894865609 - Uday Margadi
891037731 - Anant Pahuja
887347102 - Asmita Koirala
887347474 - Srinidhi Pande
892057126 - Freddy Aklobessi

CPSC 558 Advanced Networking - Spring 2020

Project Report

SDN POX Controller for Network Load Balancing

The problem that we are solving involves writing an algorithm able to manage network traffic using POX load balancer.

To accomplish this we used:

- Virtual Box, Ubuntu 16.04 OS,
- Mininet, Mininet comes along with a POX Controller.

Algorithms implemented:

- Random load balancing strategy which is included in Mininet.
- Round-Robin
- Weighted Round-Robin
- Least Bandwidth Method

After a team brainstorming and considerations, we decided to go with Random load balancing, Round-Robin, Weighted Round-Robin and Least Bandwidth Method algorithms and the following is the code implemented based on examples from our research.

Algorithm Portion Implemented also accessible through the following link in GitHub
https://github.com/udayreddy29/CPSC_558

Highlighted portions are the team implementation

```
# Copyright 2013,2014 James McCauley

# Licensed under the Apache License, Version 2.0 (the "License");

y  not use this file except in compliance with the License.

# You may obtain a copy of the License at:

#

#         http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
or implied.

# See the License for the specific language governing permissions and

# limitations under the License.

"""

A very sloppy IP load balancer.

Run it with --ip=<Service IP> --servers=IP1,IP2,...

By default, it will do load balancing on the first switch that connects.  If

you want, you can add --dpid=<dpid> to specify a particular switch.


Please submit improvements. :)

"""

from pox.core import core
```

```python
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 = 5
FLOW_MEMORY_TIMEOUT = 60 * 5
UPDATE_DATA_FLOW = 12
class MemoryEntry (object):
  """
  Record for flows we are balancing
  Table entries in the switch "remember" flows for a period of time, but
  rather than set their expirations to some long value (potentially leading
  to lots of rules for dead connections), we let them expire from the
  switch relatively quickly and remember them here in the controller for
  longer.
  Another tactic would be to increase the timeouts on the switch and use
  the Nicira extension which can match packets with FIN set to remove them
  when the connection closes.
  """
```

```
  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):
  """
```

A simple IP load balancer

Give it a service_ip and a list of server IP addresses. New TCP flows

to service_ip will be randomly redirected to one of the servers.

We probe the servers to see if they're alive by sending them ARPs.

"""

```
def __init__ (self, connection, service_ip,
servers,method,weights,loadBalancerType):

    self.service_ip = IPAddr(service_ip)

    self.servers = [IPAddr(a) for a in servers]

    self.method = method

    self.con = connection

    self.mac = self.con.eth_addr

    self.weights = weights

    self.live_servers = {} # IP -> MAC,port

    self.loadBalancerType = loadBalancerType
  //push IP's into self.select_servers to perform our load distribution

    self.select_servers = []

  //add weights to respective IP address into self.select_servers to perform our
load distribution

    self.server_weights = {}

    for index,ip in enumerate(self.servers):

    self.server_weights[ip] = weights[index]

  //if chosen algorithm is weighted round robin

    if loadBalancerType == 2:

    self.select_servers = []
```

```
for index,server in enumerate(self.servers):

temp = []

temp.append(server)

//pushing IP address according to the weights of IP address
    self.select_servers = self.select_servers + temp *
int(self.server_weights[server])

else:

self.select_servers = self.servers

log.info('selected server list is {}'.format(self.select_servers))

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.last_update = time.time()

# Data transferred map (IP -> data transferred in the last

# UPDATE_DATA_FLOW seconds).

    //we use data_flow to maintain data transferred on each server host. We use
this during the least bandwidth method based on the input weight of the host.

self.data_flow = {}

for server in self.servers:
```

```
        self.data_flow[server] = 0
        # We remember where we directed flows so that if they start up again,
        # we can send them to the same server if it's still up.  Alternate
        # approach: hashing.
        self.memory = {} # (srcip,dstip,srcport,dstport) -> MemoryEntry
        self._do_probe() # Kick off the probing
        # As part of a gross hack, we now do this from elsewhere
        #self.con.addListeners(self)

    def _do_expire (self):
        """
        Expire probes and "memorized" flows
        Each of these should only have a limited lifetime.
        """
        t = time.time()
        # Expire probes
        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]
        # del self.server_weights[ip]
        while ip in self.select_servers:
            self.select_servers.remove(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):
        """
        Send an ARP to a server to see if it's still up
        """
        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
```

//if the chosen algorithm is round_robin or weighted_round_robin, we call this function.

```
  def round_robin(self):
        choose_server = self.select_servers.pop(0)
          self.select_servers.append(choose_server)
        log.info('server chosen is {} using round_robin'.format(choose_server))
        return choose_server
```

//if the chosen algorithm is a random based load balancing strategy, we call this function.

```
  def random_selection(self):

        choose_server = random.choice(self.live_servers.keys())

        log.info('server chosen is {} using random
method'.format(choose_server))

        return choose_server
```

//if the chosen algorithm is a least bandwidth based load balancing strategy, we call this function.

```
  def least_bandwidth(self):

        servers = self.servers

        weights = self.server_weights

        data_flow = self.data_flow

        choose_server = self.servers[0]

        priorityValue = data_flow[choose_server] / int(weights[choose_server])

        for id in range(1,len(self.servers)):

        priorityValue2 = self.data_flow[servers[id]] / int(weights[servers[id]])

        if priorityValue > priorityValue2:

              choose_server = servers[id]

        log.info('server chosen is {} using least connection
method'.format(choose_server))

        return choose_server
```

//As soon as we hit request from client host, this function is called to select server

```
  def _select_server (self, key, inport):

        """

        select a server for a (hopefully) new connection

        """
```

```
if self.loadBalancerType == 0:
return self.random_selection()
elif self.loadBalancerType == 3:
return self.least_bandwidth()
else:
return self.round_robin()
def _handle_PacketIn (self, event):
inport = event.port
packet = event.parsed
#log.info('packet response {}'.format(packet))
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.data_flow[arpp.protosrc] = 0
# if arpp.protosrc not in self.weights.keys():
#  self.weights[arpp.protosrc] = 1
# tempServerList = []
# tempServerList.append(arpp.protosrc)
# self.select_servers += tempServerList
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')

# Update the data count table, if needed.
if time.time() - self.last_update > UPDATE_DATA_FLOW:
for server in self.data_flow.keys():
self.data_flow[server] = 0
self.last_update = time.time()
```

```
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()
# select a server for this flow
server = self._select_server(key, inport)
self.log.debug('selected server is %s', server)

# self.servers.append(server)
self.log.debug('re-arranged server list is %s %s',server,self.select_servers)
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,method='default',weights=[]):
  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)
  weights_selected = []
```

```
if weights and len(weights) > 0:
    weights_selected = weights.split(',')
else:
    for i in servers:
    weights_selected.append(1)

if len(weights_selected) is not len(servers):
    log.error('length of weights and servers are not equal')
    exit(1)
loadBalancerType = 0
if method == 'round_robin':
    loadBalancerType = 1
elif method == 'weighted_round_robin':
    loadBalancerType = 2
elif method == 'least_bandwidth':
    loadBalancerType = 3
# 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...
        # log.info('server_weights'.format(server_weights))
        core.registerNew(iplb, event.connection,
IPAddr(ip),servers,method,weights_selected,loadBalancerType)
        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)
def _handle_FlowStatsReceived (event):
        for data in event.stats:
        ip_dst = data.match.nw_dst
```

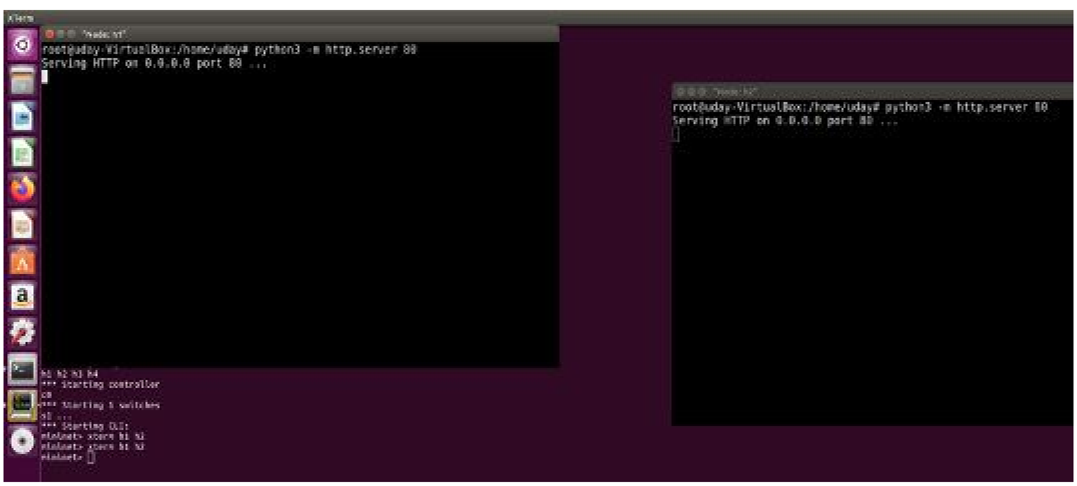
```
        ip_src = data.match.nw_src
        if ip_dst != None and IPAddr(ip_dst) in core.iplb.servers:
        core.iplb.data_flow[IPAddr(ip_dst)] += data.byte_count
        if ip_src != None and IPAddr(ip_src) in core.iplb.servers:
          core.iplb.data_flow[IPAddr(ip_src)] += data.byte_count
 core.openflow.addListenerByName("FlowStatsReceived",
_handle_FlowStatsReceived)
 core.openflow.addListenerByName("ConnectionUp", _handle_ConnectionUp)
 from pox.lib.recoco import Timer
 def _timer_getStats ():
        for connection in core.openflow._connections.values():
        connection.send(of.ofp_stats_request(body=of.ofp_flow_stats_request()))

 # Request flow stats every FLOW_IDLE_TIMEOUT second.
 Timer(FLOW_IDLE_TIMEOUT, _timer_getStats, recurring=True)
```

Demo screenshots

The below command, emulates a network topology with 4 hosts on a single switch.

Command: *mn --topo single,4 --controller=remote,port=6633*

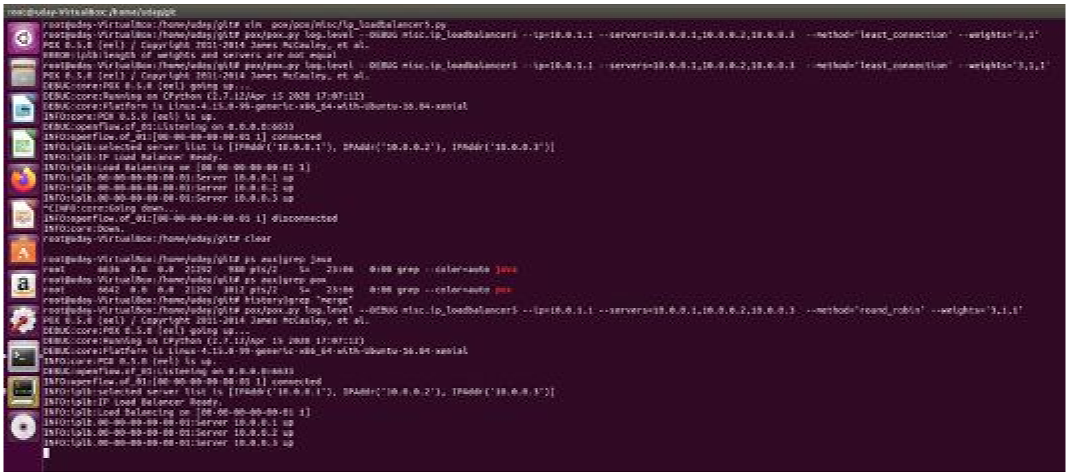
In the screen below, we are launching multiple hosts and running basic http server on port 80.

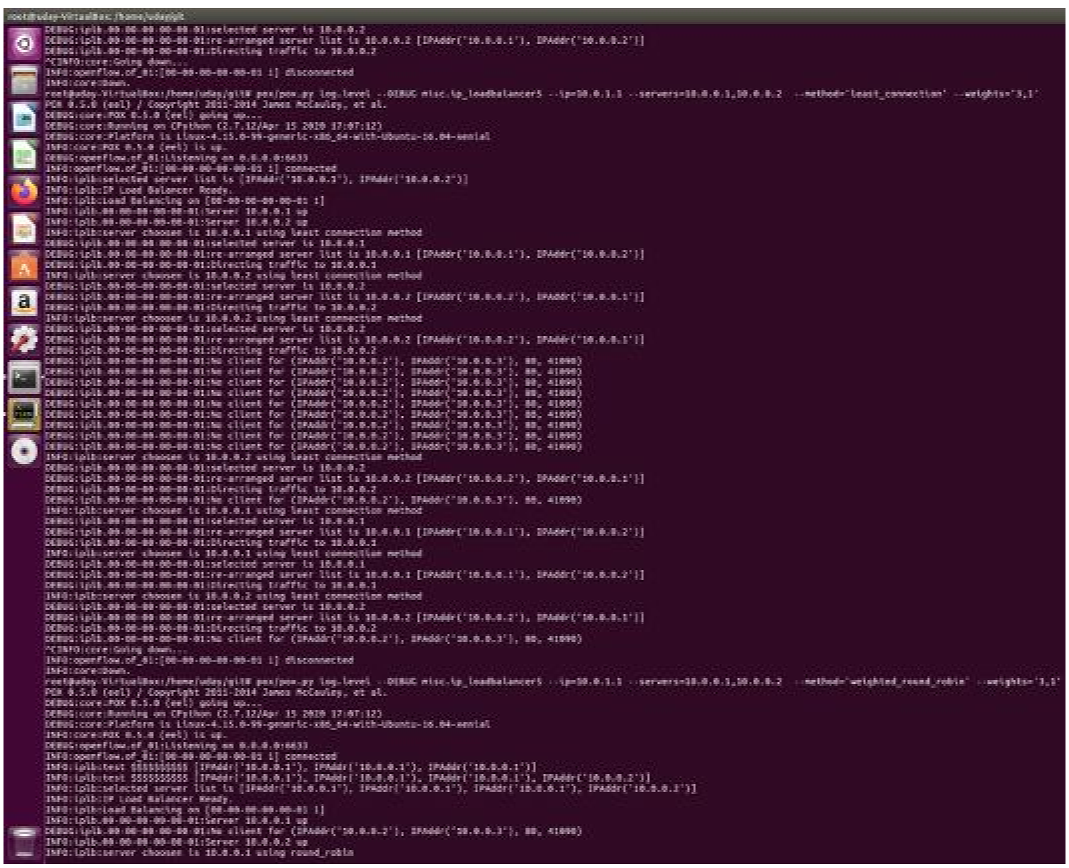


Now, starting the loadbalancer under pox controller and passing ip address, list of server hosts, load balancing algorithm and weights of server hosts.

Command: *pox/pox.py log.level --DEBUG misc.ip_loadbalancer --ip 10.0.1.1 --servers=10.0.0.1,10.0.0.2,10.0.0.3 --method='weighted_round_robin' --weights='3,1,1'*

Once we launch the pox controller, we will hit the load balancer from the client host. When we hit the load balancer, the packet will be sent to the controller and it will update the switch on how and to whom the request should be forwarded.





Load management/distribution

Results of our experiment

Round robin works best when all servers have roughly identical computing capabilities and storage capabilities. Requests are divided equally to servers.

When servers have non identical capabilities, servers are assigned with weights. Servers with higher weight receive a higher number of requests. This is called weighted round robin load balancing algorithms.

At least bandwidth, we look up in the data flow object and get data transferred to the host per weights associated with it. Compare this among the list of servers and choose the server with lowest value.

Some research have been done as well on the possibility of using OpenDayLight Controller and here are the findings:

- OpenDaylight is more advanced than POX

- Obtain path/route information (using Dijkstra thereby limiting search to shortest paths and only one segment of fat tree topology) from Host 1 to Host 2 i.e. the hosts between load balancing has to be performed.

- Total link cost between Host 1 and Host 2 gives only transmitted data
- Specify host between which we want to test load balancing

Some advantages of OpenDayLight

- More functionality
- Well built
- Production ready controller

Comparison with other controllers:

Controller Summary

	NOX	POX	Ryu	Floodlight	ODL OpenDaylight
Language	C++	Python	Python	JAVA	JAVA
Performance	Fast	Slow	Slow	Fast	Fast
Distributed	No	No	Yes	Yes	Yes
OpenFlow	1.0 / 1.3	1.0	1.0 to 1.4	1.0	1.0 / 1.3
Learning Curve	Moderate	Easy	Moderate	Steep	Steep
		Research, experimentation, demonstrations	Open source Python controller	Maintained Big Switch Networks	Vendor App support

Source: Georgia Tech SDN Class

World Wide Technology, Inc.

Conclusion

In Conclusion, This research was a good learning experience for the team in terms of applying hand on what has been conceptually explained in class.

Traditionally, doing load balancing, the administrator runs a preset code from each vendor therefore he will not be able to customize the code whereas with SDN controllers nowadays it’s much easier to alter the code based on the network administrator objectives.

With POX load balancer Controller, there is no need to worry about hardware and all can be done virtually with efficiency. In addition it’s cost effective.

One more thing we could have considered with more time is to implement a switch that would be able to check the status of a host (either up or down) before directing traffic to it.

We are thankful for the opportunity to have a better understanding of how Load balancing works using POX SDN Controller.

References

http://www.brianlinkletter.com/how-to-install-mininet-sdn-network-simulator/

https://www.citrix.com/en-in/glossary/load-balancing.html

https://www.imperva.com/learn/availability/load-balancing-algorithms/

https://github.com/mininet/mininet

https://ask.opendaylight.org/question/5312/how-to-get-shortest-path-between-two-hosts/

https://www.slideshare.net/joelwking/introduction-to-openflow-41257742

https://github.com/nayanseth/sdn-loadbalancing