Programming Assignment

AY 2019 – 2020 / Semester 1

Overview

In this programming assignment, you will learn to use the Mininet network emulation environment to setup a virtual network, and program the OpenFlow controller POX to implement several network management applications. You will learn through this assignment (1) to create virtual hosts, switches and links that connect them together, (2) to set parameters such as link capacity, and (3) to implement applications such as virtual LANs and premium service class for this network.

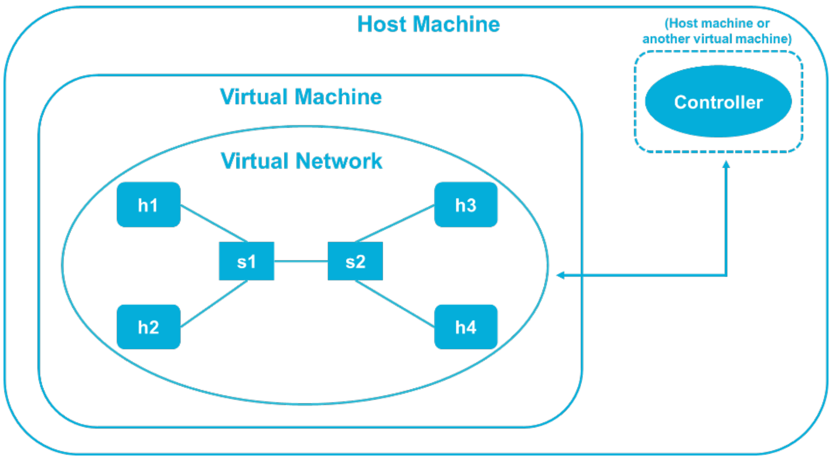

Figure 1. The logical overview of the system environment.

The virtual network created by the Mininet emulator resides in a virtual machine (VM) in the host machine. A pre-configured VM image with Mininet installed can be downloaded from Mininet's official site. This Mininet VM image itself also comes with the POX controller (under ~/pox).

However, we recommend that you install the POX controller on another virtual machine (**Ubuntu 14.04** is recommended), or on your own host machine, to help yourself get a clear view of how the controller runs and communicates with the virtual network in the Mininet VM. Figure 1 provides a logical view of the environment you will work on.

Task 1: Building a Virtual Network (20%)

In this section, your task is to build a virtual network in the Mininet network emulation environment. The network topology is given in an input file. Below is an example of a network topology, consisting of 7 **hosts**, 4 **switches** and 11 **links** that connect them together, as shown in Figure 2. In this network, the 7 hosts (with IP addresses from 10.0.0.1 to 10.0.0.7) are connected with 3 switches (with dpid from 1 to 3) by 7 links, each of which has a capacity of 10Mb/s. The 3 switches and another switch (with dpid

4) are then connected by 4 links, each of which has a capacity of 100Mb/s. The network topology and the capacity of each link are also shown in the following figure.

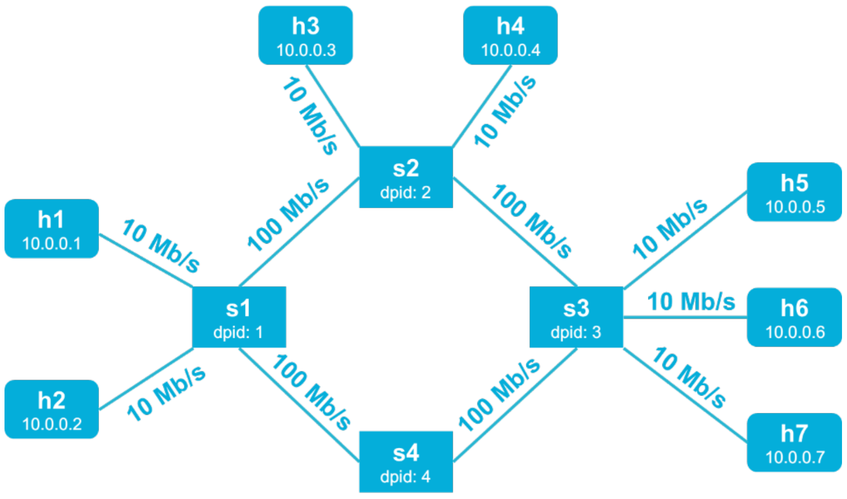

Figure 2. An example of network topology with 7 hosts, 4 switches and 11 links.

You need to build a network with the topology and specifications described in the input file **topology.in**. The first line of the file has 3 integers N and M, L indicating that there are N hosts $(h_1, h_2, \ldots, h_N)$, M switches $(s_1, s_2, \ldots, s_M)$ and L links in the network. The following are L lines of tuples in the form of $< dev1,\ dev2,\ bw >$ that describe the links, meaning that $dev1$ and $dev2$ are connected via a bi-directional link of capacity bw Mb/s at both directions. A partial input file for the network in Figure 2 looks like follows:

7, 4, 11

h1, s1, 10

h2, s1, 10

h3, s2, 10

…

The sample input file **topology.in** for the network in Figure 2 will be provided for your reference. You will also get a template of script written in Python, based on which you can add your own code to create the hosts, switches and links in Mininet.

Task 2: Learning Switch (30%)

Building a virtual network is not enough for the network to operate. In order for the hosts to communicate with each other, the switches need to know how to correctly route a packet to its destination. In this part, your task is to implement the application of self-learning switches using the POX controller, which enables the switches to learn how to route packets without any prior knowledge of the network topology. In order to adapt to possible topological changes, each learned route is associated with a Time-To-Live (TTL), assumed to be 30 seconds in this task. After the TTL, the existing route expires and should be evicted, and a new route needs to be learned accordingly.

The key idea is to record the MAC address and the corresponding port number when receiving a packet that does not match any entry in the forwarding table. Thus, the switches can gradually learn the port number for reaching each host in the network. In the following, we provide a simple example for illustration.

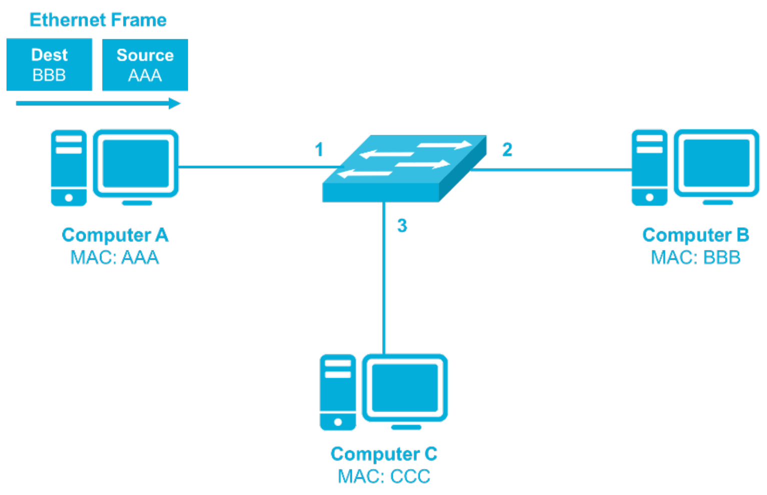

Figure 3. Example of a self-learning switch: Initial state

Initially, the switch has no knowledge of the three computers A, B and C that are connected to it as shown in Figure 3. When Computer A sends a frame to Computer B, the switch does not know how to route the frame. It then **floods** this frame to all the ports, hoping that Computer B could receive it and reply.

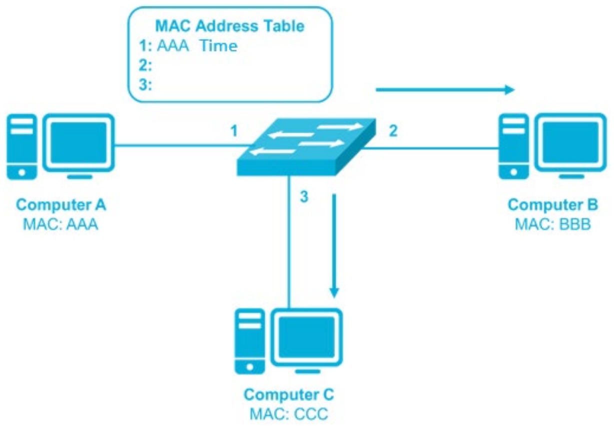

Figure 4. Example of a self-learning switch: Learning MAC address of A

Although now the switch still knows nothing about Computer B, it has learned that Computer A can be reached by port 1. It then adds this **entry** with **current timestamp** into its forwarding table as is shown in Figure 4, so that a frame destined for Computer A will be routed to port 1 in the future. Similarly, when the reply from Computer B comes back, the switch can subsequently learn that port 2 corresponds to Computer B. Last but not the least, the switch will periodically check if any route has already existed for TTL amount of time, i.e., 30s for our case, and delete the route if that happens.

Task 3: Virtual Local Area Networks (LANs) (20%)

In this section, you will implement a layer-3 application of virtual LANs using the POX controller. The physical network needs to block all **TCP** traffic between two hosts belonging to different LANs on port **`4001`**. You will be provided with an input file. The first line of the file has one integer N indicating the number of LANs. The second line of the file has N integers $L_1, L_2 \dots L_N$ indicating the number of hosts for each LAN. The following lines including a number of IP addresses in the form of

```
10.0.0.1

10.0.0.3
```

…

Hosts from 1 to L_1 hosts belong to the 1st LAN. And hosts from $L_1 + 1$ to $L_1 + L_2$ belong to the 2nd LAN, and so on.

The basic idea is that when a connection between the switch and the controller is up, the application installs flow entries that **drop** all the **TCP** packets between two hosts belonging to different LANs.

Task 4: Premium Service Class (30%)

In this part, your task is to implement a premium service class for certain hosts to receive higher throughput by allocating higher bandwidth to them. Sometimes, hosts may desire higher throughput in order to guarantee the quality of its tasks. Such hosts can pay extra for the premium class of traffic which guarantees at least X Mb/s of throughput, while the others stay with the normal service class, for which the throughput is limited to at most Y Mb/s. In our topology, assume a link between a host and a switch has the capacity of bw Mb/s, then X is set to be $0.8 \times bw$, while Y is $0.5 \times bw$. For simplicity, we also assume that links between switches will never become the bottleneck.

For Task 3 and 4, you will be provided with a file **policy.in** that describes the policies for the LAN and premium service class. The file starts with a line of two integers N and M, indicating that there are N different LANs and M hosts that have paid for the premium traffic. The second line of the file has N integers $L_1, L_2 \dots L_N$ indicating the number of hosts for each LAN. The following contains $(L_1 + L_2 + \cdots + L_N)$ lines of hosts $< ip >$ for each LAN, followed by M lines listing the hosts. An example of the file is shown as follows:

2 3

4 3

10.0.0.1

10.0.0.2

```
10.0.0.3
10.0.0.4
10.0.0.5
10.0.0.6
10.0.0.7
10.0.0.1
10.0.0.2
10.0.0.5
```

The above example shows that given the topology shown in Figure 2, h_1 (10.0.0.1), h_2 (10.0.0.2), h_3 (10.0.0.3), h_4 (10.0.0.4) belong to LAN 1, and h_5 (10.0.0.5), h_6 (10.0.0.6), h_7 (10.0.0.7) belong to LAN 2. In addition, h_1 (10.0.0.1), h_2 (10.0.0.2) and h_5 (10.0.0.5) are guaranteed to have at least 8 Mb/s throughput, while the throughput for other hosts are upper-bounded by 5 Mb/s.

The key idea of this application is to set up queues with different bandwidths at the interfaces of the switches using the **ovs-vsctl** tool that configures the Open vSwitch. Basically, **ovs-vsctl** helps to set up QoS queues in the Mininet virtual networks. When a packet arrives, the controller checks whether it belongs to premium or normal traffic, and then sends the packet to the corresponding queues.

Handling Conflicts

The POX controller supports multiple concurrent applications. However, it is possible that one application will be in conflict with another application. In this assignment, the premium traffic application should not allow communication which is blocked by the LAN application. You are to make sure that no conflicting rules are installed by two different applications.

OpenFlow message structure `ofp_flow_mod` has an attribute called `priority`. Conflicts can be handled by setting different priority values to different rules.

Problem in POX's Default Configuration

When you need to **flood** packets in POX, remember to modify the attribute `port` to `ofp.OFPP_ALL` (which is by default `ofp.OFPP_FLOOD`) to avoid the situation that the packet is not actually flooded.

Understanding Provided Files

mininetTopo.py	The script template for creating the required network topology.
controller.py	A skeleton for the controller.

topology.in A sample input file for building the virtual network. During the demo, your program should be able to read another given input file.

policy.in A sample input file for LAN policies and premium traffic. Similarly, another input file will be given for the demo.

Submission Guideline

You need to submit two Python program files, named **mininetTopo.py** and **controller.py**. The first program should take the topology file **topology.in** as input, and create the required virtual network. The second program should take the policy file **policy.in** as input, and implement the LAN and premium traffic applications. In addition, please submit a short **summary report** that describes the design of your implementation.

Please submit your assignment **BEFORE 23:59:59, 17 November (Sun) THROUGH LumiNUS**. ***Submissions after the deadline will be penalized in grades.*** Add your **name** and **student number** as a comment in the codes you submit, and also at the beginning of your summary report. Create a zip file consisting of the codes and summary report, and name it with your **student number**, for example, "A0123456X.zip" if your student number is A0123456X. ***Submissions in any other formats will be penalized.***

You are strongly encouraged to test your codes thoroughly before submission. **Your assignment (.zip) should be submitted to the "Submission" folder in the LumiNUS workbin (the "Assessments/Programming Assignment" folder)**. If you have further questions in submission, please send an email to zhaochen.s@u.nus.edu or dourengan@u.nus.edu.

Demo

You need to demonstrate your implementation to the TA. In general, the TA will use a set of topology and policy files different from the given examples to test your programs. For Task 1, we will run **pingall** command and expect to see that all hosts can ping each other. For Task 2, **`pingall`** and **`iperf`** will be used to measure the accessibility between different hosts, and Mininet's **`link down`** command will be used to emulate the scenarios of topological changes. For Task 3, we will use **`iperf`** to test whether the traffic between 2 hosts from different LANs is blocked. For Task 4, **`iperf`** will be used to measure the achieved throughput for different hosts from different service classes.

The demo will be scheduled after your final exam. Each student will have to select a 15-minute time slot on LumiNUS. The detailed dates for demo and time slots selection will be later announced on LumiNUS.