Divakaran Haritha, A0187915N

CS4226 Programming Assignment Report

**Intro**

This project is used to Mininet network emulation environment to simulate a virtual network by creating virtual hosts, switches and links and connecting them together. We use OpenFlow POX controller to set parameters such as link capacity for premium and normal service classes for the network and implement firewall for certain links.

**Task 1**

To build a virtual network, we use the following functions from Topo’s class *addHost()*, *addSwitch()* and *addLink()* methods to add the hosts, switches, and links respectively from the *topology.in* input file provided. Based on the first line of the file, the corresponding number of hosts, switches and links are read in. A nested dictionary in python is used where the keys are *dev1, dev2* and the values are the bandwidth (*bw*) capacity for each link. To start the network, *sudo python mininetTopo.py* command is used.

**Task 2**

To make sure that the hosts communicate with each other and the switches know how to correctly route a packet to its destination, a self-learning switches using POX controller is implemented. Using a nested dictionary in python, DPIDs of switches and source MAC addresses are added as keys where the ports that maps to the MAC addresses are added as values.

Initially, when the controller receives a packet from a switch or a host, it checks for corresponding DPID in the nested dictionary. If the destination MAC address is present, the controller will instruct by sending a message to map the destination MAC address to its corresponding port in the flow table, signalling all future packets with that matching destination MAC address will be routed to that port.

If the corresponding DPID of the incoming packet that the controller receives is not in the dictionary, it initializes in the table for that switch.

If the source MAC address is not in the table for that particular DPID, a new entry is added where the MAC address and the corresponding input port is added.

If the destination MAC address is not present in the table, *flood()* method is invoked, where it floods all openflow ports except the input port and those with flooding disabled via the OFPPC\_NO\_FLOOD port configuration.

**Task 3**

To ensure fault tolerance when certain links are down, a TTL value of 30 seconds is set. On controller, the port it maps to, and its time of entry is added for the source MAC address as pair values in the dictionary. When a packet is handled, if the current time exceeds the time of entry for its corresponding destination MAC address by 30 seconds, this entry is removed from the routing table.

**Task 4**

Policies are read in via the *policy.in* input file. For incoming TCP packets, it checks the source IP address, destination IP address and the destination port number and whether this combination matches any of the firewall policies in the input file using *msg.match.* If there is a match, the program drops packets and subsequently stops switches to forward packets that matches this combination.

**Task 5**

On the network side two queues, q0 and q1 are setup using *os-vsctl*. Queue q0 is hardcoded as normal while q1 is hardcoded as premium. This is done so by setting the bandwidth rates as follows:

Considering bw = bandwidth,

q0’s **maximum** bandwidth = 0.5\*bw (normal)  
q1’s **minimum** bandwidth = 0.8\*bw (premium)

On the controller side, premium hosts are read in via the *policy.in* input file where they are added to a premium list. If incoming packets’ destination IP address match the addresses in the premium list, q\_id = 1 (q\_id is queue ID), else set q\_id = 0. Following this, the appropriate bandwidth is allocated to these hosts by sending a message by using *ofp\_action\_enqueue*.