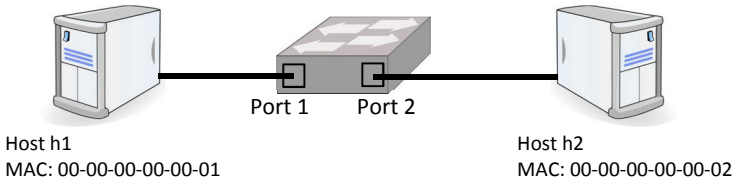
# OpenFlow / ARP Storm

**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 )
In of_flood _switch.py file 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:
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 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!

**Instructions:**
You should begin with the file (`of_flood_switch.py`, available in the zip) 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 **install a flow entry** for this destination on the switch so it will not ask the controller again about it.
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.
12. You may create more complex topologies (for correctness – check with loop free topology) 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 controller answer for ARP requests**

(File name: of_learning_switch_answer_arps.py,)
**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!***
Note: Also in this part test your solution only on loop free topology. If you want your solution to work also on more complex topology you need to activate L2 spanning tree (but this is not a requirement for this project).

A common layer 2 problem in large networks is that hosts go up and down. Each node has an ARP table, which is a mapping between layer 3 addresses (IP addresses) and layer 2 addresses (MAC addresses). When a host tries to send a packet to another host, it first looks up the host's MAC address using its ARP table. If there is a hit, the packet is then sent directly (i.e. unicast) to the host. If there is not entry for it, an ARP request packet is flooded through the whole network, for the host to answer. In large networks, with nodes which are tightly coupled, an "ARP Flood Storm" may happen, by sending a large number of ARP requests for hosts that are currently down.

Finding a solution to this problem is not an easy task, but Software-Defined networks and OpenFlow introduce us to new possibilities and solutions. Some possible solutions were introduced in paper:
Victor Boteanu, Hanieh Bagheri , Martin Pels**, "** Minimizing ARP traffic in the AMS-IX switching platform using OpenFlow"

**See the Appendix of the paper for some useful pieces of code.**

**Make your controller answer ARP requests**

First, read the research paper. The paper discusses possible solutions in order to reduce the number of ARP packets that are being flooded in the network, by using OpenFlow. The writers proposed five solutions, and implemented one of them.

Your task is:

1. Implement a similar solution to the one implemented in the paper.
2. While the original solution used an XML file containing the IP-to-MAC mapping, you will not use such a file.
3. Instead, you must install a flow to match all ARP requests packets in every switch, and make them send the ARP requests to the controller.
4. The controller must hold not only a MAC-to-Port mapping, but also a IP-to-MAC mapping. The mapping should be learned dynamically, just like in part 1.
5. Make the controller answer for every ARP request possible, or flood it otherwise.
6. Are there any ARP requests to which you should not answer?
7. When does a IP-to-MAC mapping entry changes? Should you alert of such a case?
8. Do not forget to test your controller. You **must provide a Wireshark pcap file** containing the flow of:
    - An ARP request sent by a host
    - The packet-in sent from the switch to the controller containing info about the packet
    - The packet-out sent from the controller to the switch
    - The packet sent from the switch back to the host (ARP reply) or the flooded ARP request
    - In order the view OpenFlow packets in Wireshark, you must install the correct dissector.
9. Evaluate your solution regarding the reduction of the number of ARP requests broadcasted in the network:
    - Create a several Mininet topologies, each time with a different number of hosts.
    - Ping from each host to all others. Remember to clean the ARP cache of every host after each iteration, in order to force the hosts to send an ARP request.
    - Show (using a graph) the comparison of the number of ARPs broadcasted as a function of the number of hosts, between a "normal" OpenFlow network, and a network that use your solution.

The result of this section should be a mechanism in the controller similar to the one implemented in the paper, a Wireshark pcap file, and an evaluation of your solution.

**Remember to add informative logging (use `log.debug(message)`) 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)
- A new IP-to-MAC mapping is found (state endpoints (IP, MAC) and (MAC ,Port))
- Existing link is removed (state endpoints, and the reason – port closed, not found for long time, etc.)
- A IP-to-MAC mapping is removed

**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
```