Workshop in Communication Networks

Exercise 1 – Layer 2 – Advanced Learning Switch

General Guidelines:

Submission deadline is **Sunday, March 30, 23:55 moodle server time**
Submit your answers as TXT and Python files, packed into a single ZIP file.
Pack your files in a ZIP file named ex0-YourName1-YourName2.zip,
for example: ex0-LoisLane-ClarkKent.zip
Post the ZIP file in the submission page in course website

- No late submission will be accepted!
- Document your code. Place your names at the top of every source file, as well as in the TXT file in part 2.
- Make sure your files have EXACTLY the requested file names.
- We may give a bonus for exceptional works. Thus, do not make anything more complex than required.

Important: You are not allowed to use any predefined POX modules in this exercise, except for the core modules that are already imported in the template file `of_lecture.py`

The grading for this exercise will be mainly automatic. If your code does not work as expected, the grade will be very low.

Your submission ZIP file should contain the following files:

- Part 1 code – named **of_learning_switch.py**
- Part 2 code – named **of_learning_switch_spanning_tree.py**
- A text file named **description.txt** that describes how you implemented each step in part 2

Your code in both parts must include log messages for a small set of important events. See the complete list in Appendix A. This will help you track your controller, and will help us grade it.

Prerequisites

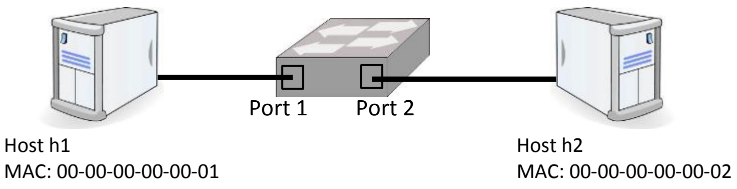
In order to complete this exercise you must install MiniNet and POX, and go over the MiniNet tutorial from class. Also, you must learn the basics of python, including Object Oriented Programming with python, lists and dictionaries.

More useful links for this exercise:
MiniNet Walkthrough | OpenFlow Switch Specification | The official OpenFlow Tutorial
Programming in POX | POX Wiki | Disjoint-set Data Structure | Kruskal's Algorithm

Part 1 – Build OpenFlow Layer 2 Learning Switches (File name: of_learning_switch.py, 40 points)

In class we have created a controller that makes switches flood every packet they receive, and thus, in practice, behave like hubs. In this part we will change this behavior such that the switches will learn the ports packets arrive from, and upon receiving a packet, if they have already seen its destination address, they will know the exact port to forward it on and avoid flooding the network.

Example in a non-OpenFlow switch:
Assume we have a single switch with two connected ports: 1 and 2, and two hosts, h1, h2, as illustrated below:



Now assume the following scenario, and the expected non-OpenFlow switch behavior:

1. h1 sends a packet to h2
   The switch does not know where h2 (MAC address …-00-02) is (there can be many ports), so it floods the packet to all ports except for the input port.
   However, the switch knows that the packet has arrived on port 1, therefore the switch can learn that host h1 (MAC address …-00-01) can be reached through port 1.
2. h2 sends a message to h1
   The switch can now use its knowledge about the location of h1 (MAC address …-00-01) to avoid flooding. The switch simply sends the packet on port 1 - that's it!

(In an OpenFlow switch this gets a bit more complex, so the above scenario does not hold exactly)

<u>Instructions:</u>
You should begin with the file we wrote in class (`of_lecture.py`, available in course website) and implement the method `act_like_switch(...)` in class `Tutorial`.

The method `act_like_switch(...)` receives three parameters:

- **`self`** – the instance of tutorial class. We use this object to call other methods in the class like send_packet().

- **`packet`** – the packet that is associated with this OpenFlow event and was sent from the switch to the controller
  Two useful fields of this object are:
  `packet.src` – the source MAC address of the packet (of type `EthAddr`)
  `packet.dst` – the destination MAC address of the packet (of type `EthAddr`)

- **`packet_in`** – the OpenFlow container packet descriptor
  Three useful fields of this object are:
  `packet_in.in_port` – the port number on which the packet arrived at the switch
  `packet_in.buffer_id` – the unique ID of the packet in the switch (may not be present)
  `packet_in.data` – the raw data of the packet (may not be present)

The method should learn the location of the source host (i.e. the port it is connected to, possibly indirectly) and look up the port to which the destination host is connected to (again, possibly indirectly). If the **destination port is known**, the controller should **<u>install a flow entry</u>** for the **current input port, source MAC and destination MAC** (think why) on the switch, so it will not ask the controller again about this route. Otherwise, the controller should direct the switch to flood the packet to all ports but the input port (using `of.OFPP_FLOOD` port value). You should NOT install a flood entry in the switch flow table, as you do not want this to be the permanent action for such packets.
The network should keep working correctly even if a link fails (goes down). In such a case, the controller should recognize that packets from already-known sources come from a different port, and update the flow table accordingly.

Note: You should change the event handler `_handle_PacketIn(...)` to call `act_like_switch(...)` instead of `act_like_hub(...)`.

Tips:

- You may start with a controller that does not install flow entries at all but still avoid flooding by learning the forwarding tables of the switches. Instead of installing a flow entry, start with just sending the specific packet to its correct output port.
- To direct the switch to send a single packet, send it an `of.ofp_packet_out()` message.
- To install a flow entry on the switch flow table, send it an `of.ofp_flow_mod()` message.
- When installing a flow entry the controller should also include the packet details so the switch will forward it correctly in addition to installing the flow entry (use the `packet_in.buffer_id` and `packet_in.data` fields for that)
- There exists a single `Tutorial` instance per switch-controller connection.
- Use `log.debug(string)` to output debugging messages to console. Use `str(...)` to convert other types (including `EthAddr`) to string.
- Use the hint comments in the code file itself.
- Remember, packets arrive to the controller (POX) only if the received packet doesn't match to all the installed flow entries.

Test your controller:

1. Open two terminal windows with `ssh` connections to the MiniNet machine (with X forwarding)
2. In one terminal, copy the exercise file to `~/pox/pox/samples/` using `scp`
3. In the same terminal, run POX:
```
     ~$ cd pox
 ~/pox$ ./pox.py log.level –DEBUG samples.of_learning_switch
```
4. In the other terminal, run MiniNet with a simple topology:
```
~$ sudo mn -c
~$ sudo mn --topo single,3 --mac --switch ovsk --controller remote
```
5. In the MiniNet console, open two xterm windows for each one of the three hosts
6. Find the IP of each host using `ifconfig`
7. On the first xterm window of each host, start `tcpdump` on its Ethernet interface
8. On the second xterm of one host, say h1, ping another host, say h2: `ping -c 1 <IP>`
   You should see an ARP message and the *echo request* packet in both terminals of h2 and h3, as the switch does not know where h2 is and it floods the packet.
9. Now ping in the opposite direction, from h2 to h1. You should see the *echo request* messages only in the terminal of h1, and the *echo response* messages only in the terminal of h2, as the switch is supposed to know both of them already.
10. Test your controller on a topology with two switches, using the provided `topologies.py` file:
```
~$ sudo mn --custom ./topologies.py --topo SimpleTreeTopology \
     --mac --switch ovsk --controller remote
```
11. Test your controller for the case of link failure / link down (follow instructions carefully):
    - Shut down your controller and mininet.
    - Clean mininet: `~$ sudo mn -c`
      Run mininet with the following topology:
```
~$ sudo mn --custom ./topologies.py --topo SimpleLoopTopology \
         --mac --switch ovsk --controller remote
```
    - In the mininet console, type:
```
mininet> link s1 s2 down
```
    - Now start your controller. Use `ping` or `pingall` to test that everything works. Then, in the mininet console, type the following commands:
```
mininet> link s2 s3 down
mininet> link s1 s2 up
```

See next step and a warning in the next page…

- Now, use `pingall` several times until all switches learn the new flow records again (you can follow the expected changes in every switch flow table to determine the convergence).

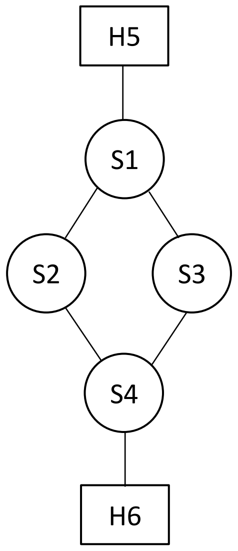
Important: Do not start the controller when all links are up as this will cause a loop that your controller is not expected to handle in this part (this will change in the next part).
When testing your switch for link failures (bullet 11 in the testing section of part 1), take a look at the ARP tables of the hosts (use **hostname arp -a** in the mininet console) - before and after each step. The learning of the new ports for each host is based on the behavior of the ARP mechanism and not only on the IP (e.g. ping) messages between hosts.

12. You may create more complex topologies to test your controller with (note that MiniNet does not support connecting a host to more than one switch, so usually, complex topologies are built between switches).
13. **Make sure your controller has the required logging as listed in Appendix A!**

Part 2 – Make your network loop-resistant

(File name: of_learning_switch_spanning_tree.py, 60 points)
You should create a copy of your code file from part 1, and use it as a basis for your work in this section. You must submit files for both parts. You can import classes from the `utils.py` file that is provided with this exercise.
That is, your Part 1 code must work correctly BEFORE you start working on this part!

A common problem in layer 2 networks is loops that are created as a result of link redundancy. Such redundancy can be created in purpose, to overcome possible link failure, or by accident, when the network is complex.
When there exists a loop in a layer 2 network, packets may loop and change the forwarding tables each time, again and again, forever.



For example, consider the network on the right. In consists of four switches and two hosts. When H5 first sends a packet to H6, S1 will flood it to both S2 and S3. S2 and S3 will learn that H5 is on their "upper" port, and will forward the packet to the "lower" ports. S4 will receive the packet twice, but let's see what happens upon receiving the first packet (say it came from S2): it will flood it to all other ports (to H6 and S3). S3 will now receive a packet from H5 on its "lower" port, and thus will change its forwarding table. It will flood the packet to the "upper" port as it still does not know where H6 is, and the packet will look forever.

You can see this in MiniNet by running it on the custom topology file `topologies.py` that is provided with this exercise, along with your learning switch from Part 1:

```
~$ sudo mn --custom ./topologies.py --topo LoopTopology --mac \
           --switch ovsk --controller remote
```

The solution to this problem is to compute a spanning tree over the graph of links, and neutralize the edges that are not in the tree. In distributed networks, this is done using the Spanning Tree Protocol. In a Software-Defined Network, this can be done centrally and much more easily.

This part is broken into steps. You must follow them and document how you implemented each one of them, in a separate file called **description.txt**. This is mandatory.
(The instructions are general and not strict, so you have a lot of implementation details to decide yourself)

See Appendix B for some useful pieces of code.

A. Learn the topology
LLDP (Link Layer Discovery Protocol) is a layer 2 protocol for discovering network topology. The basic idea is that LLDP packets are not forwarded and thus if a switch received an LLDP packet it can be sure that the sender is a direct neighbor of it.
In short, LLDP is a special type of Ethernet packets that are sent to some specified MAC address - '\x01\x80\xc2\x00\x00\x0e' in Python, and are not forwarded by receiving switch.

Your task is:

1. Create a singleton class named `Discovery` that will be responsible for topology discovery. In `launch()`, use `core.register('discovery', Discovery())` to register the class to the relevant events and create the following methods:
   - `_handle_ConnectionUp(self, event)`
     Will be called when a switch is added. Use `event.dpid` for switch ID, and `event.connection.send(...)` to send messages to the switch.
   - `_handle_ConnectionDown(self, event)`
     Will be called when a switch goes down. Use `event.dpid` for switch ID.
   - `_handle_PortStatus(self, event)`
     Will be called when a link changes. Specifically, when `event.ofp.desc.config` is 1, it means that the link is down. Use `event.dpid` for switch ID and `event.port` for port number.
   - `_handle_PacketIn(self, event)`
     Will be called when a packet is sent to the controller. Same as in the previous part. Use it to find LLDP packets (`event.parsed.type == ethernet.LLDP_TYPE`) and update the topology according to them.

   Note: To create a singleton class, use `utils.SingletonType`. See the example in `utils.py`.
2. Make every switch send an LLDP messages to its neighbors, every one second, so that its neighbors will know it. Use `utils.Timer` for that.
3. Make switches send LLDP packets they receive to the controller (do that by setting a flow record that will forward packets of L2 type (`dl_type`) of `ethernet.LLDP_TYPE`, that are sent to the LLDP destination MAC address, with an action of sending the packet to the port `of.OFPP_CONTROLLER`).
4. Also, you should prevent the forwarding of LLDP messages, by avoiding flooding/sending them when received in the `_handle_PacketIn` event handler method of your `Tutorial` class that handles switches' connections.
5. You should have timestamps for links in the network so that if a list goes down (or just becomes non-useful due to high loss rate or congestion), your Discovery class will remove it from the graph after a while. Use `time.time()` for that.
6. Once receiving a LLDP packet, update the topology accordingly. You can use `utils.Graph` to represent the graph. If there is nothing to update, just update the timestamp of the corresponding edge.
7. At every 3 seconds, scan timestamps of all edges. If an edge was not seen for more than 6 seconds, remove it.

The result of this section should be a mechanism that learns the links between all switches, to which switches and ports they are connected. This mechanism will be used later to update

NOTE: You may ignore links between hosts and switches as these are not discovered using LLDP. Since in mininet, a host is only connected to (at most) one switch, this is OK for loop avoidance.

Remember to add informative logging (use `log.debug(`*`message`*`)`) as listed in Appendix A. You may add more messages that will show the operation of the `Discovery` class, but avoid adding too much logging.

B. Find a Spanning Tree
After learning the topology, your controller can find a spanning tree over the graph of links. The spanning tree will be used to determine the links that are allowed to be used. By avoiding the links that are not part of the spanning tree (the forbidden links), we will avoid switching messages in a circular manner and break possible loops.

Implement the spanning tree mechanism using Kruskal's algorithm. You may use the `UnionFind` class from `utils.py`.

You may create another singleton class that manages the spanning tree. This class should receive the reference to the graph object maintained by the Discovery class, and when a link changes, the Discovery class should notify this class that there was a change (link went up or down), and the spanning tree should be updated accordingly.

The spanning tree mechanism should mark each port, on each switch, whether it is allowed to send packets to it or not (more specifically, this is necessary only on ports that are connected to other switches). Later on, you will have to connect this logic to the switch logic you already created in Part 1.

C. Change your switch behavior to avoid sending on forbidden links
Once you determined the spanning tree, your controller should make sure switches do not send on forbidden links. That is:

- Controller should remove any flow record that outputs messages to the forbidden port
- Controller should avoid learning new flow records on the forbidden port
- Controller should avoid flooding to the forbidden port (You should take special care for flooding as you should no longer use the `OFPP_FLOOD` generic port, but instead to send to all non-forbidden ports - by setting multiple actions).

Use `self.connection.features.ports` to get all ports of a switch (in class Tutorial). Each element in the returned list is a port object with several field, the most useful one is `port_no`. Note that this list also includes the switch's port to the controller. You should avoid forwarding packets to it when flooding. Its number is greater that `of.OFPP_MAX`, so use this constant to do that.

D. Test your controller
Run your controller and start MiniNet with the loop topology as explained above. Use tcpdump or wireshark to track packets, and send a message (using `ping`, etc.) from one host to the other. Make sure that the packet arrives correctly and only once.
Play with mininet – take links up and down, and see that your controller handles the changes correctly and that the network works as expected.
You can also create more complex topologies and test your code with them as well.

Make sure your controller has the required logging as listed in Appendix A!

Make sure you document your work in a separate file – description.txt, and attach it to your submission!

Appendix A – Required Logging

The following events must print some organized log messages in the DEBUG log level (using `log.debug(`*`message`*`)`):

Part 1:

- Switch is being told to flood a packet (state destination MAC address and input port)
- A flow record is set on a switch (state matching fields and values, and specified actions)
- A flow record is removed from a switch (state matching fields and values)

Part 2:

- Switch is being told to flood a packet (state destination MAC address and input port)
- A flow record is set on a switch (state matching fields and values, and specified actions)
- A flow record is removed from a switch (state matching fields and values)
- New link is found (state endpoints (switch, port)<->(switch,port))
- Existing link is removed (state endpoints, and the reason – port closed, not found for long time, etc.)
- Spanning tree enables a link (or a port)
- Spanning tree disables a link (or a port)

Appendix B - Useful Pieces of Code:

Install a flow record on a switch and also send the specified packet according to this rule:

```
fm = of.ofp_flow_mod()
fm.match.dl_dst = dst

# it is not mandatory to set fm.data or fm.buffer_id
if packet_in.buffer_id != -1 and packet_in.buffer_id is not None:
        fm.buffer_id = packet_in.buffer_id
else:
        if packet_in.data is None:
                return
        fm.data = packet_in.data

# Add an action to send to the specified port
action = of.ofp_action_output(port=out_port)
fm.actions.append(action)

# Send message to switch
connection.send(fm)
```

Remove a flow record from a switch:

```
fm = of.ofp_flow_mod()
fm.command = of.OFPFC_DELETE
fm.match.dl_dst = src # change this if necessary
connection.send(fm) # send flow-mod message
```

Send an LLDP packet to all ports of a switch that was discovered with the `ConnectionUp` event:

```
dst = Discovery.LLDP_DST_ADDR        # == '\x01\x80\xc2\x00\x00\x0e'

for p in event.ofp.ports:
    if p.port_no < of.OFPP_MAX:
        # Build LLDP packet
        src = str(p.hw_addr)
        port = p.port_no

        lldp_p = lldp() # create LLDP payload
        ch_id = chassis_id() # Add switch ID part
        ch_id.subtype = 1
        ch_id.id = str(event.dpid)
        lldp_p.add_tlv(ch_id)
        po_id = port_id() # Add port ID part
        po_id.subtype = 2
        po_id.id = str(port)
        lldp_p.add_tlv(po_id)
        tt = ttl() # Add TTL
        tt.ttl = Discovery.LLDP_INTERVAL # == 1
        lldp_p.add_tlv(tt)
        lldp_p.add_tlv(end_tlv())

        ether = ethernet() # Create an Ethernet packet
        ether.type = ethernet.LLDP_TYPE # Set its type to LLDP
        ether.src = src # Set src, dst
        ether.dst = dst
        ether.payload = lldp_p # Set payload to be the LLDP payload

        # send LLDP packet
        pkt = of.ofp_packet_out(action = of.ofp_action_output(port = port))
        pkt.data = ether
        event.connection.send(pkt)
```

Parse an LLDP packet upon receiving a `PacketIn` event:

```
pkt = event.parsed
lldp_p = pkt.payload
ch_id = lldp_p.tlvs[0]
po_id = lldp_p.tlvs[1]

r_dpid = int(ch_id.id)
r_port = int(po_id.id)
```