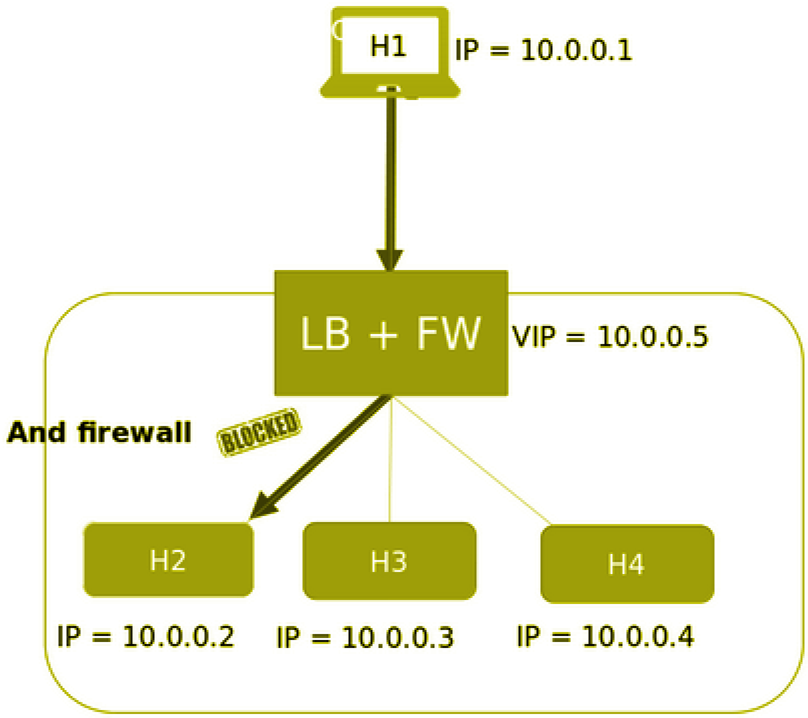



# Implementation of Load Balancer and Firewalls in Software Defined Networking

*Combined load balancer and firewall in one module and implemented on POX controller*

Rakesh Sukla, Pradhyothana Bijja | Eric Brendel, Software Defined Networking

# Table of Contents

# What is Load Balancer

*A load balancer is a device that distributes network or application traffic across a number of servers*

Requests are received by load balancers and they are distributed to a particular server based on a configured algorithm. Some standard algorithms are:

- Round robin
- Weighted round robin
- Least connections
- Least response time

Load balancers ensure reliability and availability by monitoring that the server is not getting overloaded.

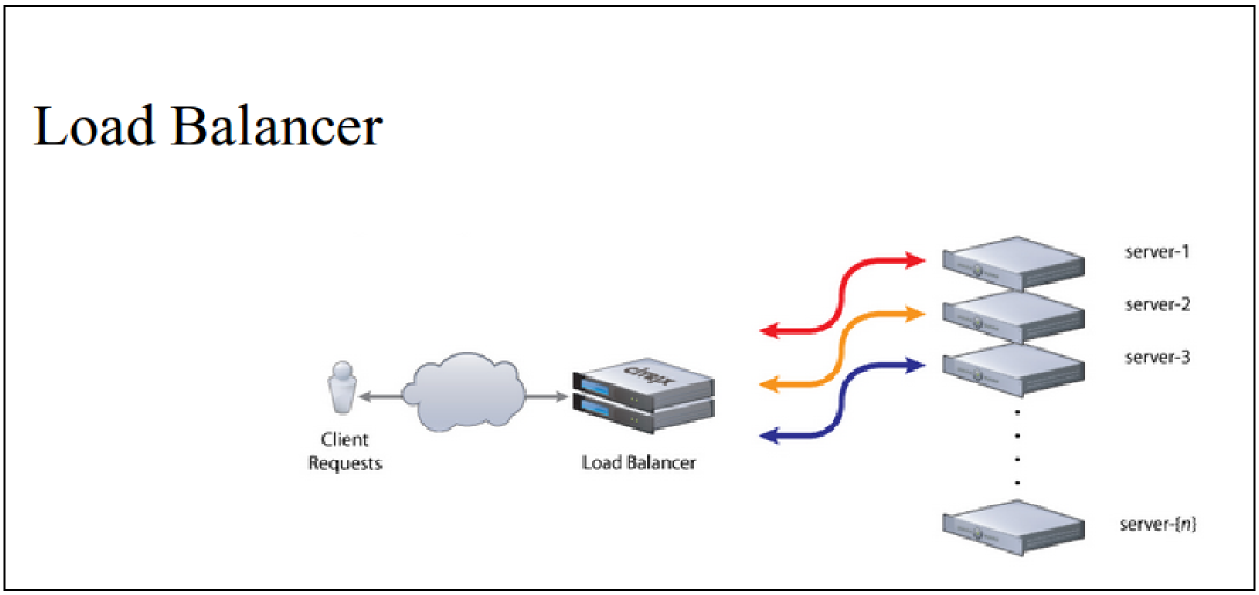



Figure 1

The advantages of using Load Balancer:

1- Optimize resource use

2- Maximize throughput

3- Minimize response time

4- Avoid overload of any single resource.

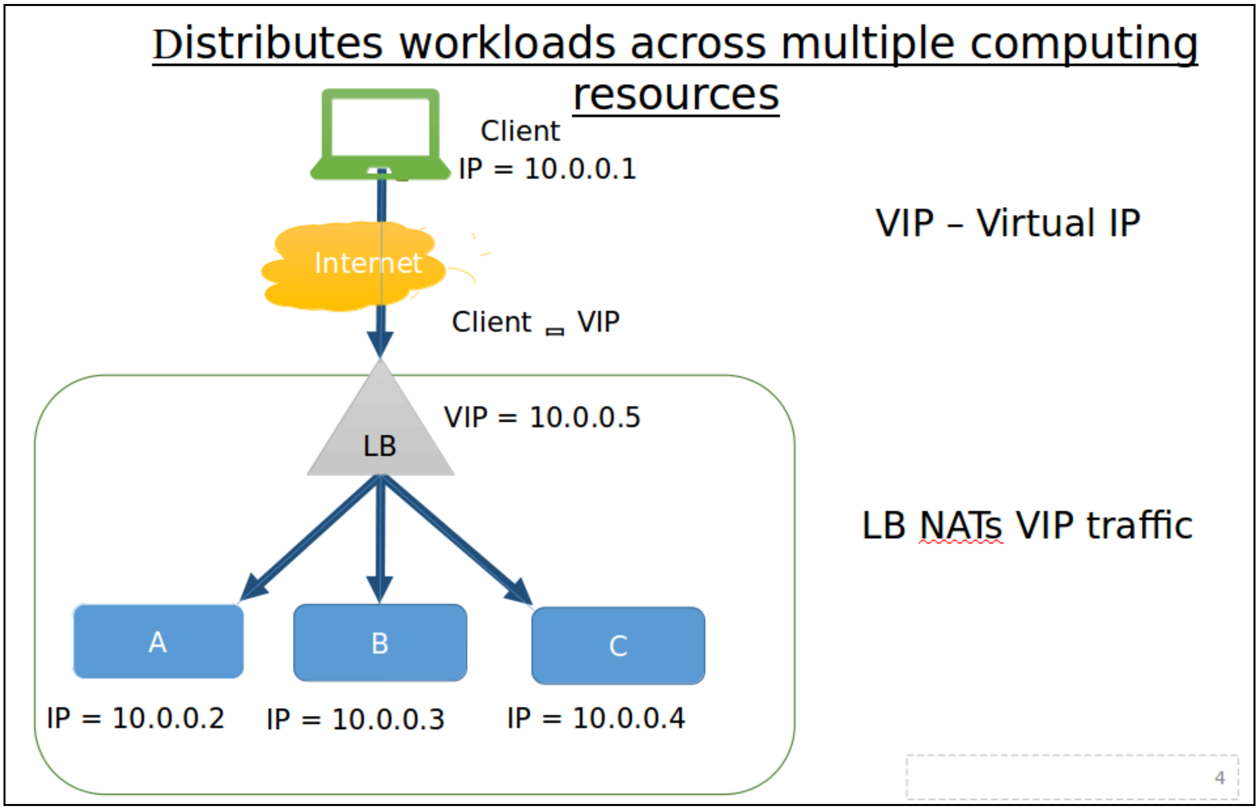
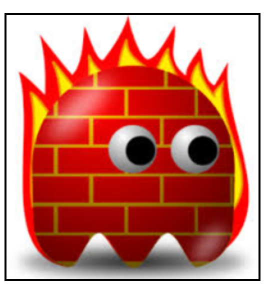



Figure 2

## What is Firewall

*A part of a computer system or network that is designed to block unauthorized access while permitting outward communication*

A firewall establishes a barrier between a trusted, secure internal network and another network (e.g., the Internet) that is assumed not to be secure and trusted. Based on the rule set, we can control the behaviour of our traffic.



System system that controls network traffic based on rule set.

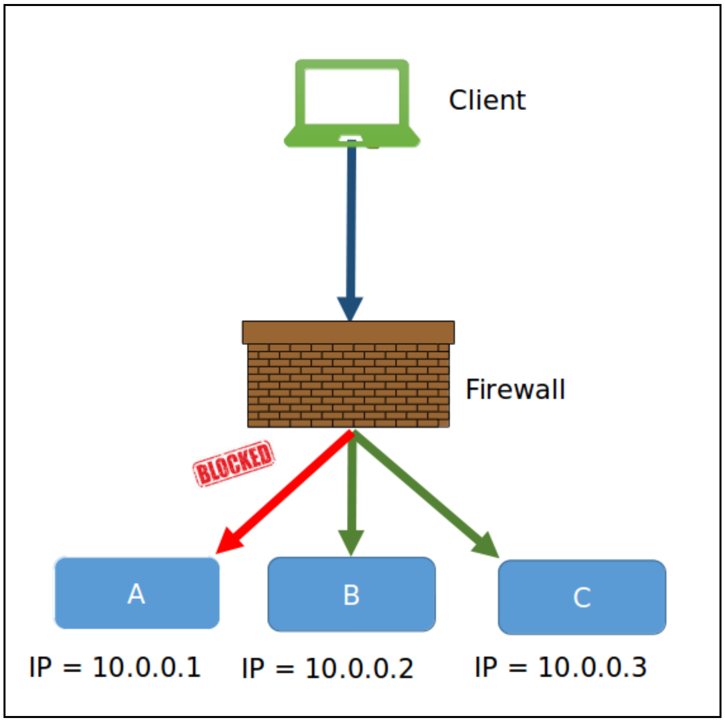



Figure 3

Why we add firewall in our system?

To establish a barrier between a trusted, secure internal network and another network that is assumed not to be secure and trusted.

## Implementation of Firewall and Load Balancer

In this project, we have implemented firewall and Load Balancer as a single module by using the function of modularity. Both the steps are simultaneously executed , as we require our system to be perform both the functionality. The figure below shows how the implementation has actually happened.

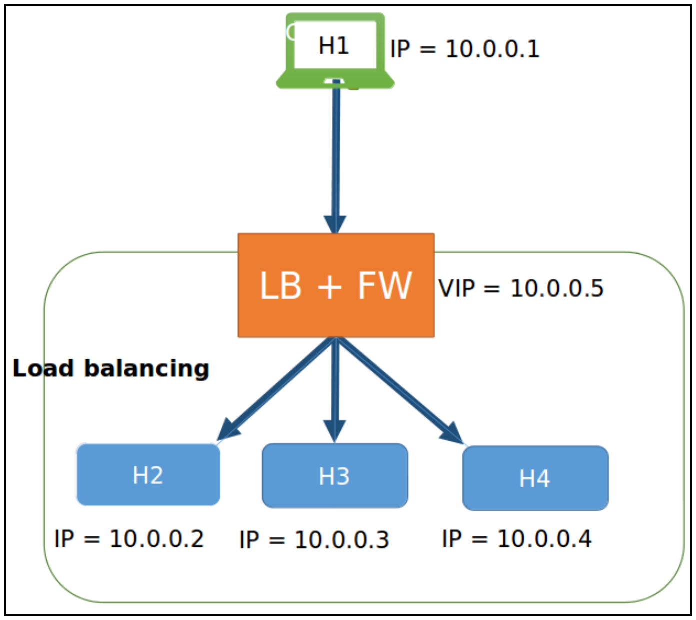



Figure 4

After adding firewall in the circuit:

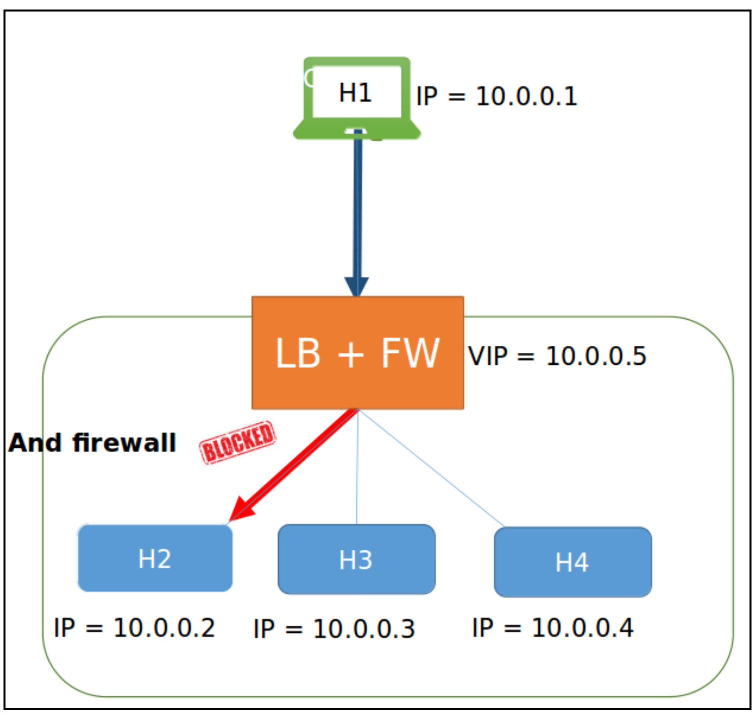



Figure 5

Implementation of whole idea in this figure:

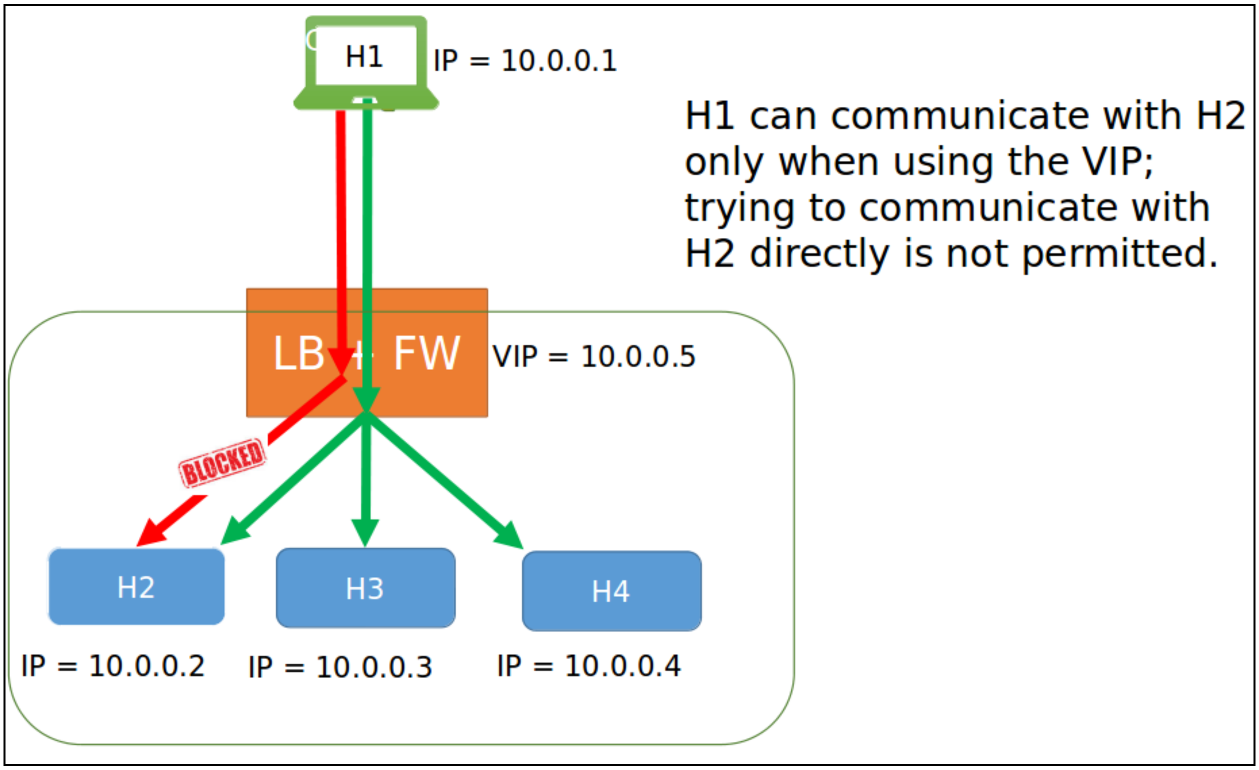



Figure 6

# POX Controller

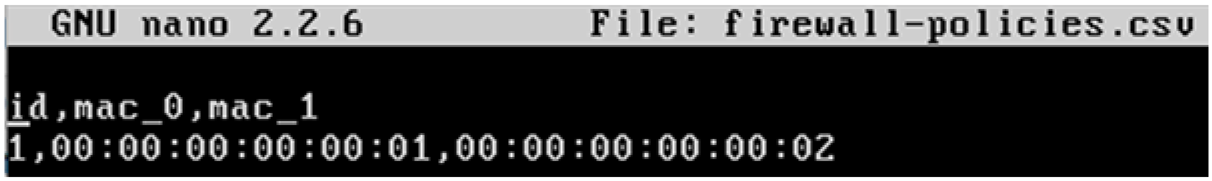
Pox is a Python based SDN controller. Some of the features of this controller:

- Pythonic" OpenFlow interface.
- Reusable sample components for path selection, topology discovery, etc.
- "Runs anywhere" – Can bundle with install-free PyPy runtime for easy deployment.
- Specifically targets Linux, Mac OS, and Windows.
- Topology discovery.
- Supports the same GUI and visualization tools as NOX.
- Performs well compared to NOX applications written in Python

**Source - NOXrepo website**

## Walkthrough

The firewall rules can be modified to block any traffic as needed by modifying the rules in firewall-policies.csv. Currently we have only one rule as can be seen below- this rule blocks any traffic between H1 and H2.



```
GNU nano 2.2.6            File: firewall-policies.csv

id,mac_0,mac_1
1,00:00:00:00:00:01,00:00:00:00:00:02
```

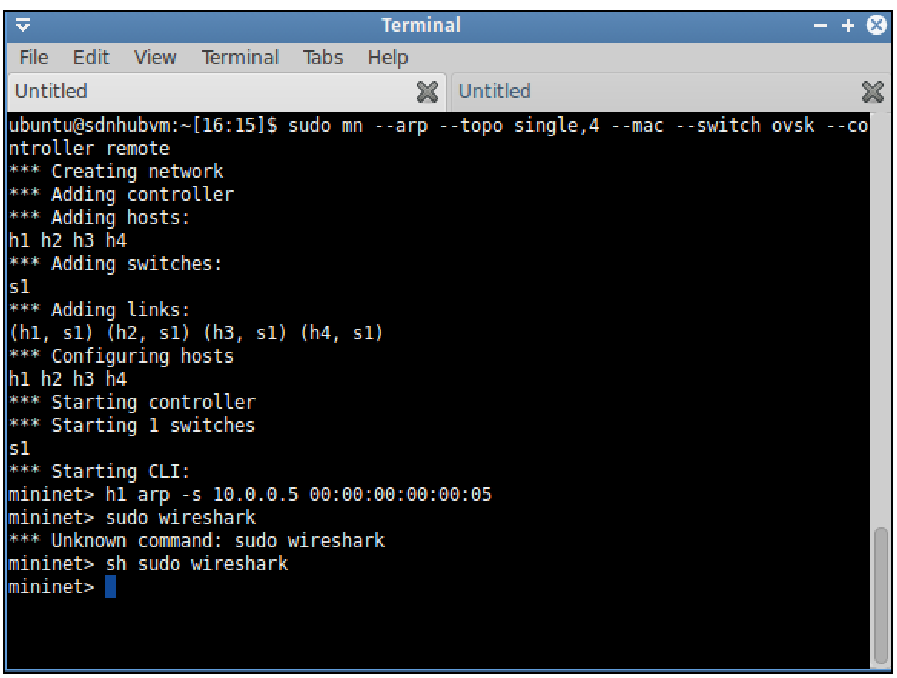
Figure 7

To stop H1 from communicating directly with H2 and H3, the following lines can be added-

2,00:00:00:00:00:01, 00:00:00:00:00:03

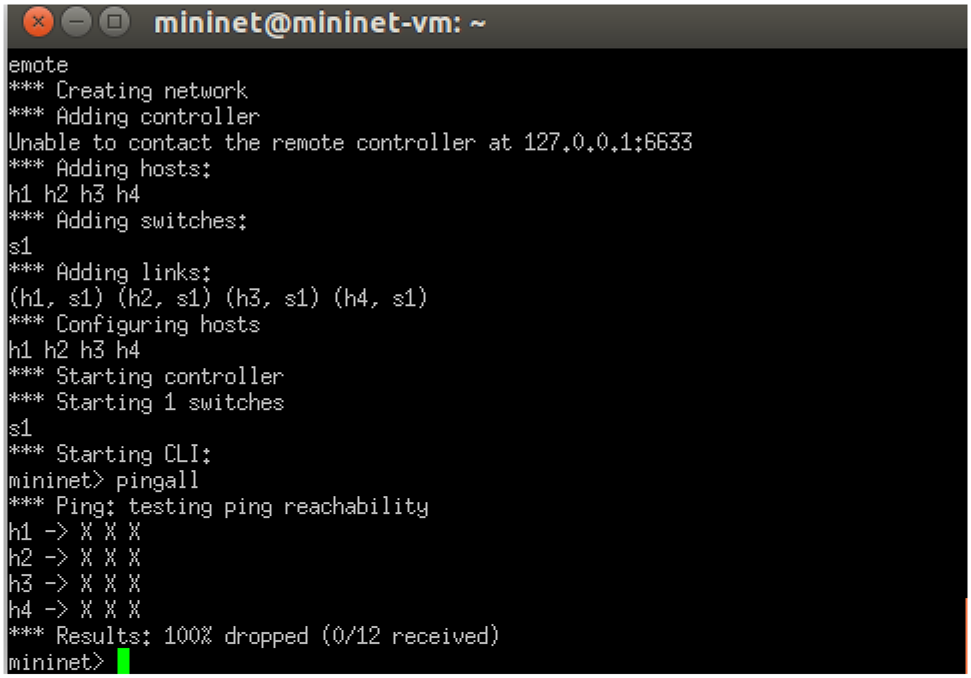
3,00:00:00:00:00:01, 00:00:00:00:00:04

**Step 1: Create a virtual network on mininet**

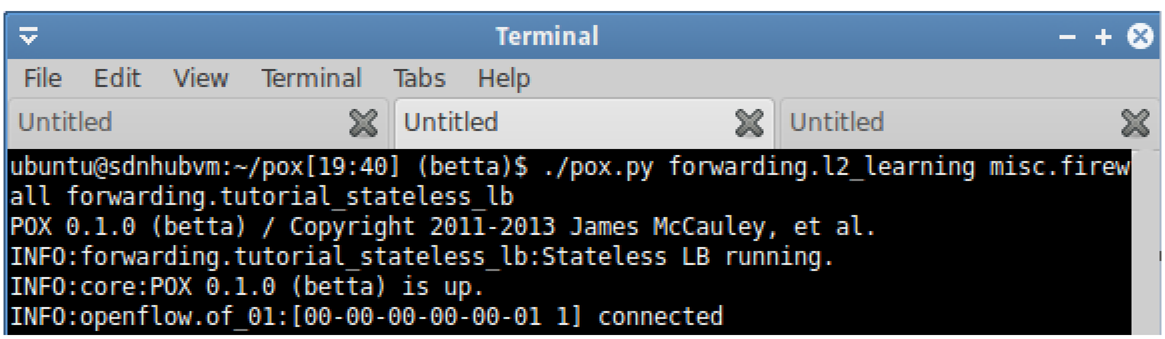


```
Terminal
File  Edit  View  Terminal  Tabs  Help
Untitled                                Untitled
ubuntu@sdnhubvm:~[16:15]$ sudo mn --arp --topo single,4 --mac --switch ovsk --co
ntroller remote
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1) (h4, s1)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
*** Starting 1 switches
s1
*** Starting CLI:
mininet> h1 arp -s 10.0.0.5 00:00:00:00:00:05
mininet> sudo wireshark
*** Unknown command: sudo wireshark
mininet> sh sudo wireshark
mininet>
```

### Step 2: No connectivity without any controller




### Step 3: Starting up the controller




### Code for Mac Learning:

```
l2_learning.py

from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.util import dpid_to_str
```

```
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):

  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()
      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:
    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:
        port = self.macToPort[packet.dst]
        if port == event.port: # 5
          # 5a
          log.warning("Same port for packet from %s -> %s on %s.%s.  Drop."
              % (packet.src, packet.dst, dpid_to_str(event.dpid), port))
          drop(10)
          return
        # 6
        log.debug("installing flow for %s.%i -> %s.%i" %
                  (packet.src, event.port, packet.dst, port))
        msg = of.ofp_flow_mod()
        msg.match = of.ofp_match.from_packet(packet, event.port)
        msg.idle_timeout = 10
        msg.hard_timeout = 30
        msg.actions.append(of.ofp_action_output(port = port))
        msg.data = event.ofp # 6a
        self.connection.send(msg)


class l2_learning (object):
  """
  Waits for OpenFlow switches to connect and makes them learning switches.
  """
  def __init__ (self, transparent):
    core.openflow.addListeners(self)
    self.transparent = transparent

  def _handle_ConnectionUp (self, event):
    log.debug("Connection %s" % (event.connection,))
    LearningSwitch(event.connection, self.transparent)
```

```
def launch (transparent=False, hold_down=_flood_delay):
  """
  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")

  core.registerNew(l2_learning, str_to_bool(transparent))


Displaying l2_learning.py.
```

**Code for Firewall:**

```
from pox.core import core

import pox.openflow.libopenflow_01 as of

from pox.lib.revent import *

from pox.lib.util import dpidToStr

from pox.lib.addresses import EthAddr

from collections import namedtuple

import os

import csv

log = core.getLogger()

policyFile = "%s/pox/pox/misc/firewall-policies.csv" % os.environ[ 'HOME' ]
```

```
''' Add your global variables here ... '''

class Firewall (EventMixin):

  def __init__ (self):

      self.listenTo(core.openflow)

      log.debug("Enabling Firewall Module")

      self.deny = []

      with open(policyFile, 'rb') as f:

          reader = csv.DictReader(f)

          for row in reader:

              self.deny.append((EthAddr(row['mac_0']), EthAddr(row['mac_1'])))

              self.deny.append((EthAddr(row['mac_1']), EthAddr(row['mac_0'])))

  def _handle_ConnectionUp (self, event):

      for (src, dst) in self.deny:

          match = of.ofp_match()

          match.dl_src = src

          match.dl_dst = dst

          msg = of.ofp_flow_mod()

          msg.match = match

          event.connection.send(msg)

      log.debug("Firewall rules installed on %s", dpidToStr(event.dpid))

def launch ():
  core.registerNew(Firewall)
```

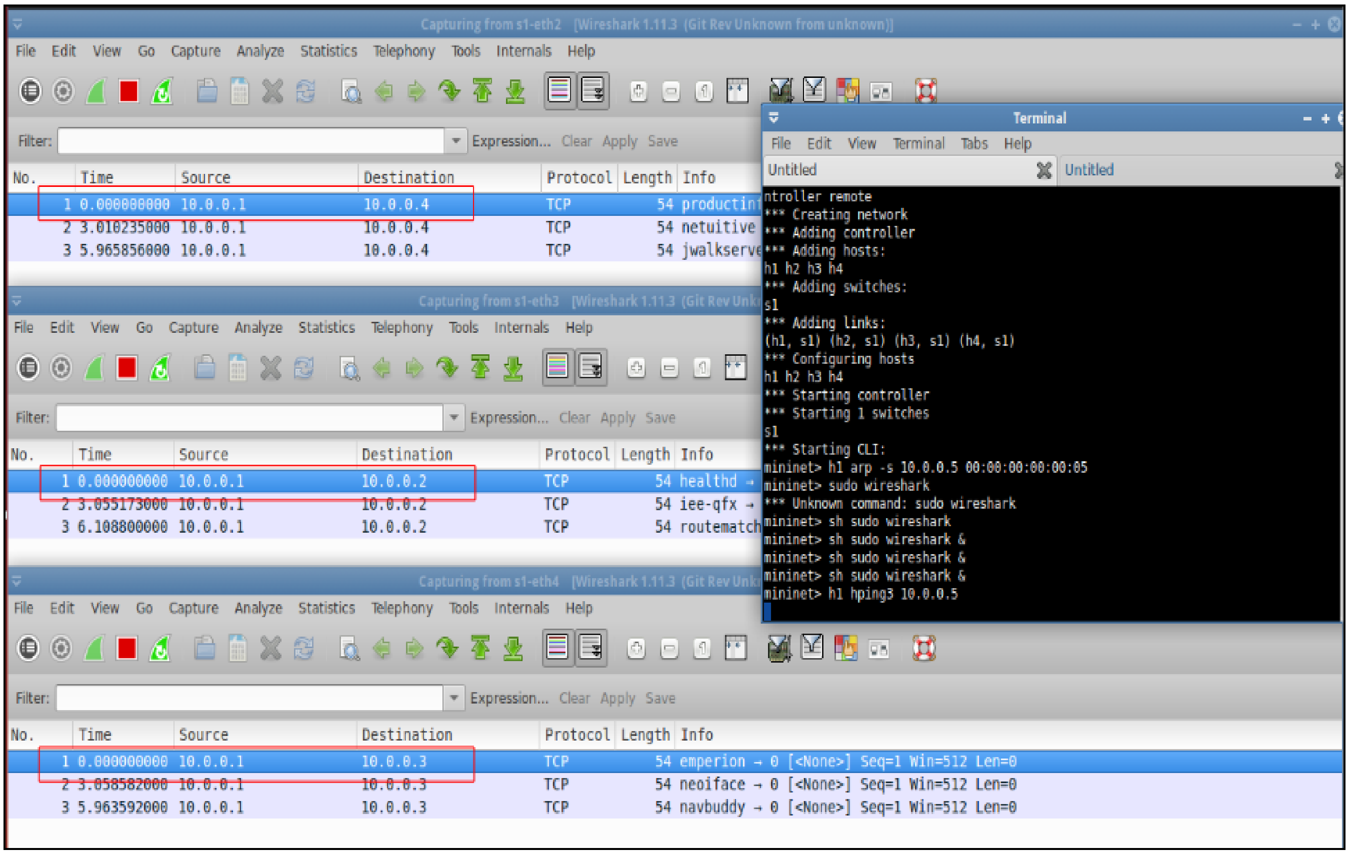
### Hping3

It is a network tool able to send custom TCP/IP packets.

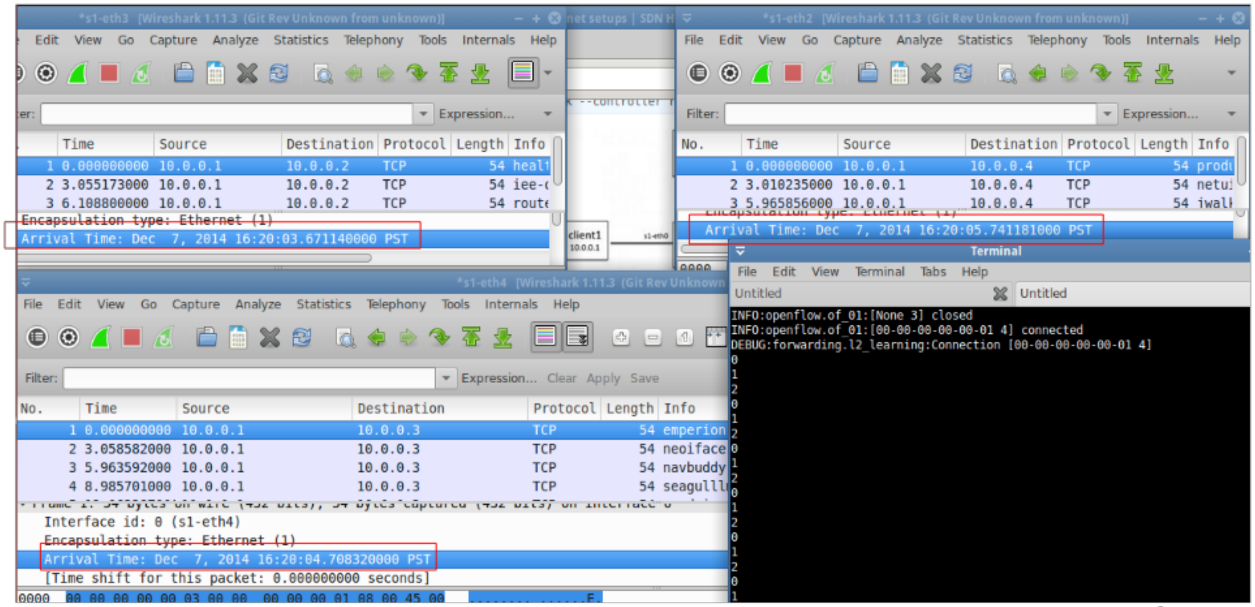
TCP packets will be sent out from H1 to the VIP 10.0.0.5

The load balancer will send out these packets to H2, H3 and H4 in round robin fashion.

**Screenshots of result:**



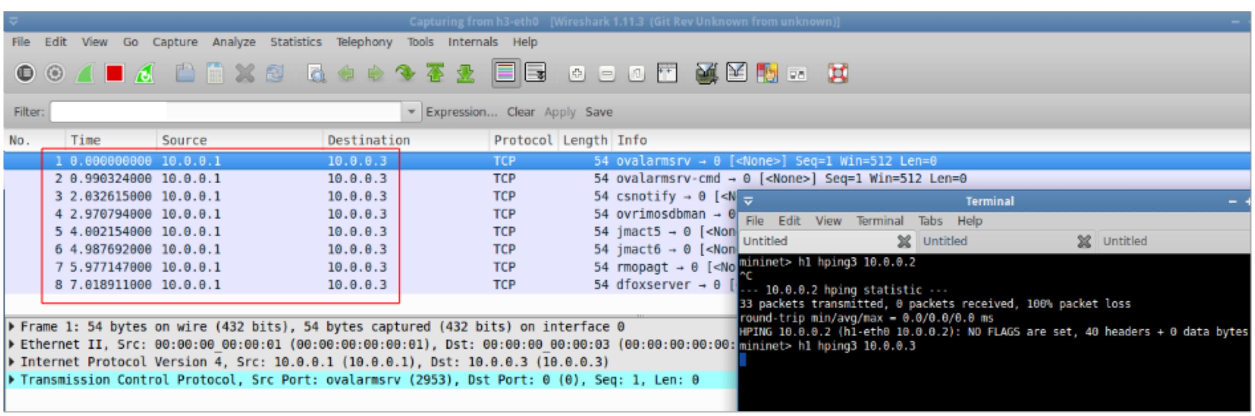
## Arrival times of the packets




1

# h1 hping3 10.0.0.3

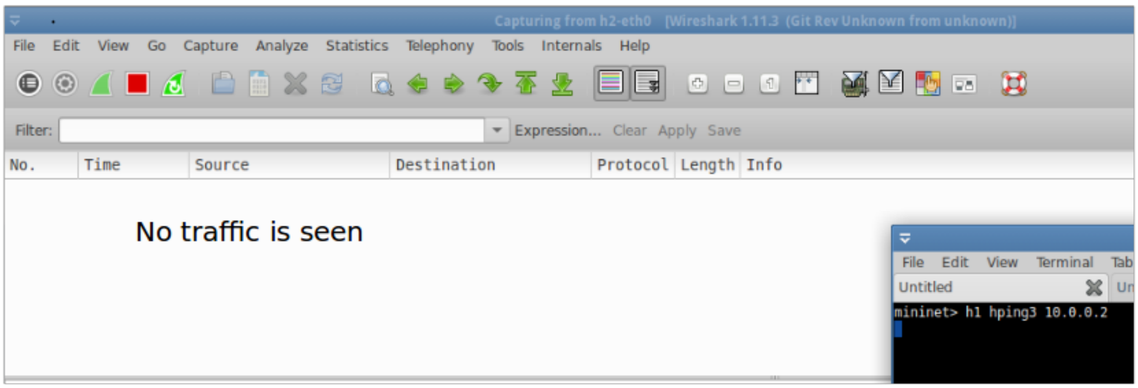
- TCP packets will be sent out from H1 to H3 directly
- The firewall won't block these packets.




1

# h1 hping3 10.0.0.2

- TCP packets will be sent out from H1 to H3 directly
- The firewall will block these packets.




1

## Lesson Learnt

SDN is a highly flexible when it comes to expanding a particular functionality. We can block the hosts just by adding a single line of code without much configuration changes.

Modularity- load balancing and firewall were implemented as separate pox modules; they can be used separately or together.

Adding a firewall rule involves including just one line.

Adding more hosts for a VIP would require minor changes in code- very useful when dynamically creating Virtual Machines in cloud computing/data centers.