SDNFV Lab2

Part1

1. How many OpenFlow headers with type “OFPT_FLOW_MOD” and command “OFPFC_ADD” are there among all the packets? What are the match fields and the corresponding actions in each “OFPT_FLOW_MOD” message? What are the Idle Timeout values for all flow rules on s1 in GUI?

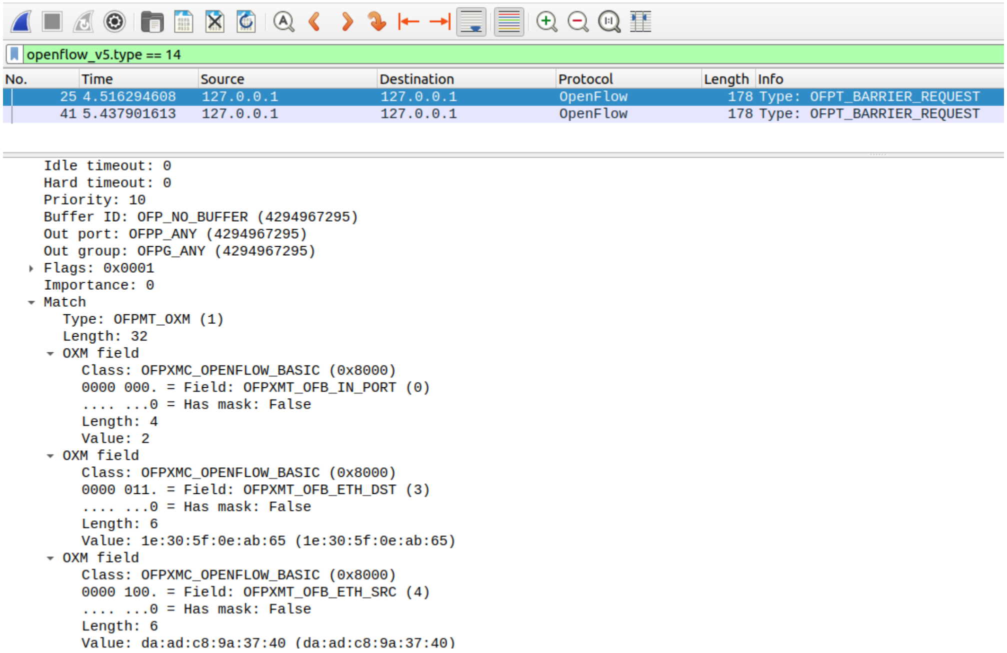
   There are 2 distinct “OFPT_FLOW_MOD” headers during the experiment.

Match fields	Actions	Timout values
In packets		

IN_PORT = 1
ETH_DST = da:ad:c8:9a:37:40
ETH_SRC = 1e:30:5f:0e:ab:65
	Output port = 2	10

IN_PORT = 2
ETH_DST = 1e:30:5f:0e:ab:65
ETH_SRC = da:ad:c8:9a:37:40
	Output port = 1	10

Other rules		
ETH_TYPE = lldp	Output port = CONTROLLER	0
ETH_TYPE = bddp	Output port = CONTROLLER	0
ETH_TYPE = ipv4	Output port = CONTROLLER	0
ETH_TYPE = arp	Output port = CONTROLLER	0

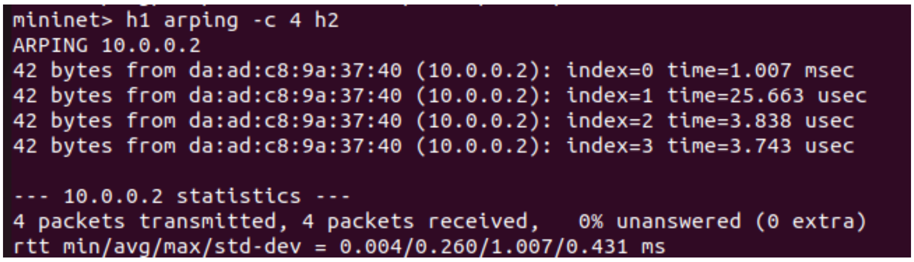


openflow_v5.type == 14

```
No.     Time         Source      Destination  Protocol    Length Info
     25 4.516294608  127.0.0.1   127.0.0.1    OpenFlow       178 Type: OFPT_BARRIER_REQUEST
     41 5.437901613  127.0.0.1   127.0.0.1    OpenFlow       178 Type: OFPT_BARRIER_REQUEST
```

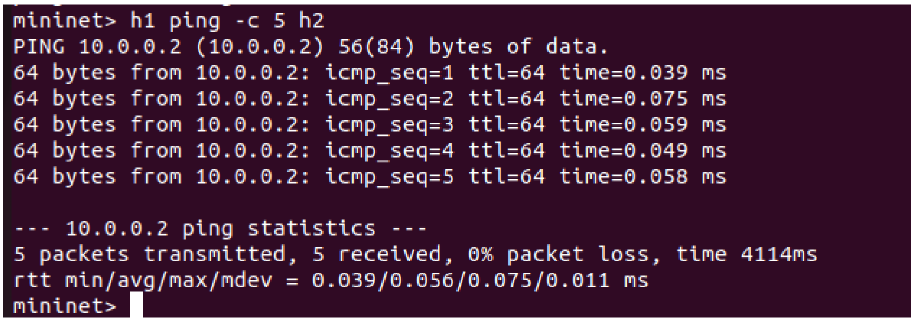
```
    Idle timeout: 0
    Hard timeout: 0
    Priority: 10
    Buffer ID: OFP_NO_BUFFER (4294967295)
    Out port: OFPP_ANY (4294967295)
    Out group: OFPG_ANY (4294967295)
  ▸ Flags: 0x0001
    Importance: 0
  ▾ Match
      Type: OFPMT_OXM (1)
      Length: 32
    ▾ OXM field
        Class: OFPXMC_OPENFLOW_BASIC (0x8000)
        0000 000. = Field: OFPXMT_OFB_IN_PORT (0)
        .... ...0 = Has mask: False
        Length: 4
        Value: 2
    ▾ OXM field
        Class: OFPXMC_OPENFLOW_BASIC (0x8000)
        0000 011. = Field: OFPXMT_OFB_ETH_DST (3)
        .... ...0 = Has mask: False
        Length: 6
        Value: 1e:30:5f:0e:ab:65 (1e:30:5f:0e:ab:65)
    ▾ OXM field
        Class: OFPXMC_OPENFLOW_BASIC (0x8000)
        0000 100. = Field: OFPXMT_OFB_ETH_SRC (4)
        .... ...0 = Has mask: False
        Length: 6
        Value: da:ad:c8:9a:37:40 (da:ad:c8:9a:37:40)
```

Part 2



```
mininet> h1 arping -c 4 h2
ARPING 10.0.0.2
42 bytes from da:ad:c8:9a:37:40 (10.0.0.2): index=0 time=1.007 msec
42 bytes from da:ad:c8:9a:37:40 (10.0.0.2): index=1 time=25.663 usec
42 bytes from da:ad:c8:9a:37:40 (10.0.0.2): index=2 time=3.838 usec
42 bytes from da:ad:c8:9a:37:40 (10.0.0.2): index=3 time=3.743 usec

--- 10.0.0.2 statistics ---
4 packets transmitted, 4 packets received,   0% unanswered (0 extra)
rtt min/avg/max/std-dev = 0.004/0.260/1.007/0.431 ms
```

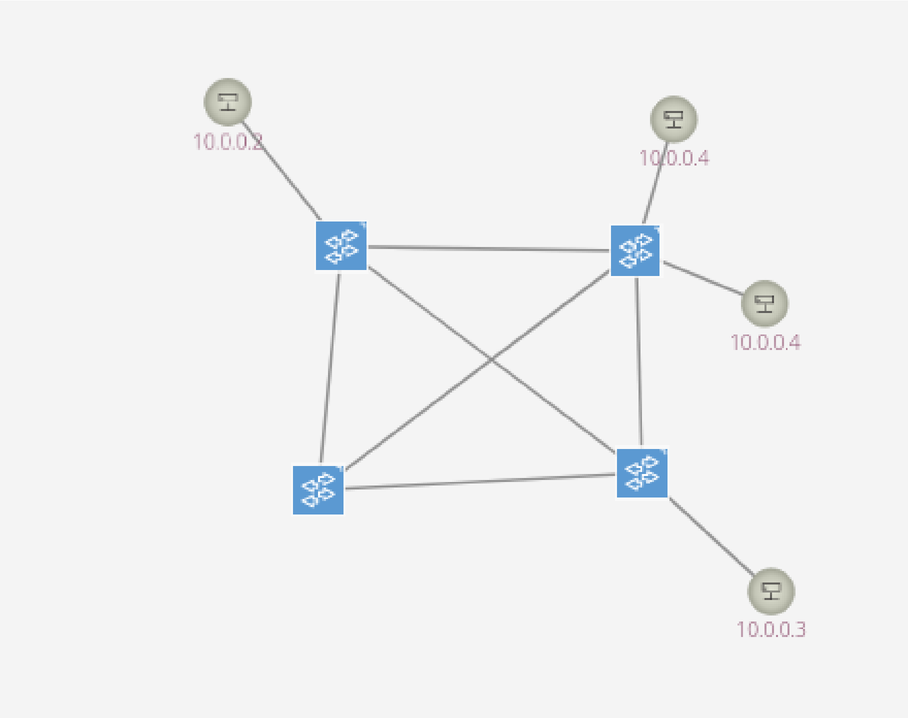


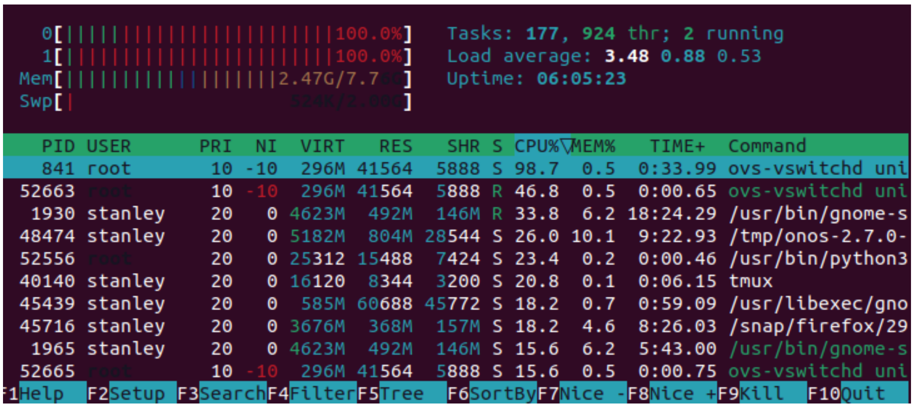
```
mininet> h1 ping -c 5 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.039 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.075 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.059 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.049 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.058 ms

--- 10.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4114ms
rtt min/avg/max/mdev = 0.039/0.056/0.075/0.011 ms
mininet> 
```

Part 3

I created the following topology. I created a rule that send all packets with ETH_TYPE = ARP to all the other ports. Once all the switches have this rule and a packet is sent, the swithes will endless sending packets to each other in a cycle, creating a broadcast storm.





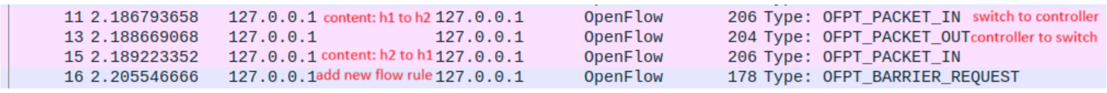
```
    0[||||||||||||||||||||||||100.0%]   Tasks: 177, 924 thr; 2 running
    1[||||||||||||||||||||||||100.0%]   Load average: 3.48 0.88 0.53
  Mem[||||||||||||||||||||2.47G/7.76G]   Uptime: 06:05:23
  Swp[|                   524K/2.00G]

    PID USER       PRI  NI  VIRT   RES    SHR S CPU%▽MEM%   TIME+  Command
    841 root        10 -10  296M 41564   5888 S 98.7  0.5  0:33.99 ovs-vswitchd uni
  52663 root        10 -10  296M 41564   5888 R 46.8  0.5  0:00.65 ovs-vswitchd uni
   1930 stanley     20   0 4623M  492M   146M R 33.8  6.2 18:24.29 /usr/bin/gnome-s
  48474 stanley     20   0 5182M  804M 28544 S 26.0 10.1  9:22.93 /tmp/onos-2.7.0-
  52556 root        20   0 25312 15488   7424 S 23.4  0.2  0:00.46 /usr/bin/python3
  40140 stanley     20   0 16120  8344   3200 S 20.8  0.1  0:06.15 tmux
  45439 stanley     20   0  585M 60688 45772 S 18.2  0.7  0:59.09 /usr/libexec/gno
  45716 stanley     20   0 3676M  368M   157M S 18.2  4.6  8:26.03 /snap/firefox/29
   1965 stanley     20   0 4623M  492M   146M S 15.6  6.2  5:43.00 /usr/bin/gnome-s
  52665 root        10 -10  296M 41564   5888 S 15.6  0.5  0:00.75 ovs-vswitchd uni
F1Help  F2Setup  F3SearchF4FilterF5Tree   F6SortByF7Nice -F8Nice +F9Kill   F10Quit
```

Part 4

1. In data plane, h1 send an ICMP packet to the s1
2. In the control plane, the packet meets the selector ETH_TYPE = ipv4 on the switch, hence forwarded to the controller.
3. The fwd app handles the packet, creating a rule using flowObjectiveService, and forward the packet to the destination host. (There are several steps in the app, such as detecting whether to drop the packet, or checking it is the edge...)

In the source code, the app calls the function to install the flow rule first, then forward packet to the destination. However, wireshark captures the packet received from h2 before capturing the request to add the flow rule. This is because the action of making new flow rules are asychronous, and the packet forwarding is slightly faster than adding new flow rules.



```
   11 2.186793658    127.0.0.1 content: h1 to h2 127.0.0.1    OpenFlow    206 Type: OFPT_PACKET_IN  switch to controller
   13 2.188669068    127.0.0.1                   127.0.0.1    OpenFlow    204 Type: OFPT_PACKET_OUTcontroller to switch
   15 2.189223352    127.0.0.1 content: h2 to h1 127.0.0.1    OpenFlow    206 Type: OFPT_PACKET_IN
   16 2.205546666    127.0.0.1add new flow rule  127.0.0.1    OpenFlow    178 Type: OFPT_BARRIER_REQUEST
```

Part 5

In this lab,

1. I learned how to install flow rules with json files using curl and GUI
2. I learned how to capture and inspect packets using wireshark
3. I traced the code the reactive forwarding app.
4. I observed how the controller and switch interacts.