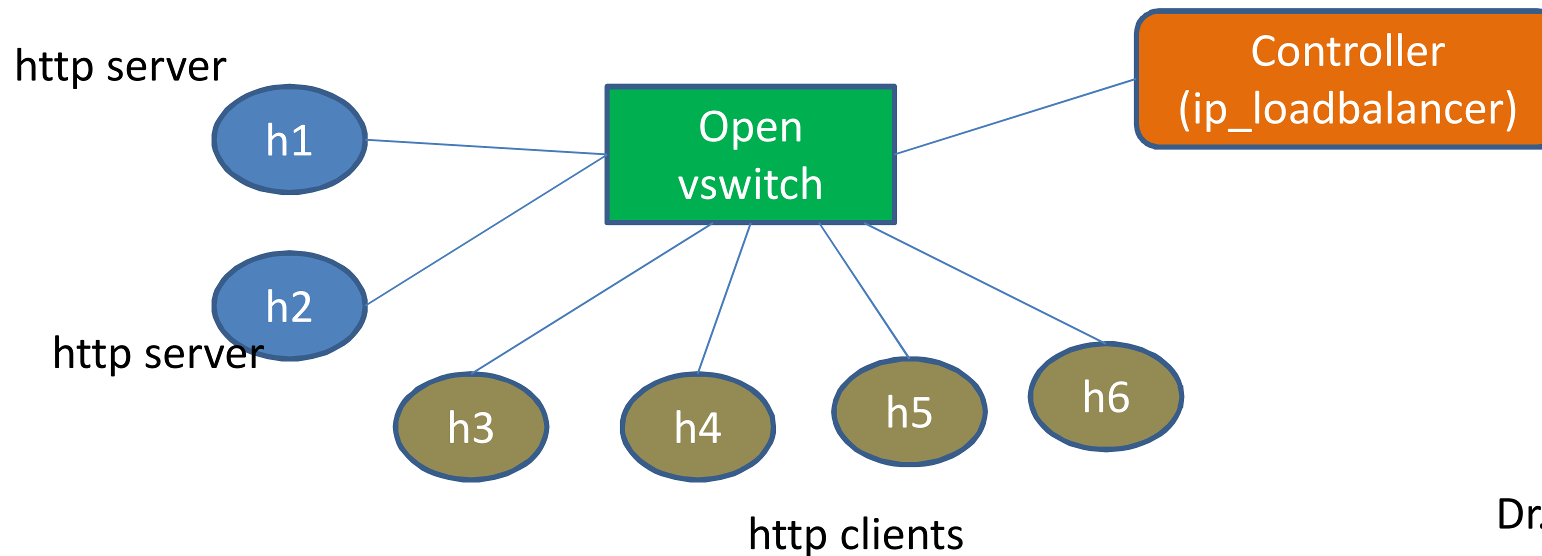
Lab 8 (IP Load Balancer)

- In this lab, the http requests from different clients will be directed to different pre-defined http servers. The server is chosen based on round robin scheduling.



Dr. Chih-Heng Ke
smallko@gmail.com

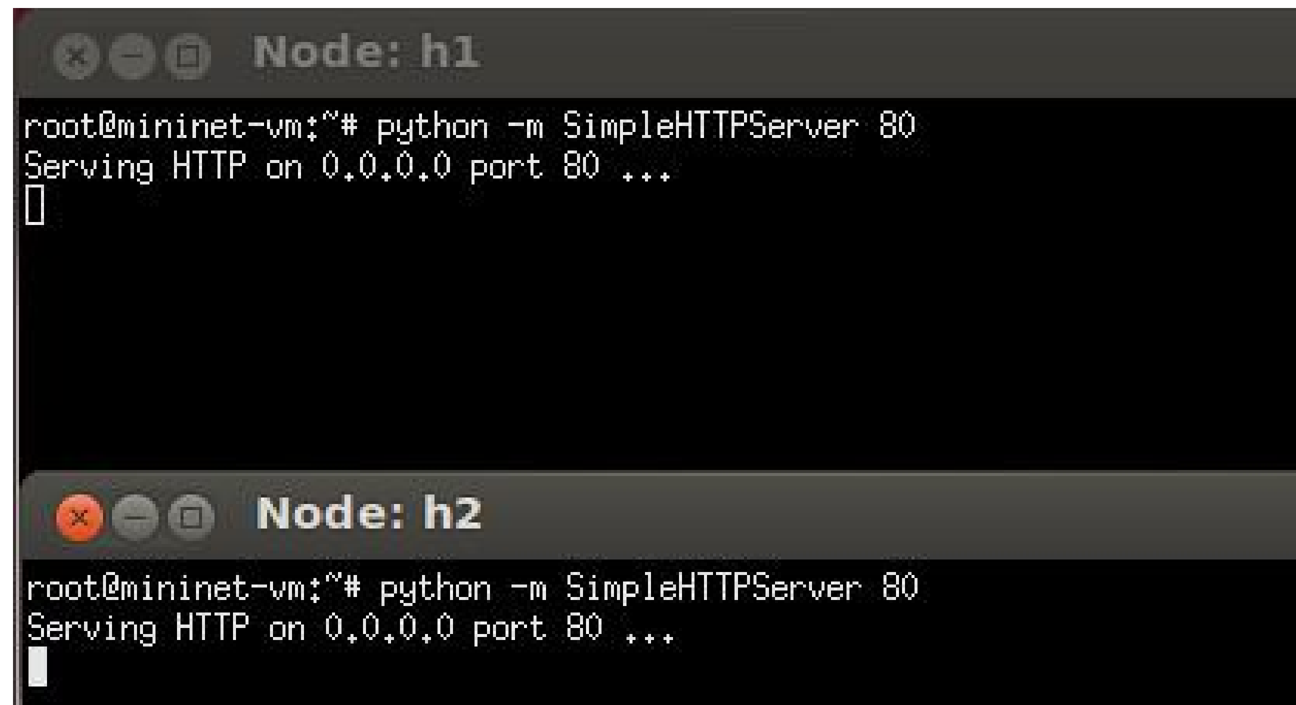
mininet@mininet-vm: ~/pox

```
mininet@mininet-vm:~/pox$ ./pox.py log.level --DEBUG misc.ip_loadbalancer --ip=1
0.0.1.1 --servers=10.0.0.1,10.0.0.2
POX 0.1.0 (betta) / Copyright 2011-2013 James McCauley, et al.
DEBUG:core:POX 0.1.0 (betta) going up...
DEBUG:core:Running on CPython (2.7.3/Sep 26 2013 20:08:41)
DEBUG:core:Platform is Linux-3.2.0-49-generic-i686-with-Ubuntu-12.04-precise
INFO:core:POX 0.1.0 (betta) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[None 1] closed
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
INFO:iplb:IP Load Balancer Ready.
INFO:iplb:Load Balancing on [00-00-00-00-00-01 2]
INFO:iplb.00-00-00-00-00-01:Server 10.0.0.1 up
INFO:iplb.00-00-00-00-00-01:Server 10.0.0.2 up
```

/bin/bash

```
mininet@mininet-vm:~$ sudo mn --topo single,6  --mac --arp --controller=remote
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5 h6
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1) (h4, s1) (h5, s1) (h6, s1)
*** Configuring hosts
h1 h2 h3 h4 h5 h6
*** Starting controller
*** Starting 1 switches
s1
*** Starting CLI:
mininet> xterm h1 h2 h3 h4 h5 h6
```

Start http server on h1 and h2



Get the webpage from 10.0.1.1

Node: h1

```
root@mininet-vm:~# python -m SimpleHTTPServer 80
Serving HTTP on 0.0.0.0 port 80 ...
10.0.0.3 - - [27/Jan/2014 05:49:18] "GET / HTTP/1.1" 200 -
```

Node: h2

```
root@mininet-vm:~# python -m SimpleHTTPServer 80
Serving HTTP on 0.0.0.0 port 80 ...
```

Node: h3

```
root@mininet-vm:~# curl 10.0.1.1
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><html>
<title>Directory listing for /</title>
<body>
<h2>Directory listing for /</h2>
<hr>
<ul>
<li><a href=".bash_history">.bash_history</a>
<li><a href=".bash_logout">.bash_logout</a>
<li><a href=".bashrc">.bashrc</a>
<li><a href=".cache/">.cache/</a>
<li><a href=".config/">.config/</a>
<li><a href=".dbus/">.dbus/</a>
<li><a href=".eclipse/">.eclipse/</a>
<li><a href=".filezilla/">.filezilla/</a>
<li><a href=".fontconfig/">.fontconfig/</a>
<li><a href=".gconf/">.gconf/</a>
2/">.gnome2/</a>
2_private/">.gnome2_private/</a>
amer-0.10/">.gstreamer-0.10/</a>
ookmarks">.gtk-bookmarks</a>
">.gvfs/</a>
thority">.ICEauthority</a>
">.lftp/</a>
```

```
root@mininet
```

mininet@mininet-vm: ~/pox

```
mininet@mininet-vm:~/pox$ ./pox.py log.level --DEBUG misc.ip_loadbalancer --ip=1
0.0.1.1 --servers=10.0.0.1,10.0.0.2
POX 0.1.0 (betta) / Copyright 2011-2013 James McCauley, et al.
DEBUG:core:POX 0.1.0 (betta) going up...
DEBUG:core:Running on CPython (2.7.3/Sep 26 2013 20:08:41)
DEBUG:core:Platform is Linux-3.2.0-49-generic-i686-with-Ubuntu-12.04-precise
INFO:core:POX 0.1.0 (betta) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[None 1] closed
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
INFO:iplb:IP Load Balancer Ready.
INFO:iplb:Load Balancing on [00-00-00-00-00-01 2]
INFO:iplb.00-00-00-00-00-01:Server 10.0.0.1 up
INFO:iplb.00-00-00-00-00-01:Server 10.0.0.2 up
Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
```

Node: h1

```
root@mininet-vm:~# python -m SimpleHTTPServer 80
Serving HTTP on 0.0.0.0 port 80 ...
10.0.0.3 - - [27/Jan/2014 05:49:18] "GET / HTTP/1.1" 200 -
10.0.0.5 - - [27/Jan/2014 05:52:30] "GET / HTTP/1.1" 200 -
```

Node: h2

```
root@mininet-vm:~# python -m SimpleHTTPServer 80
Serving HTTP on 0.0.0.0 port 80 ...
10.0.0.4 - - [27/Jan/2014 05:52:22] "GET / HTTP/1.1" 200 -
10.0.0.6 - - [27/Jan/2014 05:52:46] "GET / HTTP/1.1" 200 -
```

Node: h3

```
root@mininet-vm:~# curl 10.0.1.1
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><html>
<title>Directory listing for /</title>
<body>
<h2>Directory listing for /</h2>
<hr>
<ul>
<li><a href=".bash_history">.bash_history</a>
<li><a href=".bash_logout">.bash_logout</a>
<li><a href=".bashrc">.bashrc</a>
<li><a href=".cache/">.cache/</a>
<li><a href=".config/">.config/</a>
<li><a href=".dbus/">.dbus/</a>
<li><a href=".eclipse/">.eclipse/</a>
<li><a href=".filezilla/">.filezilla/</a>
<li><a href=".fontconfig/">.fontco
<li><a href=".gconf/">.gconf/</a>
```

mininet@mininet-vm: ~/pox

```
mininet@mininet-vm:~/pox$ ./pox.py log.level --DEBUG misc.ip_loadbalancer --ip=10.0.1.1 --servers=10.0.0.1,10.0.0.2
POX 0.1.0 (betta) / Copyright 2011-2013 James McCauley, et al.
DEBUG:core:POX 0.1.0 (betta) going up...
DEBUG:core:Running on CPython (2.7.3/Sep 26 2013 20:08:41)
DEBUG:core:Platform is Linux-3.2.0-49-generic-i686-with-Ubuntu-12.04-precise
INFO:core:POX 0.1.0 (betta) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[None 1] closed
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
INFO:iplb:IP Load Balancer Ready.
INFO:iplb:Load Balancing on [00-00-00-00-00-01 2]
INFO:iplb.00-00-00-00-00-01:Server 10.0.0.1 up
INFO:iplb.00-00-00-00-00-01:Server 10.0.0.2 up
Directing traffic to  10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
Directing traffic to  10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
Directing traffic to  10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
Directing traffic to  10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
```

Node: h4

```
root@mininet-vm:~# curl 10.0.1.1
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"
```

Node: h5

```
root@mininet-vm:~# curl 10.0.1.1
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><html>
<title>Directory listing for /</title>
<body>
```

Node: h6

```
<li><a href="GNUstep/">GNUstep/</a>
<li><a href="jsvm/">jsvm/</a>
<li><a href="mininet/">mininet/</a>
<li><a href="Music/">Music/</a>
<li><a href="mylab/">mylab/</a>
<li><a href="mytest/">mytest/</a>
<li><a href="of-dissector/">of-dissector/</a>
<li><a href="oflops/">oflops/</a>
<li><a href="oftest/">oftest/</a>
<li><a href="openflow/">openflow/</a>
<li><a href="Pictures/">Pictures/</a>
<li><a href="pox/">pox/</a>
<li><a href="Public/">Public/</a>
<li><a href="sflow-rt/">sflow-rt/</a>
<li><a href="svef-1.5/">svef-1.5/</a>
<li><a href="Templates/">Templates/</a>
<li><a href="Videos/">Videos/</a>
<li><a href="vlc/">vlc/</a>
<li><a href="workspace/">workspace/</a>
</ul>
<hr>
</body>
</html>
root@mininet-vm:~#
```

Put this file, ip_loadbalancer.py, under /pox/pox/misc

```
# Copyright 2013 James McCauley
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may 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.
"""
This component (which started in the carp branch) is a simple TCP load balancer.

./pox.py misc.ip_loadbalancer --ip=<Service IP> --servers=<Server1 IP>,<Server2 IP>,...
Give it a service_ip and a list of server IP addresses.  New TCP flows to the service IP will be randomly redirected to one of the server IPs.

Servers are periodically probed to see if they're alive by sending them ARPs.
"""

"""
A very sloppy IP load balancer.

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

Please submit improvements. :)
"""
```

```
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 * 5
selected_server=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 witch 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 = []):
    self.service_ip = IPAddr(service_ip)
    self.servers = [IPAddr(a) for a in servers]
    self.con = connection
    #self.mac = self.con.eth_addr
    self.mac=EthAddr("00:00:00:11:22:33")
    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

# 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 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
    """
    #print "send an arp to 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)
```

```python
@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, round robin based
  """
  global selected_server
  #print selected_server, len(self.live_servers)
  a=self.live_servers.keys()
  if selected_server==len(self.live_servers):
    selected_server=0
  b=a[selected_server]
  selected_server+=1
  return b
  #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:
            #print "server:", arpp.hwsrc, "is still up"
            # 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
  #print "ipp.dstip == self.service_ip ", self.service_ip

  # 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)
    print "Directing traffic to ", server
    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]
  #print mac,port,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 misc.arp_responder import launch as arp_launch
  arp_launch(eat_packets=False,**{str(ip):True})
  import logging
  logging.getLogger("misc.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)
```

References

- Run a simple web server and client, http://mininet.org/walkthrough/#run-a-simple-web-server-and-client
- ip_loadbalancer.py, https://gitshell.com/warcy/pox_SRTP/blob/master/pox/misc/ip_loadbalancer.py