INSTITUTO SUPERIOR TÉCNICO

Traffic Engineering

Lab Project #5

Software-Defined Networking and OpenFlow (part II)

Introduction to SDN controllers

Fernando Mira da Silva

NOVEMBER 2017

1 Goal

In this lab project, we will complement the previous class with a quite simple introduction to SDN controllers using mininet. Note that this guide should be completed on a single lab class. The report of this lab class should cover the previous and this lab, and must be delivered until Saturday, 25 Nov., at 8:00PM.

In this lab class we will show how to develop a simple module for the POX controller. POX was the first Python based SDN controller. While POX was already superseded by several more recent and powerful controller frameworks (e.g. Ryu, Floodlight, OpenDaylight), it is has the advantage that it is simpler and enough to understand the basics of network programming.

Most of this guide is adapted from the mininet / openflow tutorial.

2 POX

POX provides a framework for communicating with SDN switches using OpenFlow. Developers can use POX to create an SDN controller using Pyhton. While superseded by more recent controller frameworks, it still is a popular tool for teaching about and researching software defined networks and network applications programming.

POX can be immediately used as a basic SDN controller by using the stock components that come bundled with it. Developers may create a more complex SDN controller by creating new POX components. Or, developers may write network applications that address POX's API. In this class, we only briefly discuss programming for POX.

3 Required software

The POX controller is already included built-in in the mininet VM image used in the previous class, therefore no further software is required. The controller software is under the pox directory of the home directory.

4 Starting mininet

Start mininet with the following command:

```
$ sudo mn --topo single,3 --mac --switch ovsk --controller remote
```

Note: if you have already an open mininet session, close it and type "mn –c" to clear all data before launching the new topology.

This creates the same simple topology of the previous class, with three hosts interconnected by an Open vSwitch.

Now you can try running a basic hub example from the pox directory:

```
 $./pox.py log.level --DEBUG misc.of_tutorial
```

This tells POX to enable verbose logging and to start the of_tutorial component which you'll be using. In its default version, the controller will act as an hub, flooding all ports on each received packet.

The switches may take a little bit of time to connect. When an OpenFlow switch loses its connection to a controller, it will generally increase the period between which it attempts to contact the controller, up to a maximum of 15 seconds. Since the

OpenFlow switch has not connected yet, this delay may be anything between 0 and 15 seconds. If this is too long to wait, the switch can be configured to wait no more than N seconds using the --max-backoff parameter. Alternately, you exit Mininet to remove the switch(es), start the controller, and then start Mininet to immediately connect.

Wait until the application indicates that the OpenFlow switch has connected. When the switch connects, POX will print something like this:

```
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633

INFO:openflow.of_01:[00-00-00-00-00-01 1] connected

DEBUG:misc.of_tutorial:Controlling [00-00-00-00-00-01 1]
```

The first two lines are from the portion of POX that handle OpenFlow connections. The third is from the tutorial component itself.

5 Analyzing Hub behavior

Now we verify that hosts can ping each other, and that all hosts see the exact same traffic - the behavior of a hub. To do this, we'll create xterms for each host and view the traffic in each. In the Mininet console, start up three xterms:

```
 mininet> xterm h1 h2 h3
```

Note: the xterm command does not work and will throw an error if you try to call it from the virtual box directly, instead use another terminal window to call xterm.

Arrange each xterm so that they're all on the screen at once. This may require reducing the height of to fit a cramped laptop screen.

In the xterms for h2 and h3, run tcpdump or Wireshark, and make a capture in each hx-eth0 interface.

In the xterm for h1, send a ping:

```
 # ping -c1 10.0.0.2
```

The ping packets are now going up to the controller, which then floods them out all interfaces except the sending one. You should see identical ARP and ICMP packets in both h2 and h3. This is how a hub works; it sends all packets to every port on the network.

Now, see what happens when a non-existent host doesn't reply. From h1 xterm:

```
 # ping -c1 10.0.0.5
```

You should see three unanswered ARP requests in your capture. If your code is off later, three unanswered ARP requests is a signal that you might be accidentally dropping packets.

Test in host 1 a ping of 10 packets to h3. Compute the maximum, minimum, and average RTT. Repeat the same test for h2. Discuss the results.

6 OF programming in POX

The subsections below give details about POX APIs that should prove useful for this exercise.

6.1 Sending OpenFlow messages with POX

```
connection.send( ... ) # send an OpenFlow message to a switch
```

When a connection to a switch starts, a ConnectionUp event is fired. The example code creates a new Tutorial object that holds a reference to the associated Connection object. This can later be used to send commands (OpenFlow messages) to the switch.

6.1.1 ofp_action_output class

This is an action for use with ofp_packet_out and ofp_flow_mod. It specifies a switch port that you wish to send the packet out of. It can also take various "special" port numbers. An example of this would be OFPP_FLOOD which sends the packet out all ports except the one the packet originally arrived on.

Example. Create an output action that would send packets to all ports:

```
out_action = of.ofp_action_output(port = of.OFPP_FLOOD)
```

6.1.2 ofp_match class

Objects of this class describe packet header fields and an input port to match on. All fields are optional -- items that are not specified are "wildcards" and will match on anything.

Some notable fields of ofp_match objects are:

```
dl_src - The data link layer (MAC) source address

dl_dst - The data link layer (MAC) destination address

in_port - The packet input switch port
```

Example. Create a match that matches packets arriving on port 3:

```
match = of.ofp_match()

match.in_port = 3
```

6.1.3 ofp_packet_out OpenFlow message

The ofp_packet_out message instructs a switch to send a packet. The packet might be one constructed at the controller, or it might be one that the switch received, buffered, and forwarded to the controller (and is now referenced by a buffer_id).

Notable fields are:

```
buffer_id - The buffer_id of a buffer you wish to send. Do not set if
you are sending a constructed packet.

data - Raw bytes you wish the switch to send. Do not set if you are
sending a buffered packet.

actions - A list of actions to apply (for this tutorial, this is just
a single ofp_action_output action).

in_port - The port number this packet initially arrived on if you are
sending by buffer_id, otherwise OFPP_NONE.
```

Example. of_tutorial's send_packet() method:

```
action = of.ofp_action_output(port = out_port)

msg.actions.append(action)

# Send message to switch
```

```
self.connection.send(msg)
```

6.1.4 ofp_flow_mod OpenFlow message

This instructs a switch to install a flow table entry. Flow table entries match some fields of incoming packets, and executes some list of actions on matching packets. The actions are the same as for ofp_packet_out, mentioned above (and, again, for the tutorial all you need is the simple ofp_action_output action). The match is described by an ofp_match object.

Notable fields are:

```
idle_timeout - Number of idle seconds before the flow entry is removed. Defaults to no idle timeout.

hard_timeout - Number of seconds before the flow entry is removed. Defaults to no timeout.

actions - A list of actions to perform on matching packets (e.g., ofp_action_output)

priority - When using non-exact (wildcarded) matches, this specifies the priority for overlapping matches. Higher values are higher priority. Not important for exact or non-overlapping entries.

buffer_id - The buffer_id of a buffer to apply the actions to immediately. Leave unspecified for none.

in_port - If using a buffer_id, this is the associated input port.

match - An ofp_match object. By default, this matches everything, so you should probably set some of its fields!
```

Example. Create a flow_mod that sends packets from port 3 out of port 4.

```
fm = of.ofp_flow_mod()

fm.match.in_port = 3

fm.actions.append(of.ofp_action_output(port = 4))
```

For more information about OpenFlow constants, see the main OpenFlow types/enums/structs file, openflow.h, in ~/openflow/include/openflow/openflow.h.

6.1.5 Parsing Packets with the POX packet libraries

The POX packet library is used to parse packets and make each protocol field available to Python. This library can also be used to construct packets for sending.

The parsing libraries are in:

```
pox/lib/packet/
```

Each protocol has a corresponding parsing file.

For the first exercise, you'll only need to access the Ethernet source and destination fields. To extract the source of a packet, use the dot notation:

```
packet.src
```

The Ethernet src and dst fields are stored as pox.lib.addresses.EthAddr objects. These can easily be converted to their common string representation (`str(addr)` will return something like "01:ea:be:02:05:01"), or created from their common string representation (`EthAddr("01:ea:be:02:05:01")`).

To see all members of a parsed packet object:

```
print dir(packet)
```

Many fields are common to all Python objects and can be ignored, but this can be a quick way to avoid a trip to a function's documentation.

7 POX of_tutorial code for simple hub

When running POX with the command, as

```
$./pox.py log.level --DEBUG misc.of_tutorial
```

what we are doing is to run the python code of_tutorial.py located under the pox/pox/misc directory.

Just to start with, create a backup copy of of_tutorial.py and open this file with your favorite editor and check its structure.

In the initializer (the _init_ method), the class's instance variables are set accordingly and an empty dictionary (on that will later consist of MAC Address keys to port number values) is created.

Each packet sent to the controller is processed by handle_packetIn, which in turn invokes act_like_hub().

Act_like_hub, describes the hub behavior, sending received packets to all ports, using the resend_packet. Look at the comments for these two methods for some more information about what's going on.

Analyse the code and explain, on your own words, the behaviour of act_like_hub().

8 Implementing a learning switch

8.1 *Implementing a learning switch without installing table flow entries*

In the following, you'll be editing the of_tutorial.py file. Each time you edit and save it, do not forget to (1) stop the controller and (2) restart pox from scratch. In general, you don't need to restart mininet itself.

In of_tutorial.py, *handle*packetIn(), comment the act_like_hub() call and uncomment the (unfinished) act_like_switch() call.

Try to finish the act_like_switch() method to implement a functional learning switch considering the following rules:

For each packet reaching the controller:

1. If the source mac was not seen before, register the association between the source mac and the input port in mac_to_port dictionary.
2. If the destination mac was not seen before, flood the packet to all output ports (as it happened in act_like_hub()).
3. If the destination mac is already associated to a given switch port, send it to the appropriate port only.

Recall:

Input port number is available through packet_in.in_port;

Source mac address is available in packet.src;

Destination mac address is available in packet.dst;

Resend packet can be performed by resend_packet(packet, [port number]), being that flooding is possible specifying instead of port_number the constant of.OFPP_ALL.

Perform all required tests (and include results on the written report) to show that:

1. The code implements a learning switch through the controller, e.g., where all packets are sent to the controller;
2. Show that the mac to port table is properly learned;
3. Repeat the ping statistics performed in section 5 and discuss the results.

8.2 Implementing a learning switch with table flow entries

Instead of sending all packets to the controller, one may install action rules in the open Vswitch whenever a new mapping is learned. Add to the act_like_switch method the required lines to program table flow entries whenever a new mac address is learnt;

Perform again all required tests to show that

1. The code implements a learning switch where most of the packets do not go through the controller;
2. Show that the mac to port table is properly learned;
3. Show that the flow table is properly installed and which rules are installed in different learning phases (recall the command ovs-ofctl);
4. Repeat the ping statistics performed in section 5 and discuss the results.

9 Implementing a router (optional)

If (and only if) you have completed all the steps above and you have spare time, you may try to implemente a static layer-3 forwarder/switch, following the guide available at https://github.com/mininet/openflow-tutorial/wiki/Router-Exercise.