Assignment 3: Exploring Software-Defined Networking

A. Summary

In this assignment, we will play with software-defined (SDN) networking techniques and get familiar with the main concepts of SDN. We will use Mininet to create an emulation platform with the software-based OpenFlow switches (i.e., Open vSwitch). On top of this emulated SDN platform, POX is used as the SDN controller. We need to correctly deploy the emulation platform and SDN controller, be familiar with the operations, program SDN controllers, and finish the required tasks. After these tasks, we should have a concrete idea on how the SDN controller communicates with OpenFlow switches.

B. Get to Know the Tools

a. Experimental Environment

In this assignment, we will use the GCP VM. Please make sure the Ubuntu version is **20.04 LTS image** (that way, the TAs can help you with the configuration if you need help).

Please refer back to your Assignment 1 for the Docker installation on the GCP.
Note: please switch to the "root" user for this assignment (for example, use "sudo su" command), as we notice that mininet container can only work properly under the root user.

b. Mininet Installation

Mininet is a great way to develop, share, and experiment with OpenFlow and Software-Defined Networking systems. To not mess our native network (e.g., your GCP instance), we will run all experiments using a docker instance.
We can pull Mininet docker and start it simply by:

```
$ docker pull iwaseyusuke/mininet
$ docker run -it --name=mininet --privileged -v
/tmp/.X11-unix:/tmp/.X11-unix -v /lib/modules:/lib/modules
iwaseyusuke/mininet
```

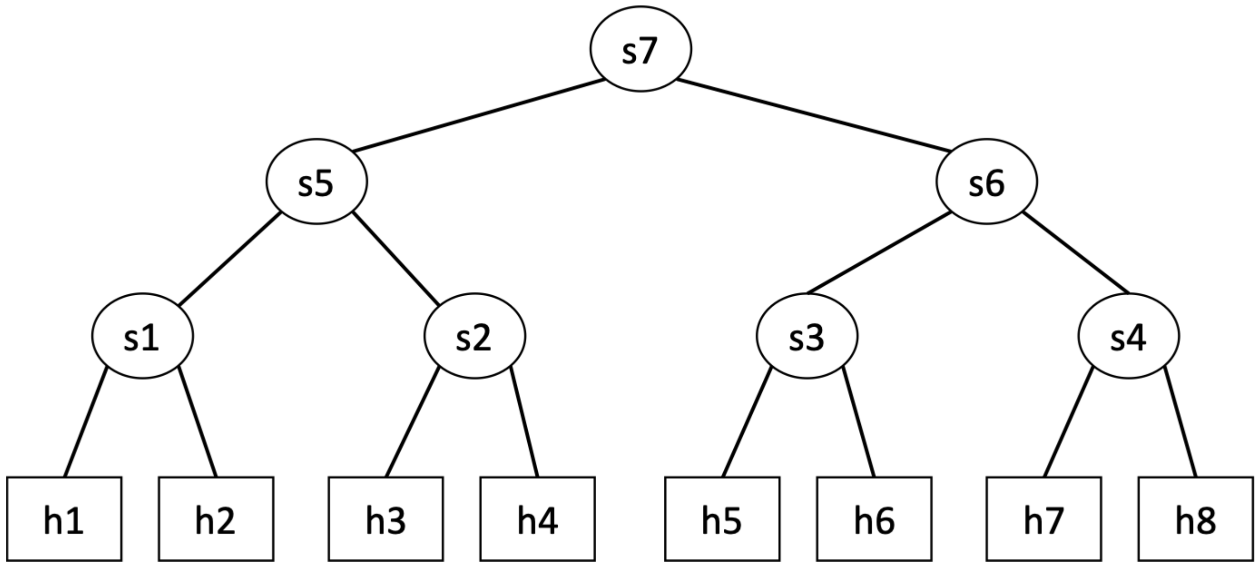

Figure 1: A binary tree network of depth 3.

We may need to practice and get familiar with Mininet by following this document (skip the Wireshark part). Particularly, we need to understand how to define a custom topology using Python scripts (e.g., “custom/topo-2sw- 2host.py” in this document). After walking through the main operations of Mininet, please finish Task I.

c. Task I: Define a Custom Topology

We know that data center networks typically have a tree-like topology. End-hosts connect to top-of-rack switches, which form the leaves of the tree; one or more core switches form the root; and one or more layers of aggregation switches from the middle of the tree. For example, a binary tree network of depth 3 looks like in Figure1. **Task I** asks you to develop a Python script that can populate such a binary tree topology in the Mininet. After completing the Python script, you should run the following commands to verify it — “h1” should reach “h8” with ping.

```
$ mn --custom binary_tree.py --topo mytopo
$ mininet> h1 ping h8
```

d. SDN Controller (POX) Installation

POX is an open source development platform for Python-based software-defined networking (SDN) control applications. In this assignment, we use POX as the OpenFlow controller [1].

Open a new terminal of the running Mininet docker:

```
$ docker exec -it mininet /bin/bash
```

(A). Download POX

```
$ apt-get update
$ apt-get install git
$ env GIT_SSL_NO_VERIFY=true git clone https://github.com/noxrepo/pox
$ apt-get install python3
```

(B). Start the SDN controller

```
$ cd pox
$ python3 pox.py log.level --DEBUG misc.of_tutorial
```

This loads the controller in ~/pox/pox/misc/of_tutorial.py. This controller acts like a “dumb” switch and floods all packets.

(C). Connect our network topology to the controller

At the same time, create our binary-tree topology (completed in Task I) that connects to the controller using the following command (hint: you need to execute another bash for the same mininet container):

```
$ mn --custom binary_tree.py --controller remote --topo mytopo
```

This connects all software switches (s1 ~ s8) to the controller. Please take a look at the output from the controller console.

e. Task II: Study the “of_tutorial” Controller

We have learned that, for the SDN setup, if there are no forwarding rules for a particular packet, such a packet will be sent to the SDN controller (i.e., “of_tutorial” controller). Please understand this concept by studying the “of_tutorial” controller, and answer the following questions.

---

[1] Notice that OpenDaylight is an enterprise-level OpenFlow controller, newer than POX, which is written in Java. It is way more sophisticated than POX, it might be more challenging to work on it. You are encouraged to explore OpenDaylight.

- Q.1 Draw the function call graph of this controller. For example, once a packet comes to the controller, which function is the first to be called, which one is the second, and so forth?
- Q.2 Have h1 ping h2, and h1 ping h8 for 100 times (e.g., h1 ping -c100 p2). How long does it take (on average) to ping for each case? What is the difference, and why?
- Q.3 Run “iperf h1 h2” and “iperf h1 h8”. What is “iperf” used for? What is the throughput for each case? What is the difference, and why?

- Q.4 Which of the switches observe traffic? Please describe your way for observing such traffic on switches (hint: adding some "print" functions in the "of_tutorial" controller).

f. Resources

Notice that this document only provides a barebone guide to the setup. You are encouraged to look through the following documents:
https://github.com/noxrepo/pox
http://mininet.org/walkthrough/
https://github.com/mininet/openflow-tutorial/wiki

C. Task III: MAC learning controller

In Task II, we have learned a controller that makes switches flood every packet they receive, and thus, in practice, behave like hubs. In this section, we will change this behavior such that the switches will learn the ports packets arrive from, and upon receiving a packet, if they have already seen its destination address, they will know the exact port to forward it on and avoid flooding the network (i.e., MAC learning controller).

Please add the following code into the "of_tutorial" controller(type the code down manually). This realizes the above MAC learning controller. Please change the other function(s) accordingly to make sure the "act_like_switch" function will be invoked in your code, analyze the behaviors, and answer the questions.

```
def act_like_switch (self, packet, packet_in):
    # Learn the port for the source MAC
    # print "Src: ",str(packet.src),":", packet_in.in_port,"Dst:", str(packet.dst)
    if packet.src not in self.mac_to_port:
        print "Learning that " + str(packet.src) + " is attached at port " + str(packet_in.in_port)
        self.mac_to_port[packet.src] = packet_in.in_port


    # if the port associated with the destination MAC of the packet is known:
    if packet.dst in self.mac_to_port:
        # Send packet out the associated port

        print str(packet.dst) + " destination known. only send message to it"
        self.resend_acket(packet_in, self.mac_to_port[packet.dst])
    else:

        # Flood the packet out everything but the input port
        # This part looks familiar, right?
```

```
print str(packet.dst) + " not known, resend to everybody"
self.resend_packet(packet_in, of.OFPP_ALL)
```

- Q.1 Please describe how the above code works, such as how the "MAC to Port" map is established. You could use a 'ping' example to describe the establishment process (e.g., h1 ping h2).
- Q.2 (Please disable your output functions, i.e., print, before doing this experiment) Have h1 ping h2, and h1 ping h8 for 100 times (e.g., h1 ping -c100 p2). How long did it take (on average) to ping for each case? Any difference from Task II (the hub case)?
- Q.3 Run "iperf h1 h2" and "iperf h1 h8". What is the throughput for each case? What is the difference from Task II?

D. Task IV: MAC learning controller with OpenFlow rules

We will try to make the switch a bit smarter. Modify the controller from Section C so that when it learns a mapping, it installs a flow rule on the switch to handle future packets. Thus, the only packets that should arrive at the controller are those that the switch doesn't have a flow entry for. Please realize such a MAC learning controller. The useful piece of code (i.e., how to insert a flow rule to the OpenFlow switch) is shown as follows.

```
# load the flow module
fm = of.ofp_flow_mod()
# fill the match filed (we match the flow with its destination address)
fm.match.dl_dst = packet.dst
# Add an action to send to the specified port
# You need to figure the out_port by yourself
action = of.ofp_action_output(port=out_port)
fm.actions.append(action)
# Send message to switch
self.connection.send(fm)
```

- Q.1 Have h1 ping h2, and h1 ping h8 for 100 times (e.g., h1 ping -c100 p2). How long does it take (on average) to ping for each case? Any difference from Task III (the MAC case without inserting flow rules)?
- Q.2 Run "iperf h1 h2" and "iperf h1 h8". What is the throughput for each case? What is the difference from Task III?
- Q.3 Please explain the above results — why the results become better or worse?

- Q.4 Run pingall to verify connectivity and dump the output.
- Q.5 Dump the output of the flow rules using "ovs-ofctl dump-flows" (in your container, not mininet). How many rules are there for each OpenFlow switch, and why? What does each flow entry mean (select one flow entry and explain)?

E. [Optional] Task V: Layer-3 routing

The task will allow us to experience a bit of layer-3 routing. This is a simplified version of an IP router. We can use "`h1 ~ h8 ifconfig`" to find out that the IP addresses of h1 ~ h8 are from 10.0.0.1 to 10.0.0.8. Install **IP-matching** rules onswitch 7, but let other switches stay as MAC learning switches (same as Section C). The goal is that all hosts from h1 to h4 can ping h5 to h8, and switch 7 forwards packets via IP-matching flow rules (not MAC-matching rules). You can install IP-matching rules in a controller or using ovs-ofctl commands (hint: figure it out how to use this command).

Submission & Grading

Submit your assignment on the blackboard as one **tar-gzipped** file (generated using the tar command with cvzf options).In the tar-gzipped file, include all your Python code, a PDF report with answers to all the questions in each task. DO NOT submit each file individually. DO NOT include the entire POX code – include only the files you change. Make sure your Python code can run without errors.

- Task I: Submit the Python script that generates a binary tree topology.
- Task II: Answer all the questions (Q.1 to Q.4) in the report.
- Task III: Submit the Python controller code, and answer the three questions..
- Task IV: Submit the Python controller code, and answer the five questions.
- Task V: (Optional) Submit the controller code, or describe the method to set up the IP-matching rules.

In addition, you will also schedule an appointment with TA to have a demo before the deadline. We will grade your assignment based on this report and the demo.

- Task II: Demo Q.2 and Q.3.
- Task III: Demo Q.2 and Q.3.
- Task IV: Demo Q.1, Q.2 and Q4.
- Task V: (Optional) Demo the code.