Note: For this version, details about how to sign in and use POWDER (which is a barebones Linux VM that can be easily modified and reset but for my class had a specific project name I want to keep private), as well as the class's specific details, submission details (section 5), and few people's names have been removed.

1 Overview

The primary goal of this assignment is to become familiar with software defined networking (SDN) concepts by developing a simple OpenFlow application using the POX SDN framework [1]. Your SDN application will realize a simple "virtual IP load balancing switch". (I.e., a switch that will map, in round-robin fashion, a virtual IP address to a set of real IP addresses associated with servers "behind" it.) You will make use of two testbed environments for this assignment. First, you will be using a virtual networking environment called Mininet[2]. Mininet allows the emulation of arbitrary network topologies inside a single physical or virtual machine. Of specific interest for our purposes is the fact that Mininet can instantiate OpenFlow capable switches in this emulated environment. Second, to simplify the task of setting up Mininet, we will make use of the POWDER testbed environment[3]. Specifically, we have created a virtual machine (VM) profile in POWDER, which you can instantiate to install the Mininet environment and the POX framework. Each student can use this profile to easily instantiated (*their own*) VM for the assignment.

2 Setup

Figure 1 depicts the setup you will use for developing your application. (And the setup that will be used for evaluating your application.)

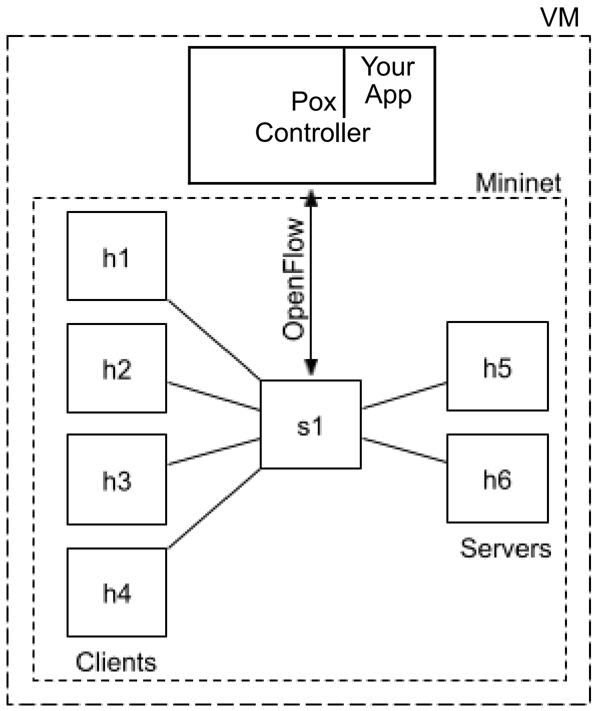
Specifically: You will instantiate a VM and install the Mininet and POX frameworks. Once your VM is operational and you have Mininet and POX installed, you can instantiate a Mininet instance using the *mn* command shown in the figure. As shown, this command instantiates a single switch (*s*1) and six hosts (*h*1 through *h*6). Once you have written your SDN application, you will use the shown *pox* commandline to instantiate the POX controller and your application.

3 SDN Application

The purpose of your SDN application is to realize a simple "virtual IP load balancing switch". I.e., a switch that will map, in round-robin fashion, a virtual IP address to a set of real IP addresses associated with servers "behind" it.

With reference to Figure 1, for the purposes of this assignment we will assume that switch *s*1 is fronting two servers, *h*5 and *h*6. We will treat the four other hosts, i.e., *h*1 through *h*4 as clients. When one of the client hosts sends traffic to a known virtual IP address, i.e., an address that is not actually associated with any of the host interfaces in the system, your application will map the virtual IP address to one of the server IP addresses in round-robin fashion. Similarly, for traffic returned by the server, the switch will perform the mapping in reverse. I.e., from the perspective of the client hosts, they will all be communicating with a

[1] https://noxrepo.github.io/pox-doc/html/
[2] https://mininet.org/walkthrough/
[3] https://powderwireless.net/



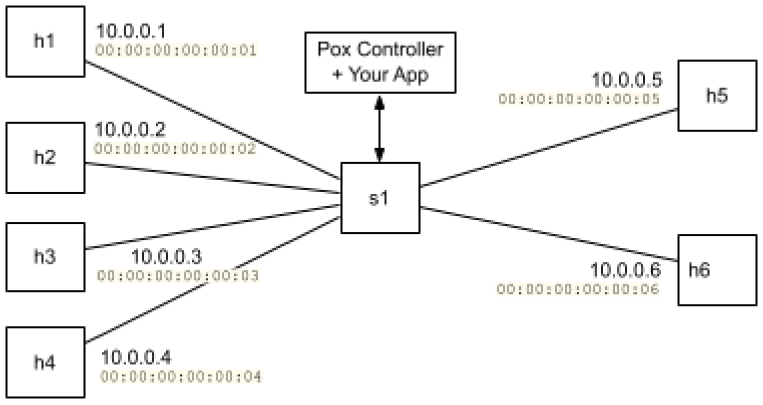
```
// In first shell, start up Mininet:
sudo mn --topo single,6 --mac --controller remote,ip=127.0.0.1,port=6633 --switch ovsk,protocols=OpenFlow10

// In second shell, run Pox with Your App,
// (assuming your app is your_app.py and copied into pox/ext):
cd pox && python pox.py openflow.of_01 --port=6633 your_app
```

Figure 1: VM and Mininet setup

single server (associated with the virtual IP address). In reality, however, each client will be communicating with one of the two "real" servers behind the switch.

To simplify your SDN application you only need to support ICMP traffic, i.e., *ping*. Further, we will make use of the fact that communication between two hosts in the same subnet is preceded by an ARP request (to map the IP address to a MAC address) to trigger assigning the (new) traffic flow (which is to follow the ARP request) to the "next" server *and* to set up the appropriate forwarding rules in the switch. I.e., each ARP request from a client node should result in mapping the virtual IP address to the "next" real IP address. (Or in the case with only two real servers, the mapping would alternate between the two servers.)

This process and sequence of events are depicted for an example exchange in Figure 2. Assume host *h*1 issues a ping to the virtual IP address 10.0.0.10 (step 1). The networking stack on host *h*1 will issue an ARP request to map the virtual IP address (10.0.0.10) to a MAC address (step 2). Your SDN application will intercept this ARP request, make a selection of which real server to use and send an ARP response with the appropriate MAC address back to the requesting host (step 3). E.g., this example assumes that host *h*5 was selected and the MAC address associated with it is returned in the response. In addition your application will push the appropriate forwarding rules into the switch to facilitate mapping between the virtual and real IP addresses (step 4). Specifically, as shown in the figure, you will need to set up a rule for traffic from *h*1 to *h*5 as well as in the reverse direction. In the example shown, for the *h*1 to *h*5 direction, you will match on



1: h1 ping 10.0.0.10
// user typed command

2: ARP Request who-has 10.0.0.10 tell 10.0.0.1
// h1's network stack

3: ARP Reply 10.0.0.10 is-at 00:00:00:00:00:05
// Your app

4: Push OF rules on s1:
- h1 to h5
match:
inport=h1-port, dst-ip=10.0.0.10
action:
set: dst-ip=10.0.0.5
output: h5-port
- h5 to h1
match:
inport=h5-port, src-ip=10.0.0.5, dst-ip=10.0.0.1
action:
set: src-ip=10.0.0.10
output: h1-port
// Your app

5: ICMP echo request
// h1's network stack

6: ICMP echo response
// h5's network stack

the switch port associated with host *h*1 and the virtual destination IP address, and then set the destination address to be that of *h*5 and forward the packet out on the port associated with *h*5. A similar rule is pushed for traffic in the reverse direction. At this point ICMP echo request and response traffic will be able to flow between hosts *h*1 and *h*5 (steps 5 and 6).

Note that for *h*5 to be able to communicate with *h*1 (i.e., respond to the ping), it will issue a similar ARP request in the opposite direction, (i.e., to find a MAC address associated with $h1's$ IP address). Your solution will have to accomodate this for the communication to succeed. A possible solution for this would be for your application to also intercept ARP requests in this direction, and respond with the appropriate MAC address.

Figure 3 shows the resulting interaction assuming that hosts *h*1 to *h*4 issued ping commands to the virtual IP address in sequence. I.e., *h*1 is mapped to *h*5, *h*2 is mapped to *h*6, *h*3 is again mapped to *h*5 etc. Such a sequence of ping commands will be used to verify the functionality of your application during evaluation.

4 Approach

This is a non-trivial assignment which will acquaint you with a variety of concepts, several of which might be new. It is strongly recommended that you start early and follow a staged approach to work through all the needed materials. Suggested/required approach is outlined in the following sections.

4.1 Become familiar with Mininet, OpenFlow and POX

Read through the complete assignment so that you know what is expected, e.g., in terms of things to capture for your progress report.

- **Set up environment**:

  Follow the steps described in Section 6 to sign up for a POWDER account and instantiate the VM profile. Follow the provided instructions to install Mininet and POX.

  VERY IMPORTANT: By default your VM will be terminated after 16 hours. The best workflow for this environment is as follow:

  - Keep your application code in a git repository of your choice.
  - When you want to work on your assignment, instantiate the VM, install Mininet and POX and clone your repository onto the VM.
  - Remember to regularly push any code changes up to you repository. (Anything "left" on the VM will be **lost** when the VM is terminated.)
  - Note: It is possible to request modest extensions to the VM termination time. Use this only when you are actively doing development.

- **Understand Mininet**:

  Work through the "Mininet Walkthrough"[4]. Main objectives are to get a practical understanding of Mininet, its namespace isolation, how it fits into the SDN VM, how to use Mininet commandline options, how to script Mininet commands, how to run an external controller (e.g., one running on the VM itself), how to clean things up when something goes wrong, how to monitor both data plane and control plane interaction using Wireshark etc.

- **Understand OpenFlow**:

  Page through the OpenFlow Switch Specification[5]. Most relevant: Sections 1 through 4.1.1. Experiment with the ovs-ofctl[6] commandline options to manually set up OpenFlow rules[7].

[4]https://mininet.org/walkthrough/
[5]https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.0.0.pdf
[6]https://docs.openvswitch.org/en/latest/ref/ [7]http://trainer.edu.mirantis.com/SDN50/ovs.html

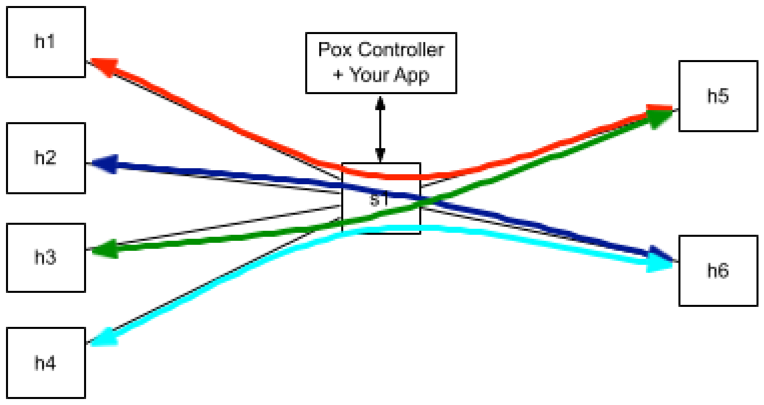


Figure 3: Operation: h1 through h4 ping “virtual IP” (10.0.0.10)

- **Understand POX and POX applications**:

  Page through the POX documentation[8] and work through relevant sections in more detail: How to invoke the POX controller; Work through the source code for some of the built-in POX components; How to add your own code to the POX framework.

- **Submit first progress report**. See details of what to submit in Section 5.

4.2 Core assignment execution

- A reasonable approach might be to first figure out how to set the necessary flows without using a virtual IP address and using the OpenFlow commandline tool (ovs-ofctl). Then do the same with a POX application. Then add the ARP intercept functionality. Then update the rules to make use of a virtual IP address and perform the necessary IP address mapping.

5 Signing up with POWDER and setting up your VM + Mininet and POX

- Once your node is ready: On the ”List View” tab, click on the gear icon in the node row and select “Open VNC window” to access a Linux desktop on your VM.
- Install Mininet on your VM:

  ```
  sudo apt-get install mininet
  ```

- Install POX on your VM:

  ```
  git clone http://github.com/noxrepo/pox
  ```

- At this point you should be ready to start exploring MiniNet, OpenFlow and POX!

6 Additional notes

Additional notes provided by TA [redacted] who updated the assignment to use the POX environment.

6.1 Useful resources

- POX Docs:

  https://noxrepo.github.io/pox-doc/html/

  This is just the main page, you can find anything you need to know about POX here.

- arp_responder.py:

  https://github.com/noxrepo/pox/blob/gar-experimental/pox/proto/arp_responder.py

  This was very helpful in learning how to handle arp requests. The _handle_PacketIn method was especially helpful to look at. (This program builds up a custom arp table and does a few other things that we don't need for our switch)

- OpenFlow Switch Specification:

  https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.0.0.pdf

  The parts of this that I found most useful were sections 1-4.1.1

- Mininet Walkthrough:

  https://mininet.org/walkthrough/

  Understanding Mininet is important in allowing you to see information about the environment and understanding what is happening

- Stanford OpenFlowTutorial:

  https://web.archive.org/web/20130116072451/http://www.stanford.edu/~brandonh/ONS/OpenFlowTutorial_ONS_Heller.pdf

  This is very old but it gives a nice overview of SDN and how OpenFlow generally works and what some of the limitations are. It can also be easier to read as it is slides with big points, rather than a textbook with massive blocks of text. I don't think any of the information outside of slides 14-78 is very useful (unfortunately the walkthrough isn't still active).

6.2 Debugging/Testing commands

In host's xterm:

```
ping -c <packet count> 10.0.0.10              # send packets to the switch
tcpdump                                       # show all packets that come across it
tcpdump host 10.0.0.10 and <arp or icmp>      # watch specifically for packets to the switch that are of arp    # show arp
table
ip neigh flush all                            # clear arp table
arping -I h1-eth0 10.0.0.10                   # continuously sends out arp requests
```

In switch's xterm:

```
tcpdump -i s1-eth1                            # watch packets that come through
ovs-ofctl dump-flows s1                       # show the flows that have been set up so far
```

In other xterm:

```
sudo mn -c                                    # clear mininet environment
```

6.3 Pitfalls/Notes

- Sometimes when opening the VNC window for POWDER it just gives a white screen that doesn't respond. Closing and reopening resolved this.
- Mixing up ip and mac addresses
  - When setting src and dst on packets, its important to keep protosrc = ip and hwsrc = mac. I mixed those up a few times and it made the program not work
  - When making flow modifications (like msg.actions.append(of.ofp_action_nw_addr.set_dst(server_ip))) its important to keep nw_addr and dl_addr seperate, as nw is for ip and dl is for mac
  - This also applies to simple things like using a src when you should be using a dst
- ARP replies
  - Only verifying that an incoming packet is an arp packet is not enough, you need to check if it is a reply or a request
- Matching
  - If you have too many rules you might miss out on packets, but not enough and you will take in packets not meant for you to handle. What I ended up matching on were these things:
    - msg.match.in_port
    - msg.match.dl_type
    - msg.match.nw_dst
    - msg.match.nw_src (only in the server to client message handler)
- Adding flow rules
  - When adding rules, set all the addresses before the output rule or it could go out without the changes
- Set specific ports rather than just relying on flooding (in cases where you know where the packet should go). I understand that it is a bad idea because of scaling issues causing congestion, but it also prevented my code from working right, I don't really understand why (it seems like it still allows the message to get where it needs to go, maybe there is something that happened in the backend, like packets getting congested or something even though it is pretty small scale).
- Add lots of logging, I couldn't figure out how to SSH into the environment so I kind of relied on changing something, commiting it to my github, then cloning it into the environment (I made commands that made it pretty easy and fast). This means that debugging becomes a lot harder though. I started out with not very much logging and ended up with a decent amount of it.
- When to use event.ofp or event.parsed
  - event.ofp just gives the raw packet, while event.parsed (shockingly) gives a parsed version
  - If you actually want to look into the packet and analyze it or really do much other than forward it, use parsed. But just for forwarding you can use ofp
- IPAddr data type
  - ip addresses can't be passed as strings or anything, I just put them into constants at the top so I didn't need to worry about it too much but I could see this being confusing after PA1 and putting things into bytestrings as just the normal ip address.