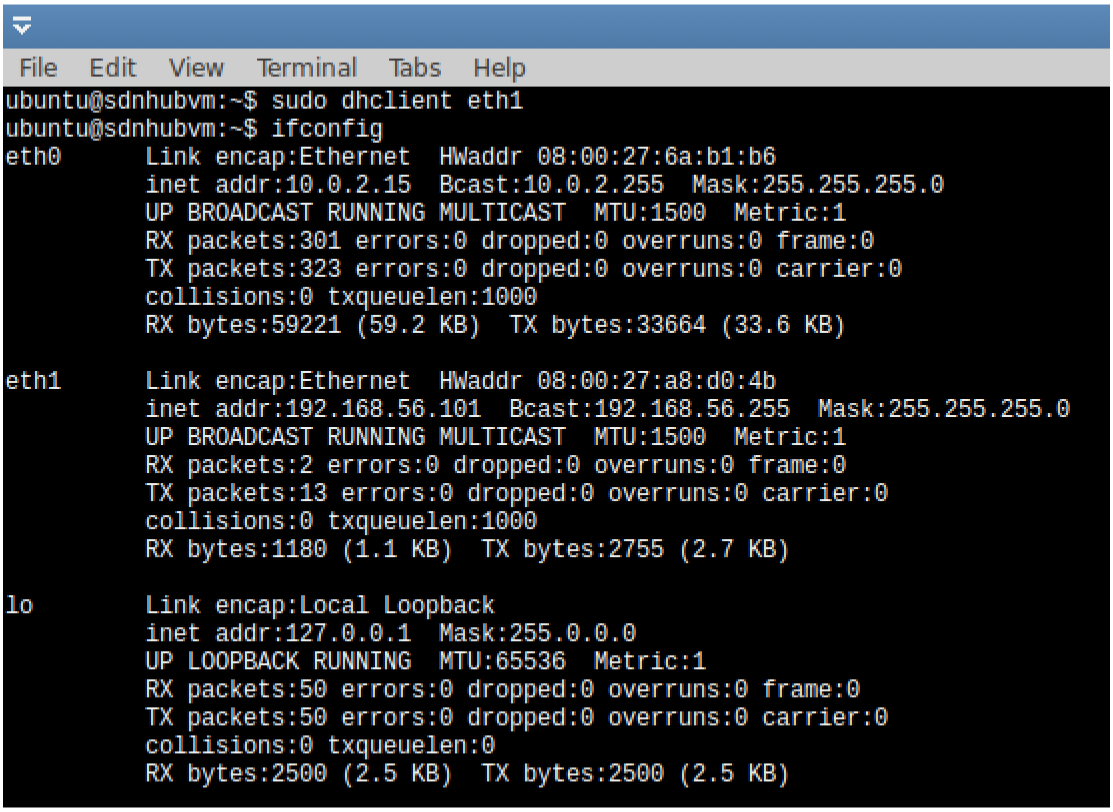
- Installing and configuring virtual box an SDN hub.
- Executing a packet flow network that can be traced graphically.
- Creating a virtual network with the switch, controller and hosts.
- Understanding basic mininet commands and its functions.
- Understanding Openflow protocol and their communication between switch and controller.
- Experimenting with Wireshark and analysing Loopback interfaces.

2) Setup:

Lab 1 procedure has following steps:

2.1: Setting up Virtual Machine –

- Open Virtual Box and import SDN Hub.
- IP Address of virtual machine can be found with commands – sudo dhclient eth1 and ifconfig.



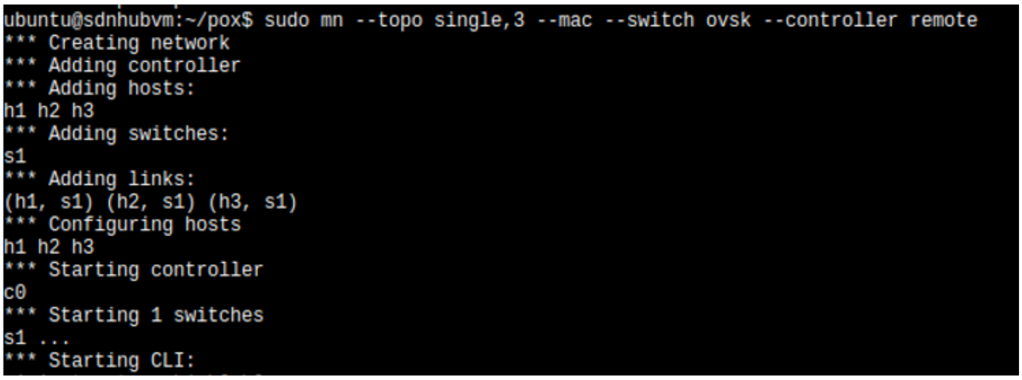
```
ubuntu@sdnhubvm:~$ sudo dhclient eth1
ubuntu@sdnhubvm:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 08:00:27:6a:b1:b6
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:301 errors:0 dropped:0 overruns:0 frame:0
          TX packets:323 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:59221 (59.2 KB)  TX bytes:33664 (33.6 KB)

eth1      Link encap:Ethernet  HWaddr 08:00:27:a8:d0:4b
          inet addr:192.168.56.101  Bcast:192.168.56.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2 errors:0 dropped:0 overruns:0 frame:0
          TX packets:13 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1180 (1.1 KB)  TX bytes:2755 (2.7 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:50 errors:0 dropped:0 overruns:0 frame:0
          TX packets:50 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:2500 (2.5 KB)  TX bytes:2500 (2.5 KB)
```

- Now, SSH into the virtual machine with the IP address of eth1 i.e 192.168..56.101
- This will now as for username and password, which in this case is Ubuntu in both the case.

- After successfully completing above steps you should now be able to work with the virtual machine.

2.2: Creating a mininet network -

- A network containing 3 hosts, 1 switch and 1 remote controller can be obtained by following command.
  Sudo mn –topo single,3 -- mac – switch ovsk – controller remote



```
ubuntu@sdnhubvm:~/pox$ sudo mn --topo single,3 --mac --switch ovsk --controller remote
*** Creating network
*** Adding controller
*** 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:
```

2.3: Wireshark packet capture and xterm -

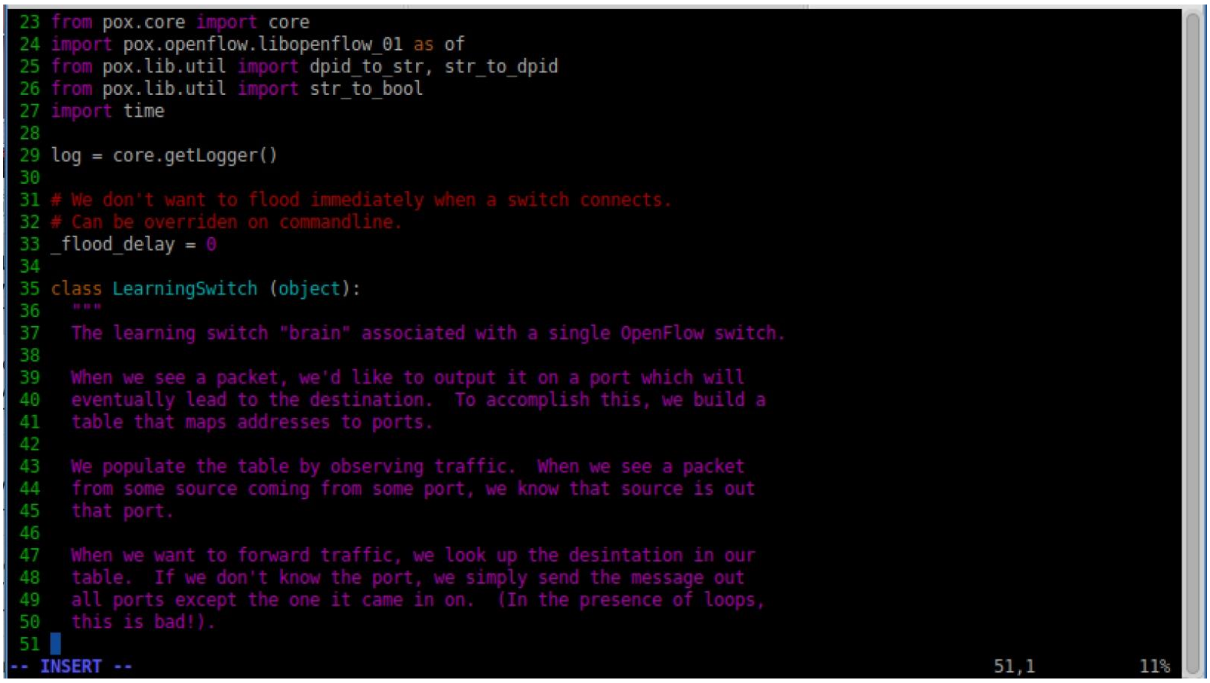
- Open a new terminal and do the following to open wireshark packet capture.
  Sudo wireshark &
- Go to capture interface and select lo i.e loopback interface from the list.
- To open nodes h1 h2 h3 and to have graphical view of packet from on terminal put xterm h1 h2 h3 command.
- To see the traffic of packets flowing enter this command
  tcpdump –xx –n –I h-eth0
- To start the controller
  Cd pox
  ./pox.py log.level –DEBUG misc.of_tutorial
- For pinging between host and controller
  h1 ping –c1 h2

3) Software Design:

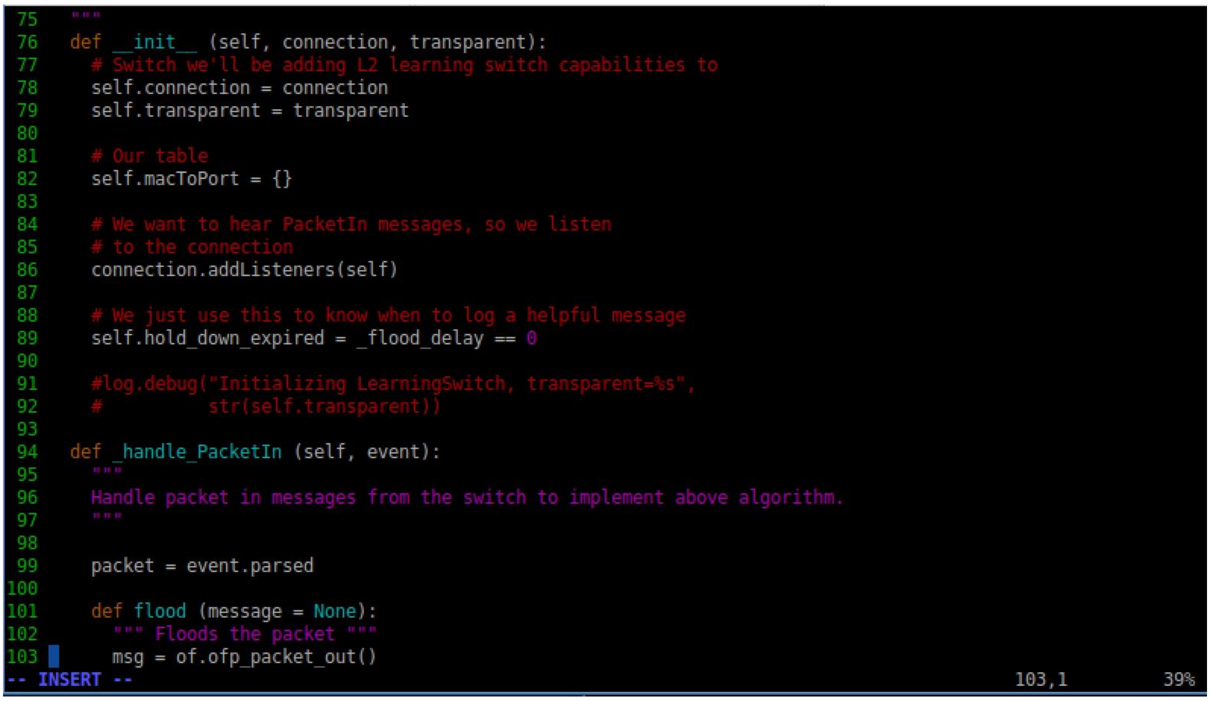
Basic commands which are used throughout the lab 1 is explained in the following section.

- **Sudo mn –** This includes one OpenFlow kernel switch connected to 2 hosts, plus the OpenFlow reference controller.
- **Sudo wireshark & -** To view traffic using the OpenFlow Dissector.
- **H1 ping –c1 h2 –** This will execute the ping command from host 0 to host 1 via ICMP echo request.
- **Xterm h1 h2 h3 –** This command will open all the nodes where graphical representation of packet flow can be seen.
- **Ifconfig –** With provide you with host address and Ip address.

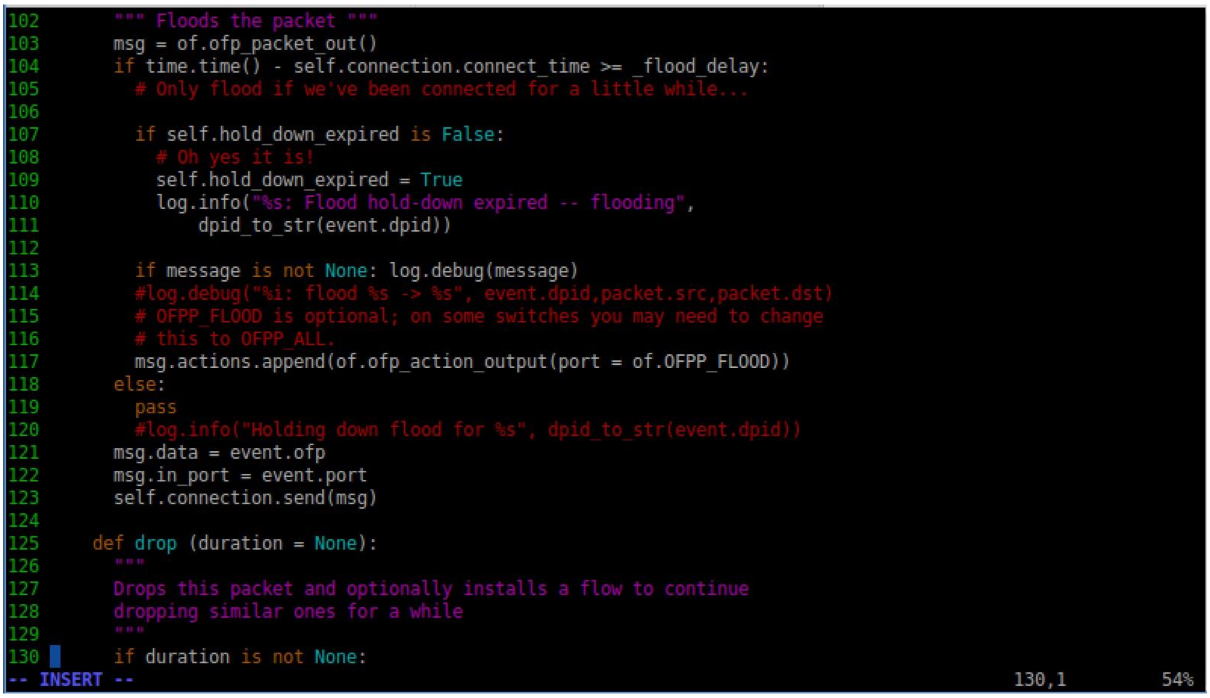
Learning Switch Source Code:



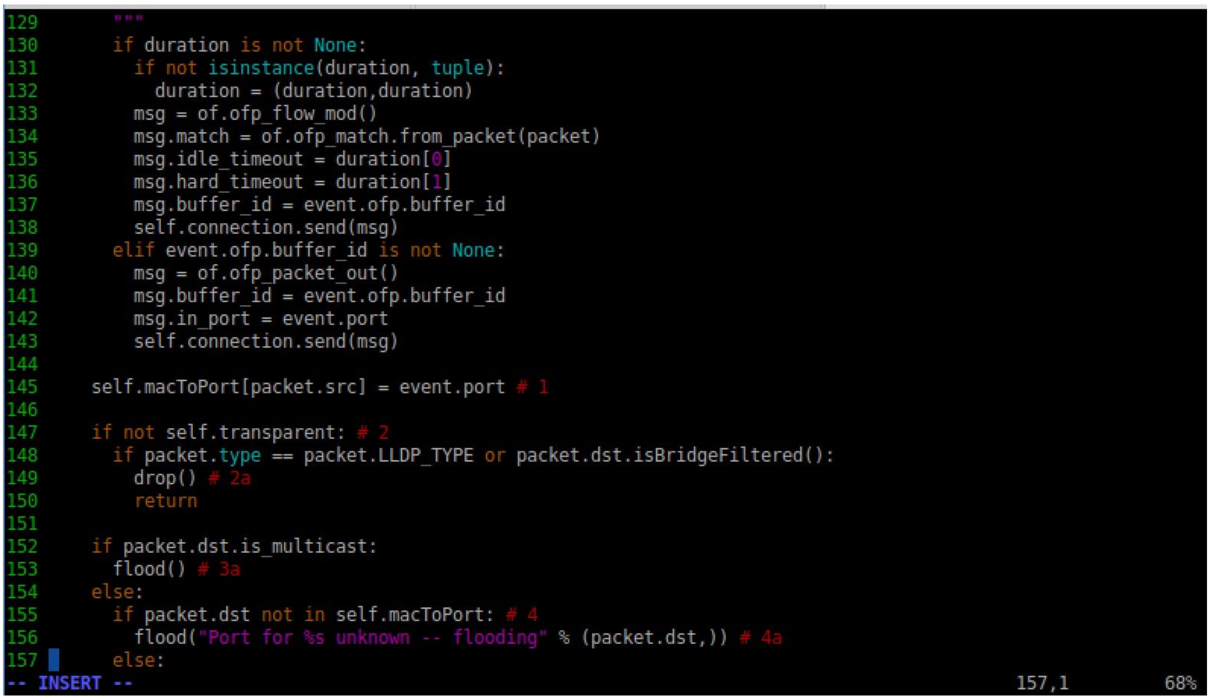
```
from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.util import dpid_to_str, str_to_dpid
from pox.lib.util import str_to_bool
import time

log = core.getLogger()

# We don't want to flood immediately when a switch connects.
# Can be overriden on commandline.
_flood_delay = 0

class LearningSwitch (object):
  """
  The learning switch "brain" associated with a single OpenFlow switch.

  When we see a packet, we'd like to output it on a port which will
  eventually lead to the destination.  To accomplish this, we build a
  table that maps addresses to ports.

  We populate the table by observing traffic.  When we see a packet
  from some source coming from some port, we know that source is out
  that port.

  When we want to forward traffic, we look up the desintation in our
  table.  If we don't know the port, we simply send the message out
  all ports except the one it came in on.  (In the presence of loops,
  this is bad!).

```



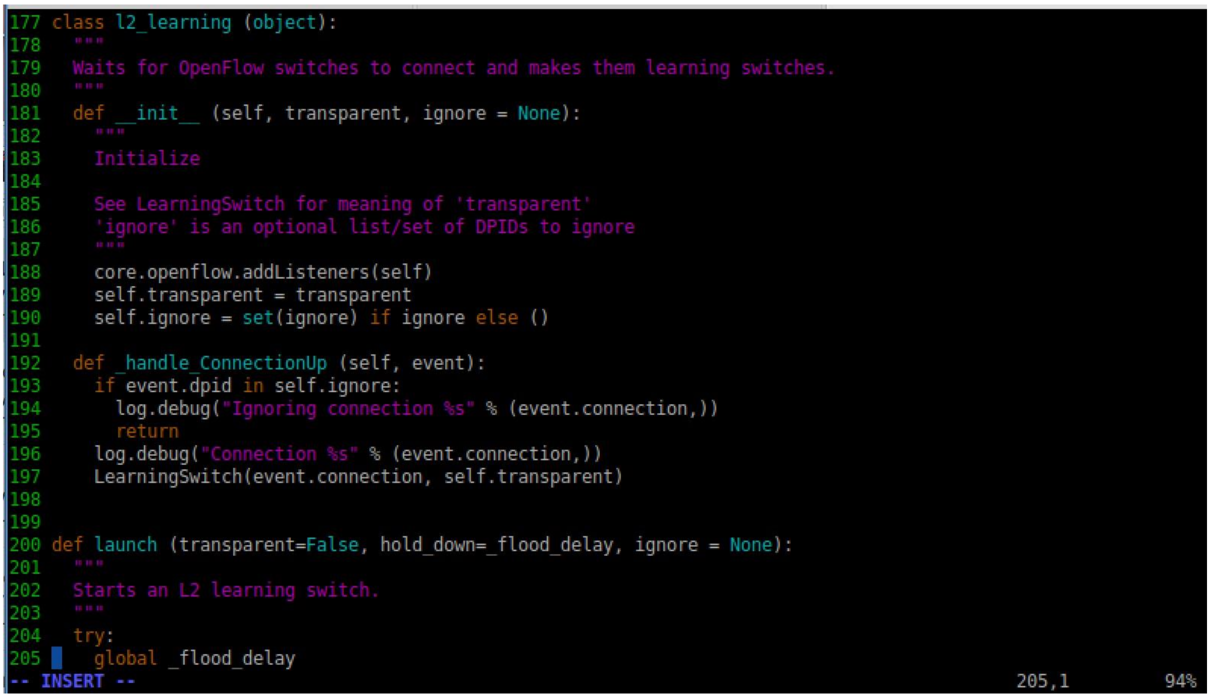
```
  """
  def __init__ (self, connection, transparent):
    # Switch we'll be adding L2 learning switch capabilities to
    self.connection = connection
    self.transparent = transparent

    # Our table
    self.macToPort = {}

    # We want to hear PacketIn messages, so we listen
    # to the connection
    connection.addListeners(self)

    # We just use this to know when to log a helpful message
    self.hold_down_expired = _flood_delay == 0

    #log.debug("Initializing LearningSwitch, transparent=%s",
    #          str(self.transparent))

  def _handle_PacketIn (self, event):
    """
    Handle packet in messages from the switch to implement above algorithm.
    """

    packet = event.parsed

    def flood (message = None):
      """ Floods the packet """
      msg = of.ofp_packet_out()
-- INSERT --                                                          103,1          39%
```



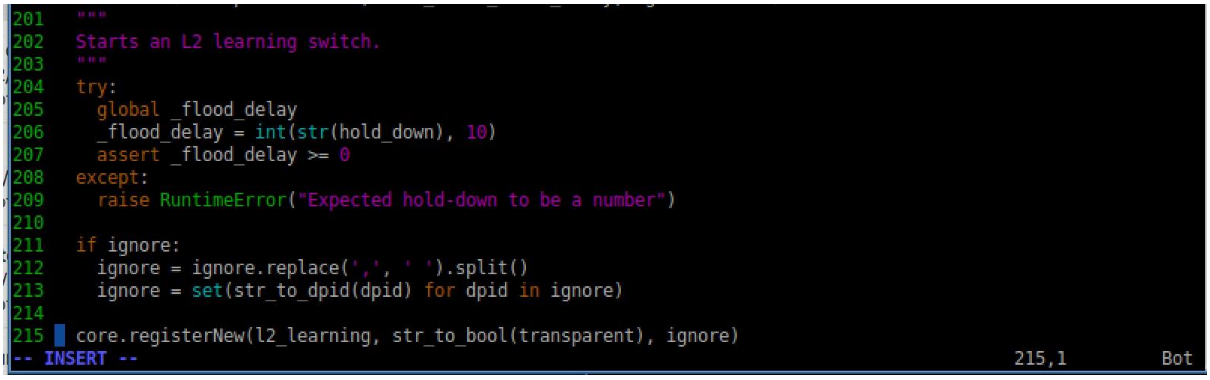
```
      """ Floods the packet """
      msg = of.ofp_packet_out()
      if time.time() - self.connection.connect_time >= _flood_delay:
        # Only flood if we've been connected for a little while...

        if self.hold_down_expired is False:
          # Oh yes it is!
          self.hold_down_expired = True
          log.info("%s: Flood hold-down expired -- flooding",
              dpid_to_str(event.dpid))

        if message is not None: log.debug(message)
        #log.debug("%i: flood %s -> %s", event.dpid,packet.src,packet.dst)
        # OFPP_FLOOD is optional; on some switches you may need to change
        # this to OFPP_ALL.
        msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
      else:
        pass
        #log.info("Holding down flood for %s", dpid_to_str(event.dpid))
      msg.data = event.ofp
      msg.in_port = event.port
      self.connection.send(msg)

    def drop (duration = None):
      """
      Drops this packet and optionally installs a flow to continue
      dropping similar ones for a while
      """
      if duration is not None:
-- INSERT --                                                          130,1          54%
```



```
      """
      if duration is not None:
        if not isinstance(duration, tuple):
          duration = (duration,duration)
        msg = of.ofp_flow_mod()
        msg.match = of.ofp_match.from_packet(packet)
        msg.idle_timeout = duration[0]
        msg.hard_timeout = duration[1]
        msg.buffer_id = event.ofp.buffer_id
        self.connection.send(msg)
      elif event.ofp.buffer_id is not None:
        msg = of.ofp_packet_out()
        msg.buffer_id = event.ofp.buffer_id
        msg.in_port = event.port
        self.connection.send(msg)

    self.macToPort[packet.src] = event.port # 1

    if not self.transparent: # 2
      if packet.type == packet.LLDP_TYPE or packet.dst.isBridgeFiltered():
        drop() # 2a
        return

    if packet.dst.is_multicast:
      flood() # 3a
    else:
      if packet.dst not in self.macToPort: # 4
        flood("Port for %s unknown -- flooding" % (packet.dst,)) # 4a
      else:
-- INSERT --                                                  157,1         68%
```



```
class l2_learning (object):
  """
  Waits for OpenFlow switches to connect and makes them learning switches.
  """
  def __init__ (self, transparent, ignore = None):
    """
    Initialize

    See LearningSwitch for meaning of 'transparent'
    'ignore' is an optional list/set of DPIDs to ignore
    """
    core.openflow.addListeners(self)
    self.transparent = transparent
    self.ignore = set(ignore) if ignore else ()

  def _handle_ConnectionUp (self, event):
    if event.dpid in self.ignore:
      log.debug("Ignoring connection %s" % (event.connection,))
      return
    log.debug("Connection %s" % (event.connection,))
    LearningSwitch(event.connection, self.transparent)


def launch (transparent=False, hold_down=_flood_delay, ignore = None):
  """
  Starts an L2 learning switch.
  """
  try:
    global _flood_delay
-- INSERT --                                                  205,1         94%
```

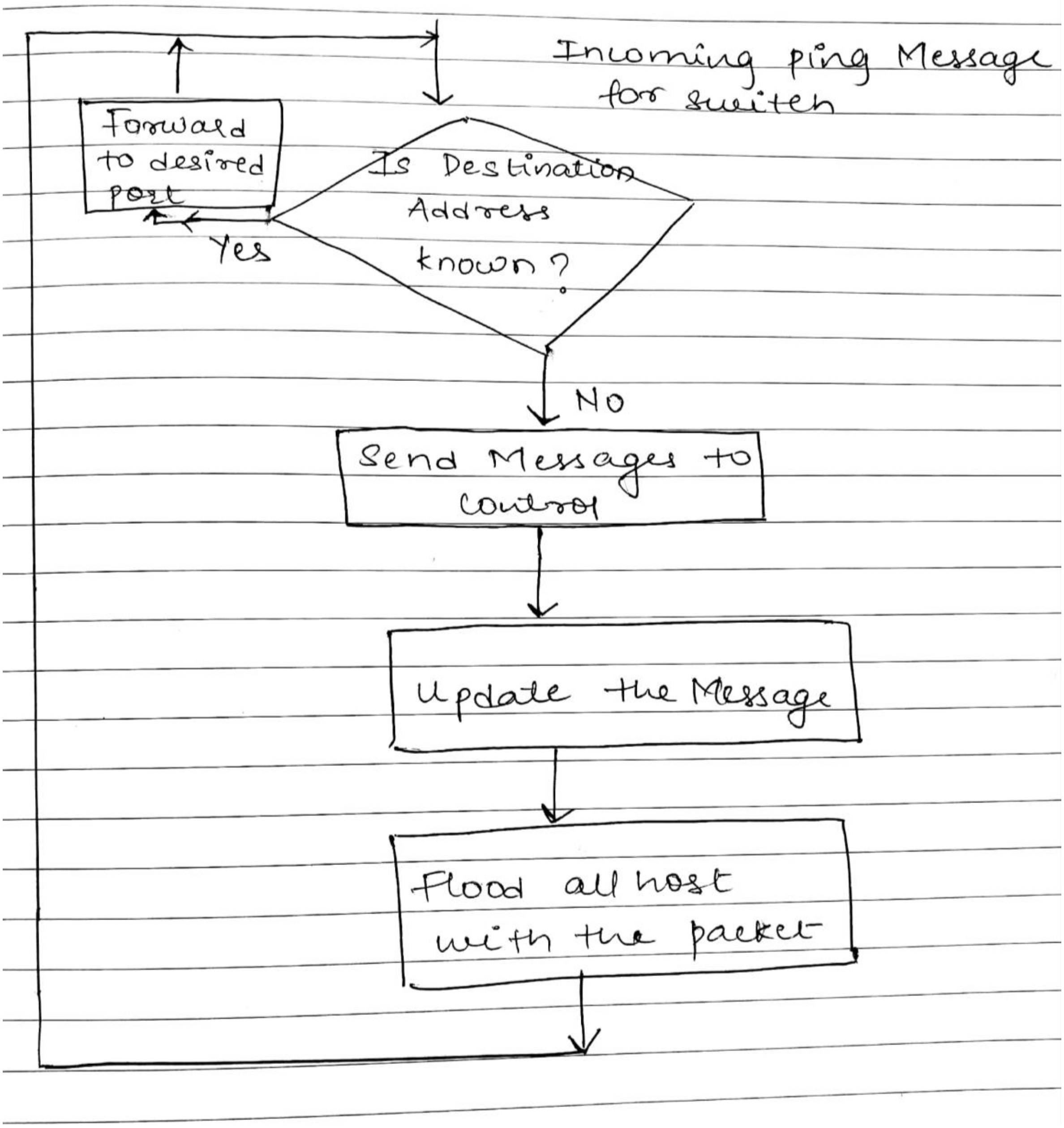


```
  """
  Starts an L2 learning switch.
  """
  try:
    global _flood_delay
    _flood_delay = int(str(hold_down), 10)
    assert _flood_delay >= 0
  except:
    raise RuntimeError("Expected hold-down to be a number")

  if ignore:
    ignore = ignore.replace(',', ' ').split()
    ignore = set(str_to_dpid(dpid) for dpid in ignore)

  core.registerNew(l2_learning, str_to_bool(transparent), ignore)
-- INSERT --                                                  215,1         Bot
```

Flowchart:

Following commands are assumed executed:

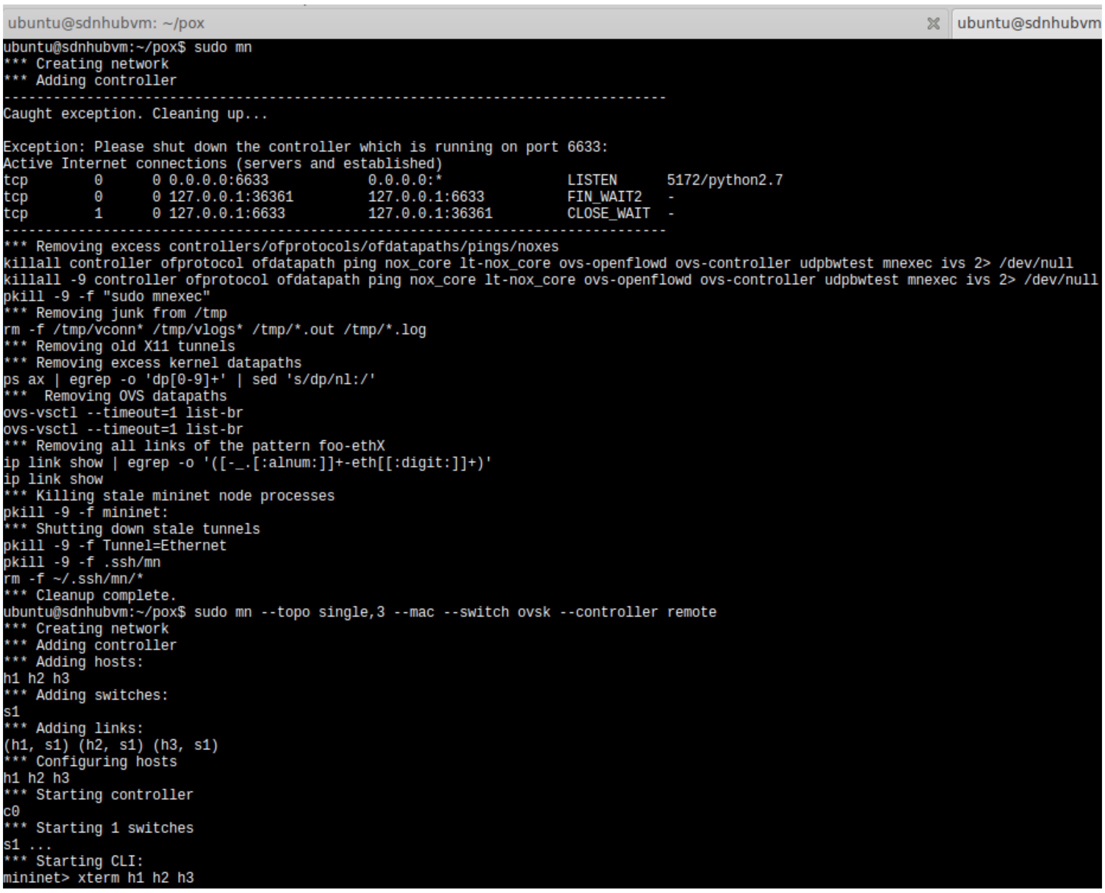
- Virtual Machine is running.
- Mininet Topology is created.
- Wireshark is configured.
- Nodes are opened and the ping command is given.
- Pox controller is initiated.


Incoming ping Message for switch
Forward to desired port
Is Destination Address known?
Yes
No
Send Messages to control
Update the Message
Flood all host with the packet

4) Trouble shooting:

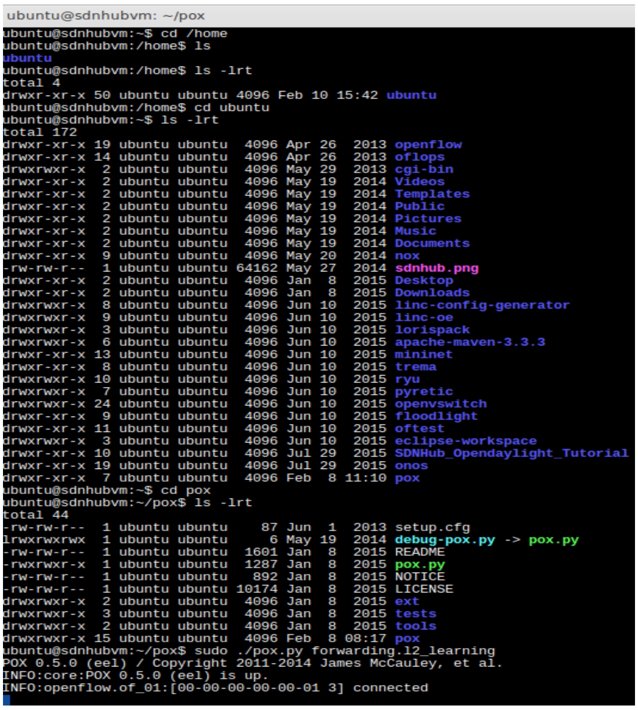
- Understanding the concept of openflow and wireshark – The concepts and software were new so at the beginning it was difficult to understand why particular command worked in this way. To get used to the software I went through different videos and papers and understood the basic concept of wireshark, its use in packet flow and different commands that can be given to achieve the intended results.
- Connecting to the host controller – I was getting a message that controler is not connected to the system. This was due to SSH. I realised that my system name is sdnhubvm and not mininet. Difficult to SSH into the system was solved by first finding the IP address of eth1 and then SSH it using that IP address.
- Finding the IP Address of the eth1 – This was solved using sudo dhclient eth1 and ifconfig.

5) Results:



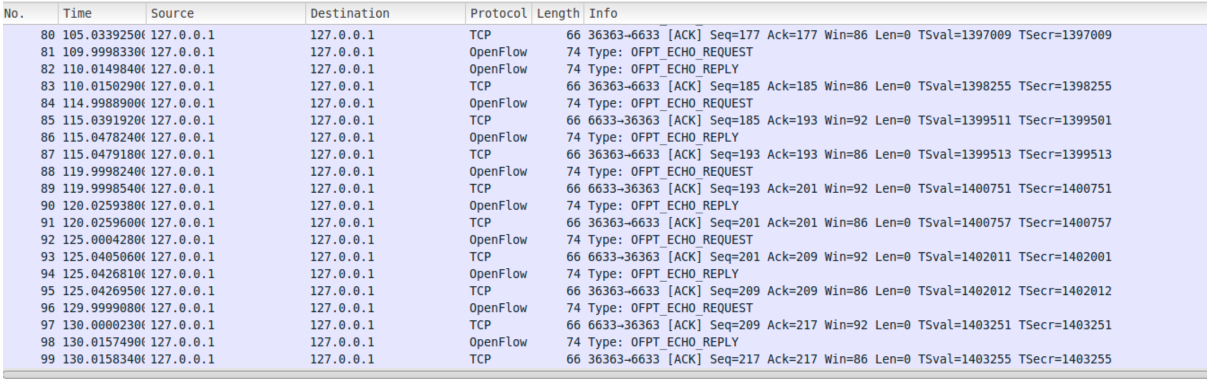
```
ubuntu@sdnhubvm:~/pox$ sudo mn
*** Creating network
*** Adding controller
--------------------------------------------------------------------------------
Caught exception. Cleaning up...

Exception: Please shut down the controller which is running on port 6633:
Active Internet connections (servers and established)
tcp        0      0 0.0.0.0:6633            0.0.0.0:*               LISTEN      5172/python2.7
tcp        0      0 127.0.0.1:36361         127.0.0.1:6633          FIN_WAIT2   -
tcp        1      0 127.0.0.1:6633          127.0.0.1:36361         CLOSE_WAIT  -
--------------------------------------------------------------------------------
*** 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 --timeout=1 list-br
*** Removing all links of the pattern foo-ethX
ip link show | egrep -o '([-_.[:alnum:]]+-eth[[:digit:]]+)'
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.
ubuntu@sdnhubvm:~/pox$ sudo mn --topo single,3 --mac --switch ovsk --controller remote
*** Creating network
*** Adding controller
*** 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> xterm h1 h2 h3
```

- The above image shows how the system makes a network comprising of host and controller and initiating links between them.



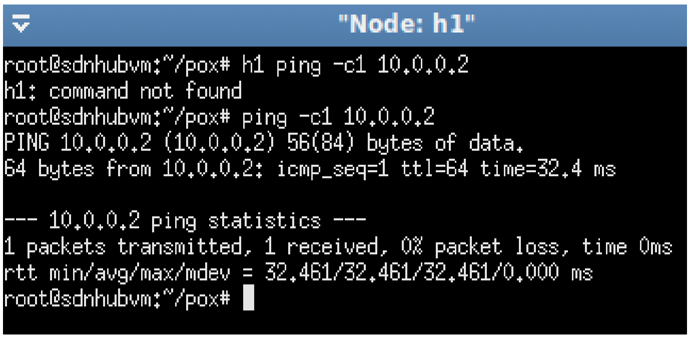
```
ubuntu@sdnhubvm: ~/pox
ubuntu@sdnhubvm:~$ cd /home
ubuntu@sdnhubvm:/home$ ls
ubuntu
ubuntu@sdnhubvm:/home$ ls -lrt
total 4
drwxr-xr-x 50 ubuntu ubuntu 4096 Feb 10 15:42 ubuntu
ubuntu@sdnhubvm:/home$ cd ubuntu
ubuntu@sdnhubvm:~$ ls -lrt
total 172
drwxr-xr-x 19 ubuntu ubuntu  4096 Apr 26  2013 openflow
drwxr-xr-x 14 ubuntu ubuntu  4096 Apr 26  2013 oflops
drwxrwxr-x  2 ubuntu ubuntu  4096 May 29  2013 cgi-bin
drwxr-xr-x  2 ubuntu ubuntu  4096 May 19  2014 Videos
drwxr-xr-x  2 ubuntu ubuntu  4096 May 19  2014 Templates
drwxr-xr-x  2 ubuntu ubuntu  4096 May 19  2014 Public
drwxr-xr-x  2 ubuntu ubuntu  4096 May 19  2014 Pictures
drwxr-xr-x  2 ubuntu ubuntu  4096 May 19  2014 Music
drwxr-xr-x  2 ubuntu ubuntu  4096 May 19  2014 Documents
drwxr-xr-x  9 ubuntu ubuntu  4096 May 20  2014 nox
-rw-rw-r--  1 ubuntu ubuntu 64162 May 27  2014 sdnhub.png
drwxr-xr-x  2 ubuntu ubuntu  4096 Jan  8  2015 Desktop
drwxr-xr-x  2 ubuntu ubuntu  4096 Jan  8  2015 Downloads
drwxrwxr-x  8 ubuntu ubuntu  4096 Jun 10  2015 linc-config-generator
drwxrwxr-x  9 ubuntu ubuntu  4096 Jun 10  2015 linc-oe
drwxrwxr-x  3 ubuntu ubuntu  4096 Jun 10  2015 lorispack
drwxrwxr-x  6 ubuntu ubuntu  4096 Jun 10  2015 apache-maven-3.3.3
drwxr-xr-x 13 ubuntu ubuntu  4096 Jun 10  2015 mininet
drwxr-xr-x  8 ubuntu ubuntu  4096 Jun 10  2015 trema
drwxrwxr-x 10 ubuntu ubuntu  4096 Jun 10  2015 ryu
drwxrwxr-x  7 ubuntu ubuntu  4096 Jun 10  2015 pyretic
drwxrwxr-x 24 ubuntu ubuntu  4096 Jun 10  2015 openvswitch
drwxr-xr-x  9 ubuntu ubuntu  4096 Jun 10  2015 floodlight
drwxr-xr-x 11 ubuntu ubuntu  4096 Jun 10  2015 oftest
drwxrwxr-x  3 ubuntu ubuntu  4096 Jun 10  2015 eclipse-workspace
drwxr-xr-x 10 ubuntu ubuntu  4096 Jul 29  2015 SDNHub_Opendaylight_Tutorial
drwxr-xr-x 19 ubuntu ubuntu  4096 Jul 29  2015 onos
drwxr-xr-x  7 ubuntu ubuntu  4096 Feb  8 11:10 pox
ubuntu@sdnhubvm:~$ cd pox
ubuntu@sdnhubvm:~/pox$ ls -lrt
total 44
-rw-rw-r--  1 ubuntu ubuntu    87 Jun  1  2013 setup.cfg
lrwxrwxrwx  1 ubuntu ubuntu     6 May 19  2014 debug-pox.py -> pox.py
-rw-rw-r--  1 ubuntu ubuntu  1601 Jan  8  2015 README
-rwxrwxr-x  1 ubuntu ubuntu  1287 Jan  8  2015 pox.py
-rw-rw-r--  1 ubuntu ubuntu   892 Jan  8  2015 NOTICE
-rw-rw-r--  1 ubuntu ubuntu 10174 Jan  8  2015 LICENSE
drwxrwxr-x  2 ubuntu ubuntu  4096 Jan  8  2015 ext
drwxrwxr-x  3 ubuntu ubuntu  4096 Jan  8  2015 tests
drwxrwxr-x  2 ubuntu ubuntu  4096 Jan  8  2015 tools
drwxrwxr-x 15 ubuntu ubuntu  4096 Feb  8 08:17 pox
ubuntu@sdnhubvm:~/pox$ sudo ./pox.py forwarding.l2_learning
POX 0.5.0 (eel) / Copyright 2011-2014 James McCauley, et al.
INFO:core:POX 0.5.0 (eel) is up.
INFO:openflow.of_01:[00-00-00-00-00-01 3] connected
```

- Above images shows the directory that followed to reach the l2_learning python file which was used to study the switch. It is seen in the last line that openflow is connected.

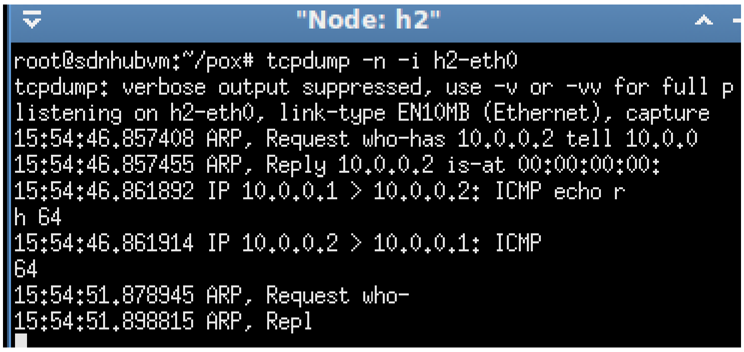


No.	Time	Source	Destination	Protocol	Length	Info
80	105.0339250(	127.0.0.1	127.0.0.1	TCP	66	36363→6633 [ACK] Seq=177 Ack=177 Win=86 Len=0 TSval=1397009 TSecr=1397009
81	109.9998330(	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REQUEST
82	110.0149840(	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REPLY
83	110.0150290(	127.0.0.1	127.0.0.1	TCP	66	36363→6633 [ACK] Seq=185 Ack=185 Win=86 Len=0 TSval=1398255 TSecr=1398255
84	114.9988900(	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REQUEST
85	115.0391920(	127.0.0.1	127.0.0.1	TCP	66	6633→36363 [ACK] Seq=185 Ack=193 Win=92 Len=0 TSval=1399511 TSecr=1399501
86	115.0478240(	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REPLY
87	115.0479180(	127.0.0.1	127.0.0.1	TCP	66	36363→6633 [ACK] Seq=193 Ack=193 Win=86 Len=0 TSval=1399513 TSecr=1399513
88	119.9998240(	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REQUEST
89	119.9998540(	127.0.0.1	127.0.0.1	TCP	66	6633→36363 [ACK] Seq=193 Ack=201 Win=92 Len=0 TSval=1400751 TSecr=1400751
90	120.0259380(	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REPLY
91	120.0259600(	127.0.0.1	127.0.0.1	TCP	66	36363→6633 [ACK] Seq=201 Ack=201 Win=86 Len=0 TSval=1400757 TSecr=1400757
92	125.0004280(	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REQUEST
93	125.0405060(	127.0.0.1	127.0.0.1	TCP	66	6633→36363 [ACK] Seq=201 Ack=209 Win=92 Len=0 TSval=1402011 TSecr=1402001
94	125.0426810(	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REPLY
95	125.0426950(	127.0.0.1	127.0.0.1	TCP	66	36363→6633 [ACK] Seq=209 Ack=209 Win=86 Len=0 TSval=1402012 TSecr=1402012
96	129.9999080(	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REQUEST
97	130.0000230(	127.0.0.1	127.0.0.1	TCP	66	6633→36363 [ACK] Seq=209 Ack=217 Win=92 Len=0 TSval=1403251 TSecr=1403251
98	130.0157490(	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REPLY
99	130.0158340(	127.0.0.1	127.0.0.1	TCP	66	36363→6633 [ACK] Seq=217 Ack=217 Win=86 Len=0 TSval=1403255 TSecr=1403255

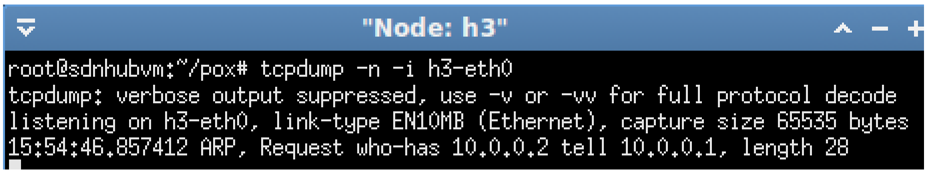
- The above snapshot is from wireshark where open flow packets are traced real time. It follows OFPT protocol. Echo request and Echo reply Messages are exchanged between them.



```
root@sdnhubvm:~/pox# h1 ping -c1 10.0.0.2
h1: command not found
root@sdnhubvm:~/pox# ping -c1 10.0.0.2
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=32.4 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 32.461/32.461/32.461/0.000 ms
root@sdnhubvm:~/pox#
```



```
root@sdnhubvm:~/pox# tcpdump -n -i h2-eth0
tcpdump: verbose output suppressed, use -v or -vv for full p
listening on h2-eth0, link-type EN10MB (Ethernet), capture
15:54:46.857408 ARP, Request who-has 10.0.0.2 tell 10.0.0
15:54:46.857455 ARP, Reply 10.0.0.2 is-at 00:00:00:00:
15:54:46.861892 IP 10.0.0.1 > 10.0.0.2: ICMP echo r
h 64
15:54:46.861914 IP 10.0.0.2 > 10.0.0.1: ICMP
64
15:54:51.878945 ARP, Request who-
15:54:51.898815 ARP, Repl
```



```
root@sdnhubvm:~/pox# tcpdump -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 65535 bytes
15:54:46.857412 ARP, Request who-has 10.0.0.2 tell 10.0.0.1, length 28
```

- The above three images shows the graphical representation of message flow.

6) Conclusion:

This lab thus made us understand the basic mininet terminology, for what it can be used. It also helped us learn the switch. When the openflow packets are passes through switch concepts and when packets are sent from controller to one respective host, all of them get the message just once. Once the controller Knows which host is it actually directed to, further packets are only sent to that host and thus flooding of packets at all ports is prevented.