CSci4211: Introduction to Computer Networks Programming Assignment III: Software Defined Networking

Due Date: Monday November 14, 2016 at 11:59pm

November 2, 2016

In this project you will learn how to use Mininet to create virtual networks and run simple experiments. Also, you get familiar with writing a simple controller for managing the network for these virtual networks.

Part1: Introduction to the tools

Mininet:

For the this assignment, you will learn how to use Mininet to create virtual networks and run simple experiments. According to the Mininet website, Mininet creates a realistic virtual network, running real kernel, switch and application code, on a single machine (VM or native), in seconds, with a single command. We will use Mininet in the following programming assignments.

The first step is to Install mininet using this link.
To control and manage the virtual network from a single console, Mininet includes a network-aware command line interface (CLI). To launch the CLI, the command is sudo mn.

Mininet Walkthrough

Once you have a Mininet VM, you should complete the first three sections of the standard Mininet walkthrough. This helps you to get familiar with Mininet environment.
Next is an introduction to controllers used to manage network.

POX Controller:

After installing POX, Go to the directory holding pox.py file in another remote SSH terminal: $cd ~/pox
Then, start the hub: $ pox.py log.level –DEBUG forwarding.hub
Then you can check the connectivity of each host by commands like pingall.

After setting up the network environment, a controller is needed to install flow entries to the switch so that packets can be forwarded among hosts.

In general a POX controller consists of three parts:

1. Listener
2. Control logic
3. Messenger

First you need to figure out the type of the event you want the controller to listen to (e.g., ConnectionUp, PacketIn, etc). Then using some logic you can distinguish between different flows and attach a proper action for a specific flow. Finally you send the message to the switch to add the new rule in the Openflow table.

Listening to an event in POX

There are two common ways for an application to register with the controller for events.

1. Register callback functions for specific events thrown by either the OpenFlow handler module or specific modules like Topology Discovery
    - From the launch or from the init of a class, perform core.openflow.addListenerByName("EVENTNAME", CALLBACK_FUNC, PRIORITY)
    - For instance you already developed a switching function named switch(). If you want to trig the function when the controller starts to work, add a listener in lunch to provoke function switch when ConnectionUp handler is called: core.openflow.addListenerByName("ConnectionUp", switch).
1. Register object with the OpenFlow handler module or specific modules like Topology Discovery
    - From typically the init of a class, perform addListeners(self). Once this is added, the controller will look for a function with the name _handle_EVENTNAME(self, event). This method is automatically registered as an event handler.

Event handlers

As explained above you need to set a listener on your POX and when an event happens the relevant handler function will be activated. Your control logic should be placed in one of these handlers.
The two main handlers that you might need to modify in your program are:

- ConnectionUp, which is activates when a switch turns on (or connects to the controller). The name of the handler function for this event is: _handle_ConnectionUp. Be careful about timers when implementing this handler. ConnectionUp triggers only once when the switch starts to work.
- PacketIn, activates by arriving a packet into the controller. The name of the handler is _handle_PacketIn.

Parsing Packets:

POX provides several primitives to parse well-known packets. The ethernet packet received through a*packet_in* event can be extracted using this command and then the match for the flow rules can be created using the *from_packet* function:

```
packet = event.parsed
src_mac = packet.src
dst_mac = packet.dst
if packet.type == ethernet.IP_TYPE:
   ipv4_packet = event.parsed.find("ipv4")
   # Do more processing of the IPv4 packet
   src_ip = ipv4_packet.srcip
   src_ip = ipv4_packet.dstip
```

Useful POX API

- ofp_flow_mod OpenFlow message

  This composes a message which tells a switch to install a flow entry. It has a match attribute and a list of actions. Notable fields are:
  - actions – A list of actions to perform on matching packets.
  - priority – This specifies the priority for overlapping matches. Higher values are of higher priority.
  - match – A ofp_match object. By default, none of the fields of this object is set. You may need to set some of its fields
- ofp_match class

  This class describes packet header fields and an input port to match on. Fields not specified are "wildcards" and will match on any value.
  For example, create a match which matches packets arriving on port 3:
  ```
  match = of.ofp_match()
  match.in_port = 3
  ```
- ofp_action_output class

  This is an action which specifies a switch port that you want to send the packet out of. There is a variety of pre-defined "port numbers" such as OFPP_FLOOD.
  For example, create an action which sends packet out of port 2.
  ```
  out_action = of.ofp_action_output(port = 2)
  ```
- connection.send(…)

  This function sends an OpenFlow message to a switch

For example, create a flow_mod that sends packets arriving on port 3 out of port 4
```
msg = of.ofp_flow_mod()
msg.match.in_port = 3
msg.actions.append(of.ofp_action_output(port = 4))
connection.send(msg)
```

Floodlight Controller:

Floodlight is Java-based and intended to run with standard JDK tools and ant and can optionally be run in Eclipse.

Use this link for installing floodlight on your system. First you need to figure out the type of the event you want the controller to listen to (e.g., ConnectionUp, PacketIn, etc). Then using some logic you can distinguish between different flows and attach a proper action for a specific flow. Finally you send the message to the switch to add the new rule in the Openflow table.

Getting messages from switches (Listening)

IOFMessageListener:

First you need to register your module:

```
… implements IFloodlightModule, IOFMessageListener{
      protected IFloodlightProviderService floodlightProvider;
      …
      public void init(FloodlightModuleContext context) throws FloodlightModuleException {
            floodlightProvider = context
                  .getServiceImpl(IFloodlightProviderService.class);
            floodlightProvider.addOFMessageListener(OFType.PACKET_IN, this);
            …
      }
      …
}
```

Then you need to handle the packet:

```
… implements IFloodlightModule, IOFMessageListener{
      …
      public Command receive(IOFSwitch sw, OFMessage msg, FloodlightContext cntx) {
            OFPacketIn pi = (OFPacketIn) msg;
            Ethernet eth = IFloodlightProviderService.bcStore.get(cntx,
                              IFloodlightProviderService.CONTEXT_PI_PAYLOAD);
            …
      }
      …
}
```

Operations on Packets

Creating match

```
OFMatch match = new OFMatch();
match.setWildcards(Wildcards.FULL.matchOn(Flag.DL_TYPE).matchOn(Flag.NW_DST).
withNwDstMask(24) );
match.setDataLayerType( Ethernet.TYPE_IPv4 );
match.setNetworkSource( IPv4.toIPv4Address("152.3.140.0") );
```

Result:

<ip packet, ip=152.3.140.0/24>

Flow Rules: Actions

In SDN switches are dumb and action tells them what to do with a matched packet

Actions are:

- Send packet out to a port
- Modify the packet's header

Examples:

- <output=3>
- <mod_nw_src=123.45.67.89, output=1>

Creating an Action:

```
ArrayList<OFAction> actions = new ArrayList<OFAction>();
OFActionOutput action = new OFActionOutput().setPort((short) 3);
OFActionNetworkLayerSource ofanls = new OFActionNetworkLayerSource();
ofanls.setNetworkAddress( IPv4.toIPv4Address("8.8.8.8") );
```

Result:

<output = port 3, modify IP address to = 8.8.8.8>

Putting it together:

```
OFFlowMod flowMod = new OFFlowMod();
flowMod.setMatch( match );
flowMod.setActions( actions );
flowMod.setLength( OFFlowMod.MINIMUM_LENGTH + OFActionOutput.MINIMUM_LENGTH +
                   OFActionNetworkLayerSource.MINIMUM_LENGTH) );
try {
         sw.write(flowMod, cntx);
         sw.flush();
} catch (IOException e) {
         log.error("Failure writing flowMod", e);
}
```

Check out this link to see how to process a PacketIn packet. Check out this link for floodlight tutorial.

Notice: you can use this VM on which what ever you need for this project is installed.

Part2: Controller

Consider the topology in Figure 1, in this task, you should implement a controller module that adds forwarding entries in switches to route traffic between hosts by extracting information from the packets sent between them. One possible approach (similar with the self-learning algorithm) is to add the forwarding entries based on the incoming port from which the packet is received, and the fields in the packet header (e.g., source MAC, source IP, or any other proper fields). This entry will be used for forwarding the future packets towards that source host, since the switch assumes that source host is reachable through that port. As an example shown in Figure 1, assume s1's forwarding table is initially empty, and h1 sends a packet to h2. When s1 receives the packet from the incoming port (say port 1), it checks the forwarding table but does not find a forwarding entry. s1 then sends this packet to the controller. The controller parses the packet and knows that h1 is reachable for s1 through port 1, so it adds a forwarding entry in s1 to instruct s1 to forward the future packets destined to h1 out from port 1. In addition, the controller instructs s1 to flood this packet to the other ports except for the incoming port. Similar process happens on s2, and the packet eventually gets delivered to h2. Later if a packet destined to h1 arrives at s1, s1 should be able to forward the packet to the correct port by checking its forwarding table.

- **POX Controller**
  To get started writing your own controller in POX, you should put sample_routing.py code in the directory pox/pox/samples. Implement your controller using this file. You will then run your controller using the following commands:
  ```
  you@yourmachine$ cd pox
  you@yourmachine$ ./pox.py samples.simple_routing
  ```
- Floodlight Controller
  To get started writing your own controller in Floodlight, put SampleRouting.java code in src/main/java/net/floodlightcontroller/simplerouting.
  Need to tell Floodlight where the application is:
  Add your application path to: src/main/resources/META-INF/services/net.floodlightcontroller.core.module.IFloodlightModule
  Add: net.floodlightcontroller.simplerouting.SimpleRouting
  Tell floodlight to run your application:
  - Add your application to <src/main/resources/floodlightdefaultproperties>
    Add: net.floodlightcontroller.simplerouting.SimpleRouting
  - In ~/floodlight directory (where ever build.xml) is, run ant to build a jar file. It will be in the ~/floodlight/target directory.
  - Run <java -jar target/floodlight.jar> command and the controller is now listening on port 6633.

Part3: Link Latency and Throughput

After completing part2 consider the topology in Figure1. In this part you are asked to measure the throughput and delay of each link in the topology and c0 is your controller from part2. Please run your controller first, and then run Mininet with the provided topology (written in a Python script assign1_topo.py) using sudo:

sudo python assign1_topo.py

Hosts (h1 - h5) are represented by squares and switches (s1 - s5) are represented by circles and c0 is the controller; the names in the diagram match the names of hosts and switches in Mininet. The hosts are assigned IP addresses 10.0.0.1 through 10.0.0.5; the last number in the IP address matches the host number. When running ping and Iperf in Mininet, you should use IP addresses.

You should measure the RTT and bandwidth of each of the four individual links between switches (L1 -L4). You should run ping with 20 packets and store the output of the measurement on each link in a file latency_L#.txt, replacing # with the link number from the topology diagram. You should run Iperf for 20 seconds and store the output of the measurement on each link in a file throughput_L#.txt, replacing # with the link number from the topology diagram in Figure 1.

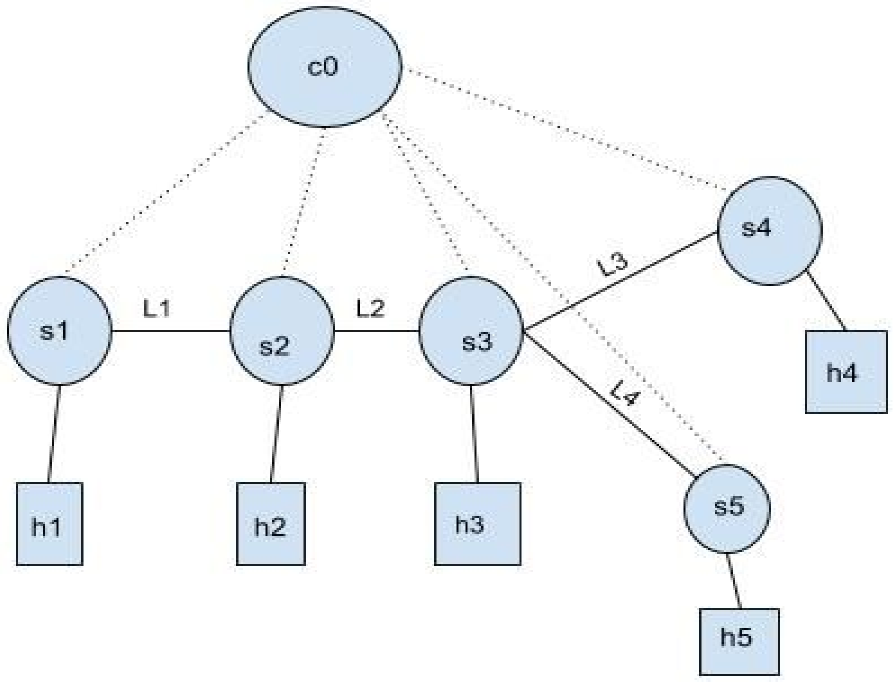


Figure1 : Network topology

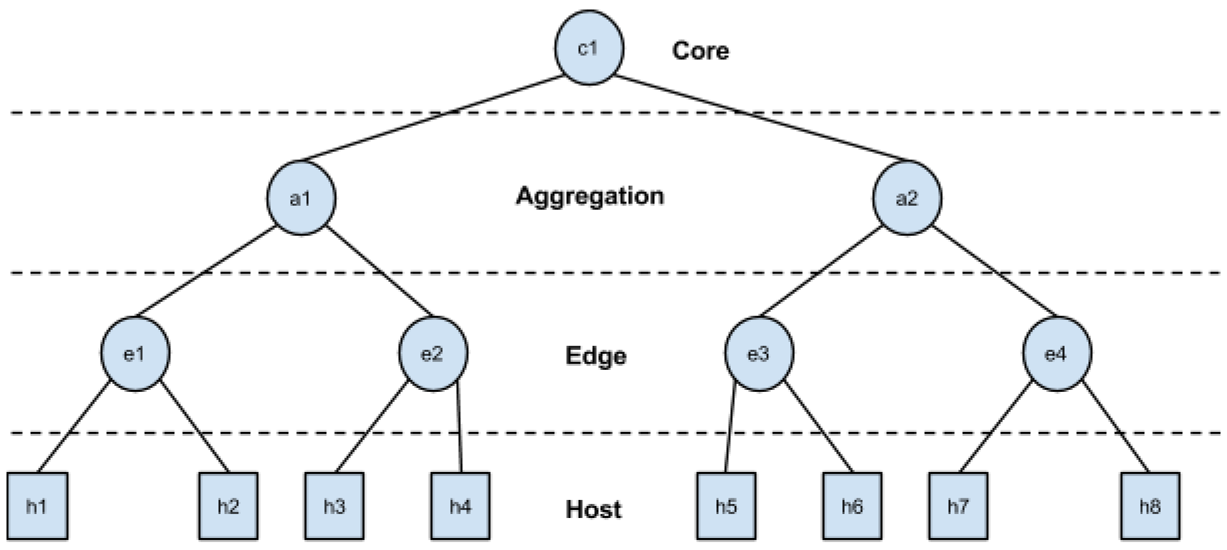
Part4: Path Latency and Throughput

Now, assume h1 wants to communicate with h4. What is the expected latency and throughput of the path between the hosts? Write down your prediction in your report.
Measure the latency and throughput between h1 and h4 using ping and Iperf. It does not matter which host is the client and which is the server. Use the same parameters as above (20 packets / 20 seconds) and store the output in files latency_Q3.txt and throughput_Q3.txt. Put the average RTT and measured throughput in the report file and explain the results. If your prediction was wrong, explain why.

Part5: Building your own topology

Data center networks typically have a tree-like topology. End-hosts connect to top-of-rack switches, which form the leaves (edges) of the tree; one or more core switches form the root; and one or more layers of aggregation switches form the middle of the tree. In a basic tree topology, each switch (except the core switch) has a single parent switch. Additional switches and links may be added to construct more complex tree topologies (e.g., fat tree) in an effort to improve fault tolerance or increase inter-rack bandwidth.

In this assignment, your task is to create a simple tree topology. You will assume each level i.e., core, aggregation, edge and host to be composed of a single layer of switches/hosts.



Simple Tree Topology

For creating this topology, you should consider that each level links, have specified performance parameters. Here there are 3 types of links, the links between core and aggregation switches, the links between aggregation and edge switches, links between edge switches and host. Your logic should support setting at least bandwidth and delay parameters for each link. The parameters are given in the tree_topo.py file.

After building your topology, try pinging all hosts to see if the connections are working. Store the output to Q5_test.txt.

Methods you need in creating and testing a topology:

1. Topo: the base class for Mininet topologies
2. addSwitch(): adds a switch to a topology and returns the switch name
3. addHost(): adds a host to a topology and returns the hostname
4. addLink(): adds a bidirectional link to a topology (and returns a link key, but this is not important). Links in Mininet are bidirectional unless noted otherwise.
5. Mininet: main class to create and manage a network
6. start(): starts your network
7. pingAll(): tests connectivity by trying to have all nodes ping each other
8. stop(): stops your network
9. net.hosts: all the hosts in a network
10. dumpNodeConnections(): dumps connections to/from a set of nodes.

What to submit:

- For each part write down the scripts and commands you use in mininet for measuring. the asked values. Write down these scripts in script.txt file .
- Answer the questions asked in each part and write them down in answers.txt file
- Please put all of the above results for each part and codes you implement in a folder naming part#, replace # with the part number.
- Please put comments in your codes
- Submit a zip file with the format of ID_project3.zip (e.g. babai008_project3.zip)

References:

http://sdnhub.org/tutorials/sdn-tutorial-vm/
https://floodlight.atlassian.net/wiki
http://mininet.org/
https://github.com/mininet/openflow-tutorial/wiki/Create-a-Learning-Switch