

---

# Assignment 1

---

## Step 0

Before proceeding with the exercise please implement the steps **1 to 4** from the OpenFlow tutorial [3]. This will help you deal with the tutorial pre-requisites, install required software, download and setup the Mininet VM and learn the development and debugging tools, such as wireshark, dpctl, etc. **This is very important to continue.**

## Introduction

In this exercise, you will learn about the open-source OpenFlow controller POX. You will learn how to write network applications, i.e., Hub and Layer 2 MAC Learning etc., on POX and run them on a virtual network based on Mininet. Hub and MAC learning examples will only serve for getting familiar with POX: the final objective of the exercise is to write yourself a load balancer application which uses an OpenFlow switch to balance load stemming from a set of clients towards a set of servers (4 clients-4 servers in particular). More details on creating and submitting the code will be provided later on in the instructions. So, make sure that you follow each step carefully.

**(Note: you can skip this section and start directly with the assignment at page 7, if you feel confident and are already familiar with POX and its basic functions)**

## Getting Familiar with POX

### Invoking the hub application

POX is a Python based SDN controller platform geared towards research and education. **Currently it supports OpenFlow v1.0 [7] and this is the version that will be used for this exercise.** For more details on POX, see POX wiki [4]. To start, navigate to the ~/pox directory in your Mininet VM, pull the latest code and checkout the *carp* branch:

```
$ cd ~/pox
$ git pull
$ git checkout carp
$ cd ~
```

We are going to use POX as the exercise's OpenFlow controller. We will not use the reference controller anymore, which is the default controller that Mininet uses during its simulation. So, make sure that it's not running in the background:

```
$ ps -A | grep controller
```

If so, you should kill it; either press Ctrl-C in the window running the controller program, or from another SSH window:

```
$ sudo killall controller
```

In case a running controller is not killed because you have invoked its CLI, just press Ctrl-D in the CLI window. You should also run:

```
sudo mn -c
```

and restart Mininet to make sure that everything is clean and using the faster kernel switch. From your Mininet VM run:

```
$ sudo mn --topo single,8 --mac --switch ovsk --controller remote
```

This will start the Mininet network emulator that we will use for experimentation. Essentially, this command tells Mininet to start up an 8-host network with a single switch (running Open vSwitch [2]), set the MAC address of each host equal to its IP for readability, and point to a remote controller, which by default runs in localhost. Here is what Mininet just did:

- Created 8 virtual hosts, each with a separate IP address.
- Created a single OpenFlow software switch with 8 ports, running in the kernel space of Mininet.
- Connected each virtual host to the switch with a virtual ethernet cable.
- Set the MAC address of each host equal to its IP address.
- Configured the OpenFlow switch to connect to a remote controller (running in localhost in our case).

The POX controller comes pre-installed with the provided VM image. Now, from another SSH window connected to the same Mininet vm, run the basic hub example, after going to the ~/pox directory:

```
$ cd ~/pox
$ ./pox.py log.level --DEBUG forwarding.hub
```

This tells POX to enable verbose logging (for debugging) and to start the hub component. The switch of your topology might take a little bit of time to contact POX. This is because, when an OpenFlow switch loses its connection to a controller, it will generally increase the period between which it attempts to contact the controller again, up to a maximum of 15 seconds. Since the OpenFlow switch has not connected yet, this delay may be anything between 0 and 15 seconds. If this is too long to wait, the switch can be configured to wait no more than N seconds using the max-backoff parameter. Alternately, you exit Mininet to remove the switch(es), start the controller, and then start Mininet to immediately connect. Wait until the application indicates that the OpenFlow switch has connected to POX. When the switch connects, POX will print something like this:

```
POX 0.0.0 / Copyright 2011 James McCauley
INFO:forwarding.hub:Hub running.
DEBUG:core:POX 0.0.0 going up...
DEBUG:core:Running on CPython (2.7.3/Sep 26 2012 21:51:14)
INFO:core:POX 0.0.0 is up.
This program comes with ABSOLUTELY NO WARRANTY. This program is free software,
```

```
and you are welcome to redistribute it under certain conditions.
Type 'help(pox.license)' for details.
DEBUG:openflow.of_01:Listening for connections on 0.0.0.0:6633
Ready.
POX> INFO:openflow.of_01:[Con 1/1] Connected to 00-00-00-00-00-01
INFO:forwarding.hub:Hubifying 00-00-00-00-00-01
```

## Verifying hub behavior with tcpdump

Now verify that hosts can ping each other, and that all hosts see the exact same traffic - the behavior of a hub. To do this, we'll create xterms for each host and view the traffic in each. In the Mininet console, start up three xterms:

```
mininet> xterm h1 h2 h3
```

Arrange each xterm so that they're all on the screen at once. This may require reducing the height to fit a cramped laptop screen. In the xterms for h2 and h3, run `tcpdump`, a utility to print the packets seen by a host:

```
# tcpdump -XX -n -i h2-eth0
```

and respectively:

```
# tcpdump -XX -n -i h3-eth0
```

In the xterm for h1, send a ping:

```
# ping -c1 10.0.0.2
```

The ping packets are now going through the switch, which then floods them out of all its ports except the sending one. You should see identical ARP and ICMP packets corresponding to the ping in both xterms running `tcpdump`. This is how a hub works; it sends all packets to every port on the network. All the hub flow rules have been installed by the controller on startup. Now, see what happens when a non-existent host doesn't reply. From h1 xterm:

```
# ping -c1 10.0.0.10
```

You should see three unanswered ARP requests in the xterms running `tcpdump`. You can use this technique to debug your code later (essentially, three unanswered ARP requests is a signal that you might be accidentally dropping packets). You can close the xterms now.

## Taking a closer look at POX-based network programming

First, let's look at the hub code:

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

log = core.getLogger()

def _handle_ConnectionUp (event):
    msg = of.ofp_flow_mod()
    msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
    event.connection.send(msg)
    log.info('Hubifying %s', dpidToStr(event.dpid))
```

```
def launch ():
    core.openflow.addListenerByName("ConnectionUp", _handle_ConnectionUp)

    log.info("Hub running.")
```

The following API primitives are useful for building such network applications on POX. All these primitives can be accessed after importing the openflow library of POX:

```
import pox.openflow.libopenflow_01 as of
```

Some examples are the following:

- **connection.send( ... )** function sends an OpenFlow message to a switch. When a connection to a switch starts, a ConnectionUp event is fired. The above code invokes a `_handle_ConnectionUp ()` function that implements (for example) the hub logic.
- **ofp\_action\_output** class: This is an action for use with `ofp_packet_out` and `ofp_flow_mod`. It specifies a switch port that you wish to send the packet out of. It can also take various "special" port numbers. An example of this, as shown in the hub example, would be `OFPP_FLOOD` which sends the packet out all ports except the one the packet originally arrived on.

Example: Create an output action that would send packets to all ports:

```
out_action = of.ofp_action_output(port = of.OFPP_FLOOD)
```

- **ofp\_match** class (not used in the code above but might be useful in the exercise): Objects of this class describe packet header fields and an input port to match on. All fields are optional – items that are not specified are "wildcarded" and will match on anything. Some notable fields of `ofp_match` objects are:

- **dl\_src** - The data link layer (MAC) source address
- **dl\_dst** - The data link layer (MAC) destination address
- **in\_port** - The packet input switch port

Example: Create a match that matches packets arriving on port 3:

```
match = of.ofp_match()
match.in_port = 3
```

- **ofp\_packet\_out** OpenFlow message (not used in the code above but might be useful in the exercise): The `ofp_packet_out` message instructs a switch to send a packet. The packet might be one constructed at the controller, or it might be one that the switch received, buffered, and forwarded to the controller (and is now referenced by a `buffer_id`).

Notable fields are:

- **buffer\_id** - The `buffer_id` of a buffer you wish to send. Do not set if you are sending a constructed packet.
- **data** - Raw bytes you wish the switch to send. Do not set if you are sending a buffered packet.
- **actions** - A list of actions to apply (for this tutorial, this is just a single `ofp_action_output` action).

- **in\_port**- The port number this packet initially arrived on if you are sending by buffer\_id, otherwise OFPP\_NONE.
- **ofp\_flow\_mod** OpenFlow message: This instructs a switch to install a flow table entry. Flow table entries match some fields of incoming packets, and execute some list of actions on matching packets. The actions are the same as for ofp\_packet\_out, mentioned above (and, again, for the tutorial all you need is the simple ofp\_action\_output action). The match is described by an ofp\_match object. Notable fields are:

- **idle\_timeout** - Number of idle seconds before the flow entry is removed. Defaults to no idle timeout.
- **hard\_timeout** - Number of seconds before the flow entry is removed. Defaults to no timeout.
- **actions** - A list of actions to perform on matching packets (e.g., ofp\_action\_output).
- **priority** - When using non-exact (wildcarded) matches, this specifies the priority for overlapping matches. Higher values are higher priority. Not important for exact or non-overlapping entries.
- **buffer\_id** - The buffer\_id of a buffer to apply the actions to immediately. Leave unspecified for none.
- **in\_port** - If using a buffer\_id, this is the associated input port.
- **match** - An ofp\_match object. By default, this matches everything, so you should probably set some of its fields!

Example: Create a flow\_mod that sends packets from port 3 out of port 4:

```
fm = of.ofp_flow_mod()
fm.match.in_port = 3
fm.actions.append(of.ofp_action_output(port = 4))
```

## Invoking the learning switch application and verifying behavior with tcpdump

This time, let's verify that hosts can ping each other when the controller is behaving like a Layer 2 learning switch. Kill the POX controller by pressing Ctrl-C in the window running the controller program. In case the running controller is not killed because you have invoked its CLI, just press Ctrl-D in the CLI window. Now run the l2\_learning example:

```
$ ./pox.py log.level --DEBUG forwarding.l2_learning
```

Like before, we'll create xterms for each host and view the traffic in each. In the Mininet console, start up three xterms:

```
mininet> xterm h1 h2 h3
```

Arrange each xterm so that they're all on the screen at once. This may require reducing the height to fit a cramped laptop screen. In the xterms for h2 and h3, run tcpdump, a utility to print the packets seen by a host:

```
# tcpdump -XX -n -i h2-eth0
```

and respectively:

```
# tcpdump -XX -n -i h3-eth0
```

In the xterm for h1, send a ping:

```
# ping -c1 10.0.0.2
```

Here, the switch examines each packet and learns the source-port mapping. Thereafter, the source MAC address will be associated with the port. If the destination of the packet is already associated with some port, the packet will be sent to the given port, else it will be flooded on all ports of the switch. You can close the xterms now. The code for l2\_learning application is provided under `~/pox/pox/forwarding`. **The learning switch module will be the main example that will help you solve the current exercise.**

# Assignment: Build a simple load balancer

## Overview of the Setup

The network you'll use in this exercise includes 8 hosts and a switch with an OpenFlow controller (POX). The first 4 hosts (h1 to h4) act as clients for a service offered by the hosts h5 to h8, which act as servers. The switch acts as a load balancer, and balances the flows from the clients towards the servers. The clients are addressing the public IP of the service, and the switch acts as a transparent proxy between the clients and the actual server. The setup is depicted in Fig. 1.

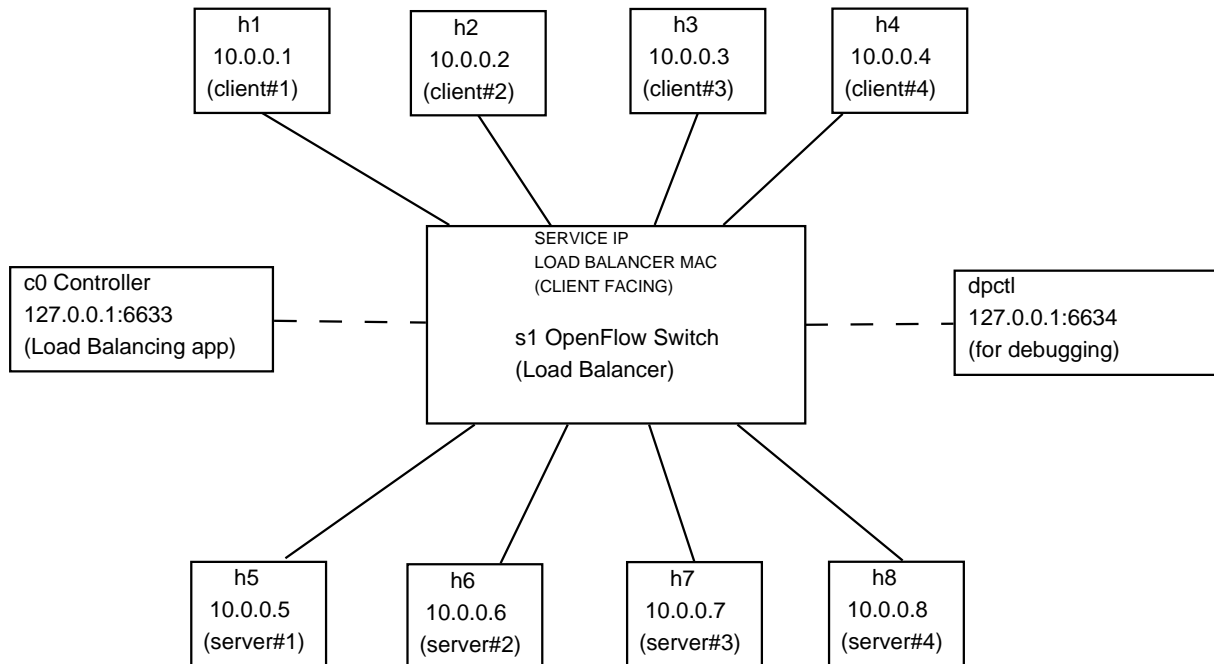


Figure 1: Load Balancer setup

## What the load balancer should do

- The switch should preemptively ask for the MAC addresses of all the servers with crafted ARP requests, in order to associate these MAC addresses and the corresponding switch ports with the real IP addresses of the servers. This query should be performed upon the connection establishment of the controller to the switch in order to avoid having client flows waiting to be forwarded to the correct server. The ARP replies by the servers will be handled as part of the packet-in handler (see code skeleton later).
- Answer to ARP requests from the clients searching the MAC of the service IP address. The switch should proxy ARP replies that answer to the clients' requests with a fake MAC that is associated with the load balancer (you can use "0A:00:00:00:00:01" for simplicity). It is useful to store the information contained in each ARP request (source MAC address of client, input port of ARP request packet). In this way, when the load balancer needs later to direct flows towards the clients, it will know their MAC addresses and ports to output the packets.
- Answer to ARP requests from the servers searching the MAC addresses of clients. The switch should proxy ARP replies that answer with the fake MAC that is associated with the load balancer. At this point you should already know the MAC of the client, since it has previously requested the MAC address of the service (see previous step).

- Redirect flows from the clients towards the servers using the following load balancing mechanism: **for each new IP flow from a client, select a server at random and direct the flow to this server**. Of course, the server should see packets with their MAC address changed to the MAC of the load balancer, but with the source client IP intact. The destination IP address should also be rewritten to the one of the destination server. Be careful: the redirection should only happen for flows that stem from client IPs (i.e., non-server IPs) and which are directed to the service IP.
- Direct flows from the servers to the clients. This should occur after rewriting the source IP address to the one of the service and the source MAC address to the load balancer fake MAC. In this way, the clients do not see any redirection happening, and they believe that all their communication takes place between their machines and the service IP (the load balancing mechanism is transparent).
- There is no need to handle forwarding between the servers themselves or between the clients themselves; in this exercise we are interested in the load-balancing behaviour and the traffic that flows between clients and servers.

For the exercise you are requested not to use microflows (i.e. flows that match the packets exactly on all fields), but flows that match on IP addresses (while their dl\_type matches to the 0x800 value, i.e. the IP protocol). All the flows should expire after a 10 sec idle timeout. Default hard timeouts should not be altered. Packet-ins (corresponding to new flows) should be properly handled by your application. You should only consider ARP and IP protocol packet types. For your convenience, a code skeleton with some initial functionality is provided below (between “CODE START” and “CODE END”):



```

#----- CODE START -----
from pox.core import core
from pox.openflow import *
import pox.openflow.libopenflow_01 as of
from pox.lib.packet.arp import arp
from pox.lib.packet.ipv4 import ipv4
from pox.lib.addresses import EthAddr, IPAddr
log = core.getLogger()
import time
import random

class SimpleLoadBalancer(object):
    def __init__(self, service_ip, server_ips = []): #initialize
        core.openflow.addListeners(self)
        #write your code here!!!

    def _handle_ConnectionUp(self, event): #new switch connection
        self.lb_mac = EthAddr("0A:00:00:00:00:01") #fake mac of load balancer
        self.connection = event.connection
        #write your code here!!!

    def update_lb_mapping(self, client_ip): #update load balancing mapping
        #write your code here!!!

    def send_proxied_arp_reply(self, packet, connection, outport, requested_mac):
        #write your code here!!!

    def send_proxied_arp_request(self, connection, ip):
        #write your code here!!!

    def install_flow_rule_client_to_server(self, connection, outport, client_ip,
                                           server_ip, buffer_id=of.NO_BUFFER):
        #write your code here!!!

    def install_flow_rule_server_to_client(self, connection, outport, server_ip,
                                           client_ip, buffer_id=of.NO_BUFFER):
        #write your code here!!!

    def _handle_PacketIn(self, event):
        packet = event.parsed
        connection = event.connection
        inport = event.port
        if packet.type == packet.ARP_TYPE:
            #write your code here!!!

        elif packet.type == packet.IP_TYPE:
            #write your code here!!!

        else:
            log.info("Unknown Packet type: %s" % packet.type)
            return
        return

```

```
#launch application with following arguments:
#ip: public service ip, servers: ip addresses of servers (in string format)
def launch(ip, servers):
    log.info("Loading Simple Load Balancer module")
    server_ips = servers.replace(","," ").split()
    server_ips = [IPAddr(x) for x in server_ips]
    service_ip = IPAddr(ip)
    core.registerNew(SimpleLoadBalancer, service_ip, server_ips)

#----- CODE END -----
```

The function prototypes above should be implemented by you (see “write your code here” instructions). Make sure that you log interesting messages that may help you debug your code.

## Running your application

Write your application as a .py file under ~/pox/ext (name it SimpleLoadBalancer.py) Then:

```
cd ~/pox
```

Your code should run with the following arguments:

- ip: the public service ip (could be anything you want, except for the already assigned addresses 10.0.0.1-10.0.0.8)
- servers: a list of the server ip addresses (here these are the 10.0.0.5-10.0.0.8)

Therefore, your application should be invoked as follows:

```
./pox.py SimpleLoadBalancer --ip=10.1.2.3
--servers=10.0.0.5,10.0.0.6,10.0.0.7,10.0.0.8
```

Now, run Mininet with the topology described above:

```
sudo mn --topo single,8 --controller remote --mac --switch ovsk
```

Try to ping from the clients (host h1-h4) to the service IP address. Do you achieve load balancing as expected?

**Please make sure that you have already studied the POX wiki [4], the Mininet documentation [1] and the OpenFlow tutorial [3] thoroughly.**

**Good luck!**

## Acknowledgments

**This assignment is based on material from a similar course at ETH Zurich [6].** The initial instructions for having POX up and running and testing it have been adapted by the “Software Defined Networking” course archive on [www.coursera.org](http://www.coursera.org) and the OpenFlow tutorial at [3].

## References

- [1] Mininet official website. <http://mininet.org/>.
- [2] Open vSwitch official website. <http://www.openvswitch.org/>.
- [3] OpenFlow Tutorial. [http://archive.openflow.org/wk/index.php/OpenFlow\\_Tutorial](http://archive.openflow.org/wk/index.php/OpenFlow_Tutorial).
- [4] POX Wiki. <https://openflow.stanford.edu/display/ONL/POX+Wiki>.
- [5] Turnin User Guide. <http://www.csd.uoc.gr/services/useful-info/use-the-turnin.html>.
- [6] ETH Zurich. Advanced Topics in Communications Networks HS 2014: Software-Defined Networking. <http://www.csg.ethz.ch/education/lectures/ATCN/hs2014>.
- [7] Open Networking Foundation. OpenFlow Switch Specification, v1.0.0. <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf>.