SDN Experiment 1

Introduction

In this experiment, we build a two-layer self-learning switch in an attempt to avoid flooding of data packages. In a SDN's environment, it can be interpreted as a strategy that for every switch, it shall learn from the mapping between the mac address of packages it receives and the port of the switch.

Furthermore, the controller shall dump flows to the switches to guild the deliver of packages in order to cope with the flooding towards every port.

Environment

Operating System: Linux version 4.15.0-20-generic
RYU Controller: 4.30 version
Mininet: 2.3.0d4 version

Content

In the previous we used POX as the controller of the mininet topology. This time we use RYU as the remote controller instead.

Constructing the controller

We add a dictionary in the class to save all the mapping between ports and mac addresses. When it receives a package, the switch shall decide whether this package has been learned or not. If yes, it will act according to the mapping, or else it shall learn from it and dump new flows.
(You shall see the code in the end.)

Activating the controller

You shall activate the ryu-manager before you build the topoloy. The instruction is:

```
1 sudo ryu-manager your_file_name
```

Constructing a simple topology

For your information, use the command below to build a simple mininet topology where there are a linear link among host1, host2 and host3 by switch1-3.

```
1 sudo mn --topo linear,3 --controller remote
```

Result

First, let's open the wireshark of host3's shell. You shall open host3'shell by using:

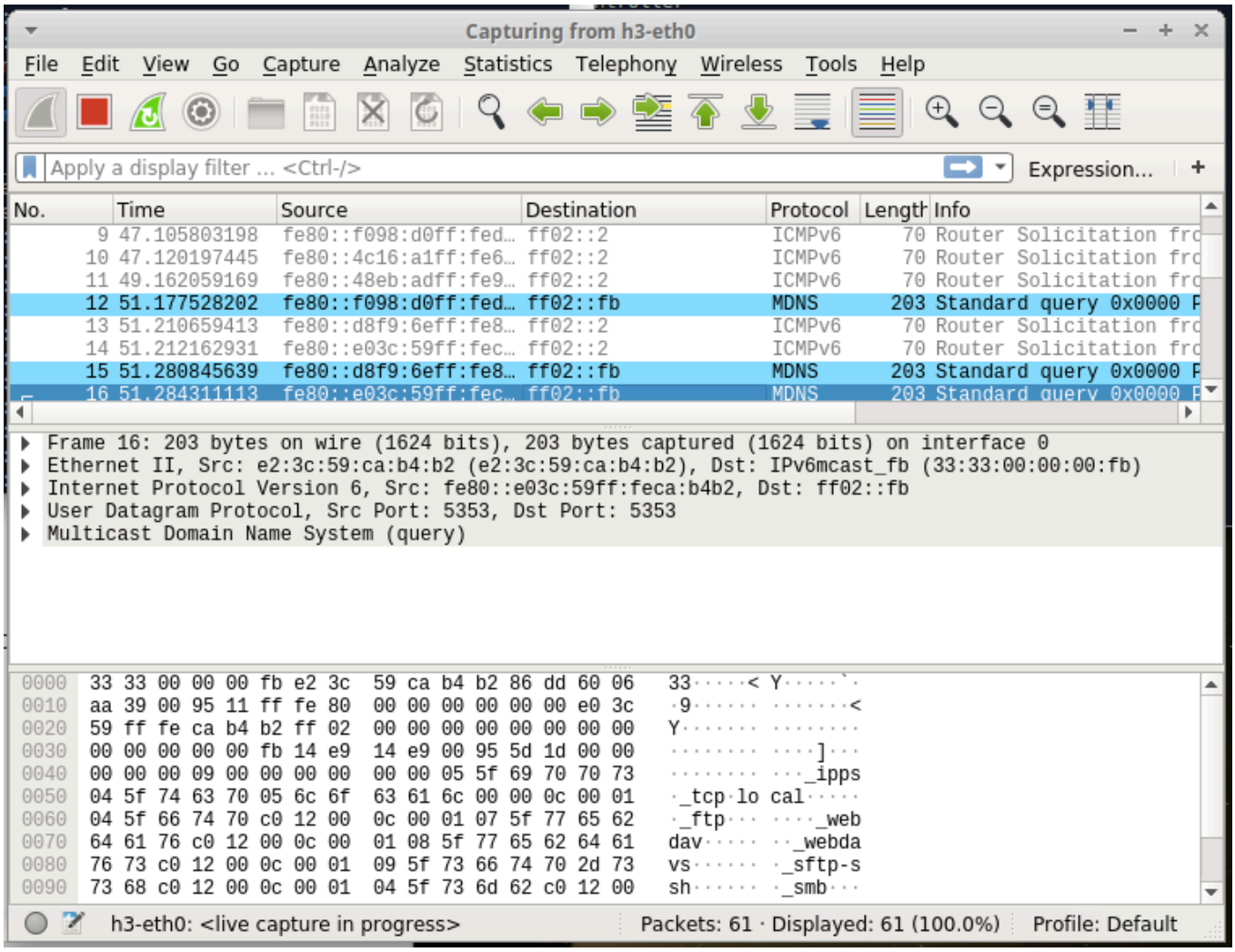
```
xterm h3
```

and then open the wireshark:

```
sudo wireshark
```

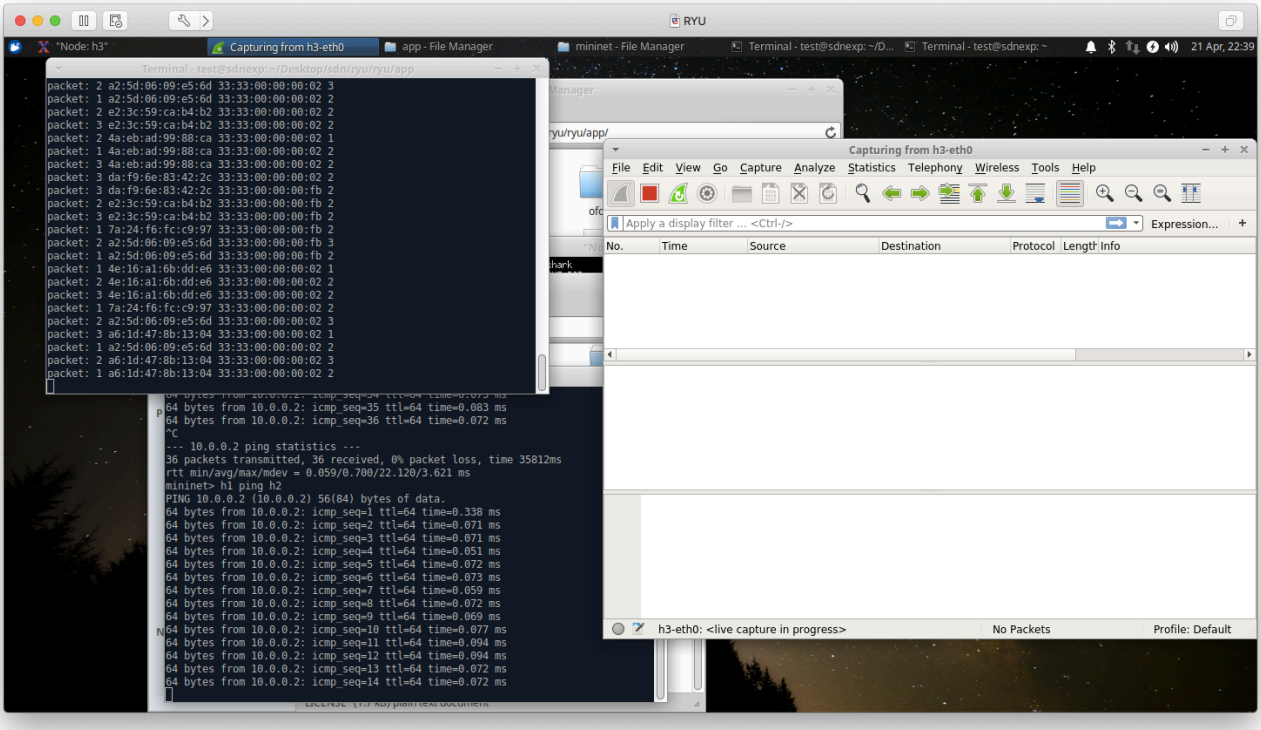
If you have seen the window of the wireshark, choose h3-eth0 port.

Next, let h1 ping h2 in the mininet shell. Since the controller has functioned correctly, the window shall be like this:



Note: The first 16 packages wireshark captured are used to create links before they don't exist, which means they are **neccesary** and are used for the switches to learn. After the learning procedure has been completed, if you let h1 ping h2, h3 shall receive **NO** packages from these two hosts.

In the picture below, it shows that when h1 is pinging h2, h3 receives no package.



Reference

[walkthrough of Mininet]
[RYU docs]
[A Blog of the Case]

*Attention: The codes on the 3rd website is **no longer working** in the mininet whose environment is descripted above. You shall modify it or refer to my codes.*

Codes

Here are the codes of the customed controller.

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER, CONFIG_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet

class LearningSwitch(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(LearningSwitch, self).__init__(*args, **kwargs)
        # maybe you need a global data structure to save the mapping
        self.mac_to_port = {}
```

```
16
17      def add_flow(self, datapath, priority, match, actions):
18          dp = datapath
19          ofp = dp.ofproto
20          parser = dp.ofproto_parser
21          inst = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
22          mod = parser.OFPFlowMod(datapath=dp, priority=priority, match=match,
    instructions=inst)
23          dp.send_msg(mod)
24
25      @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
26      def switch_features_handler(self, ev):
27          msg = ev.msg
28          dp = msg.datapath
29          ofp = dp.ofproto
30          parser = dp.ofproto_parser
31          match = parser.OFPMatch()
32          actions = [parser.OFPActionOutput(ofp.OFPP_CONTROLLER, ofp.OFPCML_NO_BUFFER)]
33          self.add_flow(dp, 0, match, actions)
34
35      @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
36      def packet_in_handler(self, ev):
37          msg = ev.msg
38          dp = msg.datapath
39          ofp = dp.ofproto
40          parser = dp.ofproto_parser
41          # the identity of switch
42          dpid = dp.id
43          self.mac_to_port.setdefault(dpid, {})
44          # the port that receive the packet
45          in_port = msg.match['in_port']
46          pkt = packet.Packet(msg.data)
47          eth_pkt = pkt.get_protocol(ethernet.ethernet)
48          # get the mac
49          dst = eth_pkt.dst
50          src = eth_pkt.src
51          # we can use the logger to print some useful information
52          self.logger.info('packet: %s %s %s %s', dpid, src, dst, in_port)
53
54          self.mac_to_port[dpid][src] = in_port
55          if dst in self.mac_to_port[dpid]:
56              out_port = self.mac_to_port[dpid][dst]
57          else:
58              out_port = ofproto_v1_3.OFPP_FLOOD
59
60          actions = [parser.OFPActionOutput(out_port)]
61
62          if out_port != ofproto_v1_3.OFPP_FLOOD:
63              match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
64              self.add_flow(dp, 1, match, actions)
65
66          out = parser.OFPPacketOut(datapath=dp,
    buffer_id=msg.buffer_id,in_port=in_port,actions=actions, data=msg.data)
67          dp.send_msg(out)
```