VIII. APPENDIX

Details of equal_load.py file,

```
"""
copyright@ BITS Pilani
Bhavik Dhandhaly
Anand Wani
"""

"""
Threshold for this file has been calculated manually and tested using "tat.py" file
"""

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

import pox.openflow.libopenflow_01 as of

import time
import random

FLOW_IDLE_TIMEOUT = 10
FLOW_MEMORY_TIMEOUT = 60

ALPHA = 0


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 from_client_to_server(self):
        ethp = self.first_packet
        ipp = ethp.find('ipv4')
        tcpp = ethp.find('tcp')

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

    @property
    def from_server_to_client(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=[]):
```

```
    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.total_connection = {} # IP -> total connection
    for ip in servers:
      self.total_connection[ip] = 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]

        # Expire flow
        memory = self.memory.copy()
        self.memory.clear()
        for key, val in memory.items():
            ip = key[0]
            if ip in self.live_servers and val.is_expired:
                # Decrease total connection for that server
                self.total_connection[ip] -= 1
            if not val.is_expired:
                self.memory[key] = val

        # Show information
        # self.log.debug("Jumlah koneksi pada server:")
        # for item in self.total_connection:
        #     self.log.debug("%s = %s", item, self.total_connection[item])

        # 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

    def _pick_server(self, key, inport):
        """
        Pick a server for a (hopefully) new connection
        """
        global ALPHA

        if len(self.total_connection) == 0:
            return self.live_servers.keys()[0]
        ipserver = self.total_connection.keys()[0]
        totalconns = self.total_connection[ipserver]

        ALPHA = (ALPHA + 1) % 33
        if ALPHA < 28:
            for x in self.total_connection:
                if str(x) == '10.0.0.1':
                    ipserver = x
```

```
                    self.log.debug("Best available Server: %s" % ipserver)
                    return ipserver
        elif ALPHA < 31:
            for x in self.total_connection:
                if str(x) == '10.0.0.2':
                    ipserver = x
                    self.log.debug("Best available Server: %s" % ipserver)
                    return ipserver
        else:
            for x in self.total_connection:
                if str(x) == '10.0.0.3':
                    ipserver = x
                    self.log.debug("Best available Server: %s" % ipserver)
                    return ipserver

        """
        for x in self.total_connection:
            if self.total_connection[x] < totalconns:
                ipserver = x
                totalconns = self.total_connection[x]
        """
        #self.log.debug("Best available Server: %s" % ipserver)
        #return ipserver

        # if len(self.total_connection) == 0:
        #     return self.live_servers.keys()[0]
        # return min(self.total_connection, key=self.total_connection.get)


    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')

# Incoming packet from server
if ipp.srcip in self.servers:
  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)

# Incoming packet from client
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.from_client_to_server] = entry
        self.memory[entry.from_server_to_client] = entry

        # Increase total connection for that server
        self.total_connection[server] += 1

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

  # Boot up ARP Responder
  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:
      log.info("IP Load Balancer Ready.")
      core.registerNew(iplb, event.connection, IPAddr(ip), servers)
      _dpid = event.dpid

    if _dpid != event.dpid:
      log.warn("Ignoring switch %s", event.connection)
    else:
      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)
```

weightedRR.py

```
"""
copyright@ BITS Pilani
Bhavik Dhandhaly
Anand Wani
"""

"""
Threshold for this file has been calculated manually and tested using "tat.py" file
"""

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

import pox.openflow.libopenflow_01 as of

import time
import random

FLOW_IDLE_TIMEOUT = 10
FLOW_MEMORY_TIMEOUT = 60

ALPHA = 0



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 from_client_to_server(self):
        ethp = self.first_packet
        ipp = ethp.find('ipv4')
        tcpp = ethp.find('tcp')

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

    @property
    def from_server_to_client(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=[]):
  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.total_connection = {} # IP -> total connection
  for ip in servers:
    self.total_connection[ip] = 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]

        # Expire flow
        memory = self.memory.copy()
        self.memory.clear()
        for key, val in memory.items():
            ip = key[0]
            if ip in self.live_servers and val.is_expired:
                # Decrease total connection for that server
                self.total_connection[ip] -= 1
            if not val.is_expired:
                self.memory[key] = val

        # Show information
        # self.log.debug("Jumlah koneksi pada server:")
        # for item in self.total_connection:
        #     self.log.debug("%s = %s", item, self.total_connection[item])

        # 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

    def _pick_server(self, key, inport):
        """
        Pick a server for a (hopefully) new connection
        """
        global ALPHA

        if len(self.total_connection) == 0:
            return self.live_servers.keys()[0]
        ipserver = self.total_connection.keys()[0]
        totalconns = self.total_connection[ipserver]

        ALPHA = (ALPHA + 1) % 33
        if ALPHA < 28:
            for x in self.total_connection:
```

```
                if str(x) == '10.0.0.1':
                    ipserver = x
                    self.log.debug("Best available Server: %s" % ipserver)
                    return ipserver
        elif ALPHA < 31:
            for x in self.total_connection:
                if str(x) == '10.0.0.2':
                    ipserver = x
                    self.log.debug("Best available Server: %s" % ipserver)
                    return ipserver
        else:
            for x in self.total_connection:
                if str(x) == '10.0.0.3':
                    ipserver = x
                    self.log.debug("Best available Server: %s" % ipserver)
                    return ipserver

        """
        for x in self.total_connection:
            if self.total_connection[x] < totalconns:
                ipserver = x
                totalconns = self.total_connection[x]
        """
        #self.log.debug("Best available Server: %s" % ipserver)
        #return ipserver

        # if len(self.total_connection) == 0:
        #     return self.live_servers.keys()[0]
        # return min(self.total_connection, key=self.total_connection.get)


    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')

# Incoming packet from server
if ipp.srcip in self.servers:
  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)

# Incoming packet from client
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.from_client_to_server] = entry
        self.memory[entry.from_server_to_client] = entry

        # Increase total connection for that server
        self.total_connection[server] += 1

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

  # Boot up ARP Responder
  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:
      log.info("IP Load Balancer Ready.")
      core.registerNew(iplb, event.connection, IPAddr(ip), servers)
      _dpid = event.dpid

    if _dpid != event.dpid:
      log.warn("Ignoring switch %s", event.connection)
    else:
      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)
```