Laboratory 5 OpenFlow Protocol

Part 1: POX Controller

POX is a Python-based SDN controller platform geared towards research and education. For more details on POX, see the POX Wiki: https://openflow.stanford.edu/display/ONL/POX+Wiki/

Python:

- is a dynamic, interpreted language. There is no separate compilation step - just update your code and re-run it.
- uses indentation rather than curly braces and semicolons to delimit code. Four spaces denote the body of a for loop, for example.
- is dynamically typed. There is no need to pre-declare variables and types are automatically managed.
- has built-in hash tables, called dictionaries, and vectors, called lists.
- is object-oriented and introspective. You can easily print the member variables and functions of an object at runtime.
- runs slower than native code because it is interpreted. Performance-critical controllers may want to distribute processing to multiple nodes or switch to a more optimized language.

Common operations:

To initialize a dictionary:

```
mactable = {}
```

To add an element to a dictionary:

```
mactable[0x123] = 2
```

To check for dictionary membership:

```
if 0x123 in mactable:
     print 'element 2 is in mactable'
if 0x123 not in mactable:
     print 'element 2 is not in mactable'
```

To print a debug message in POX:

```
log.debug('saw new MAC!')
```

To print an error message in POX:

```
log.error('unexpected packet causing system meltdown!')
```

To print all member variables and functions of an object:

```
print dir(object)
```

To comment a line of code:

```
# Prepend comments with a #; no // or /**/
```

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.

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)
```

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
```

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)
```

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 You may also wish to consult POX's OpenFlow library in pox/openflow/libopenflow_01.py and, of course, the OpenFlow 1.0 Specification.

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)
```

Here's what you'd see for an ARP packet:

```
['HW_TYPE_ETHERNET', 'MIN_LEN', 'PROTO_TYPE_IP', 'REPLY', 'REQUEST', 'REV_REPLY',

'REV_REQUEST', '', '', '', '', '',

'', '', '', '', '', '',

'', '', '', '', '',

'', '', '', '', '_init', 'err',
```

```
'find', 'hdr', 'hwdst', 'hwlen', 'hwsrc', 'hwtype', 'msg', 'next', 'opcode',
'pack', 'parse', 'parsed', 'payload', 'pre_hdr', 'prev', 'protodst', 'protolen',
'protosrc', 'prototype', 'raw', 'set_payload', 'unpack', 'warn']
```

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.

Part 2. Practice

[*https://github.com/mininet/openflow-tutorial/wiki/Create-a-Learning-Switch*]

Step 1: Start Mininet network with POX remote controller

Build a network application based on POX Controller. You will use a starter code for a hub controller. You'll modify the provided hub to act as an L2 learning switch. In this application, the switch will examine each packet and learn the source-port mapping. Thereafter, the source MAC address will be associated with the port. If the destination of the packet is already associated with some port, the packet will be sent to the given port, else it will be flooded on all ports of the switch.

Later, you'll turn this into a flow-based switch, where seeing a packet with a known source and dest causes a flow entry to get pushed down.

We will start Mininet with a remote controller:

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

Go to POX folder:

```
$ cd pox
```

If pox is not installed, you can download it from the POX repository on github into your VM::

```
$ git clone http://github.com/noxrepo/pox
```

Now you can try running a basic hub example:

```
$./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 (which currently acts like a hub).

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:

```
INFO:openflow.of_01:[Con 1/1] Connected to 00-00-00-00-00-01

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

The first line is from the portion of POX that handles OpenFlow connections. The second is from the tutorial component itself.

Step 2: Verify Hub Behavior with tcpdump

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`.

In the **xterms** for **h2** and **h3**, run `tcpdump`, a utility to print the packets seen by a host:

```
# tcpdump -XX -n -i h2-eth0
```

and respectively:

```
# tcpdump -XX -n -i h3-eth0
```

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 corresponding to the ping in both **xterms** running `tcpdump`. 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 the `tcpdump` **xterms**. If your code is off later, three unanswered ARP requests is a signal that you might be accidentally dropping packets.

Step 3: Benchmarking the POX hub

Here, you'll benchmark the provided **of_tutorial hub**.

First, verify reachability. Mininet should be running, along with the POX hub in a second window. In the Mininet console, run:

```
mininet> pingall
```

This is just a sanity check for connectivity. Now, in the Mininet console, run:

```
mininet> iperf
```

Now, compare your number with the reference controller you saw before. How does that compare?

Hint: every packet goes up to the controller now.

Step 3: Creating the learning switch

Go to your SSH terminal and stop the tutorial hub controller using Ctrl-C. The file you'll modify is `pox/misc/of_tutorial.py`. Open this file in your favorite editor. **Vim** is a good choice, it comes already downloaded with terminal. It has some funky commands to edit text so http://vim.rtorr.com can be of use. To use **vim**, make sure you are in the correct directory (`pox/pox/misc`) and enter:

```
$ vi of_tutorial.py
```

The current code calls `act_like_hub()` from the handler for `packet_in` messages to implement switch behavior. You'll want to switch to using the `act_like_switch()` function, which contains a sketch of what your final learning switch code should look like.

Each time you change and save this file, make sure to restart POX, then use pings to verify the behavior of the combination of switch and controller as a (1) hub, (2) controller-based Ethernet learning switch, and (3) flow-accelerated learning switch. For (2) and (3), hosts that are not the destination for a ping should display no tcpdump traffic after the initial broadcast ARP request.

To test your code: Make sure you have mininet running and then put `./pox.py log.level --DEBUG misc.of_tutorial` in the other terminal window. Once it's connected, try a couple pings to see if the switch is working. The bandwidth returned by `iperf` from a switch (Gbits) should be much faster than a hub (Mbits).

To test your controller-based Ethernet switch, first verify that when all packets arrive at the controller, only broadcast packets (like ARPs) and packets with unknown destination locations (like the first packet sent for a flow) go out all non-input ports. You can do this with `tcpdump` running on an `xterm` for each host.

Once the switch no longer has hub behavior, work to push down a flow when the source and destination ports are known. You can use `ovs-ofctl` to verify the flow counters, and if subsequent pings complete much faster, you'll know that they're not passing through the controller. You can also verify this behavior by running `iperf` in Mininet and checking that no OpenFlow packet-in messages are getting sent. The reported `iperf` bandwidth should be much higher as well, and should match the number you got when using the reference learning switch controller earlier.

References:

- http://mininet.org/walkthrough/
- https://github.com/mininet/mininet/wiki/Introduction-to-Mininet
- https://github.com/mininet/mininet/wiki/Documentation
- https://openmaniak.com/iperf.php
- https://github.com/mininet/openflow-tutorial/wiki