Lab2 OpenFlow Protocol Observation and Flow Rule Installation

> Student ID: 313551097
> Student Name: 鄭淮薰

Part1: Answer Questions

There are 6 distinct "OFPT_FLOW_MOD" headers during the experiment.

Match Fields	Actions	Timeout Values
ETH_TYPE = ipv4(0x0800)	OUTPUT PORT = CONTROLLER	0
ETH_TYPE = arp(0x0806)	OUTPUT PORT = CONTROLLER	0
ETH_TYPE = lldp(0x088cc)	OUTPUT PORT = CONTROLLER	0
ETH_TYPE = bddp(0x8942)	OUTPUT PORT = CONTROLLER	0
IN_PORT = 1, ETH_DST = ae:a8:a3:d6:1e:d4, ETH_SRC = 12:d3:e0:36:b0:f1	OUTPUT PORT = 2	10
IN_PORT = 2, ETH_DST = 12:d3:e0:36:b0:f1, ETH_SRC = ae:a8:a3:d6:1e:d4	OUTPUT PORT = 1	10

Part2: Install Flow Rules

1. Install one flow rule to forward ARP packets Install the flow rule:

   ```
   curl -u onos:rocks -X POST \
   -H 'Content-Type: application/json' \
   -d @flows_s1-1_313551097.json \
   'http://localhost:8181/onos/v1/flows/of:0000000000000001'
   ```

   Verify the flow rule:

```
mininet> h1 arping h2
ARPING 10.0.0.2
42 bytes from 06:ab:fb:5b:7a:98 (10.0.0.2): index=0 time=6.583 usec
42 bytes from 06:ab:fb:5b:7a:98 (10.0.0.2): index=1 time=3.824 usec
42 bytes from 06:ab:fb:5b:7a:98 (10.0.0.2): index=2 time=14.882 usec
42 bytes from 06:ab:fb:5b:7a:98 (10.0.0.2): index=3 time=7.395 usec
42 bytes from 06:ab:fb:5b:7a:98 (10.0.0.2): index=4 time=14.635 usec
42 bytes from 06:ab:fb:5b:7a:98 (10.0.0.2): index=5 time=17.116 usec
42 bytes from 06:ab:fb:5b:7a:98 (10.0.0.2): index=6 time=22.152 usec
42 bytes from 06:ab:fb:5b:7a:98 (10.0.0.2): index=7 time=4.407 usec
42 bytes from 06:ab:fb:5b:7a:98 (10.0.0.2): index=8 time=22.774 usec
```

Figure 1: h1 arping h2 screenshot

2. Install two flow rules to forward IPv4 packets Install first flow rule:

```
curl -u onos:rocks -X POST \
-H 'Content-Type: application/json' \
-d @flows_s1-2_313551097.json \
'http://localhost:8181/onos/v1/flows/of:0000000000000001'
```

Install second flow rule:

```
curl -u onos:rocks -X POST \
-H 'Content-Type: application/json' \
-d @flows_s1-3_313551097.json \
'http://localhost:8181/onos/v1/flows/of:0000000000000001'
```

Verify the flow rules:

```
mininet> h1 ping 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=0.032 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.037 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.047 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.045 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.044 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.050 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.063 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.084 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=0.185 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=0.060 ms
```

Figure 2: h1 ping h2 screenshot

Part3: Create Topology with Broadcast Storm

Create topology I create a topology like Figure 3:

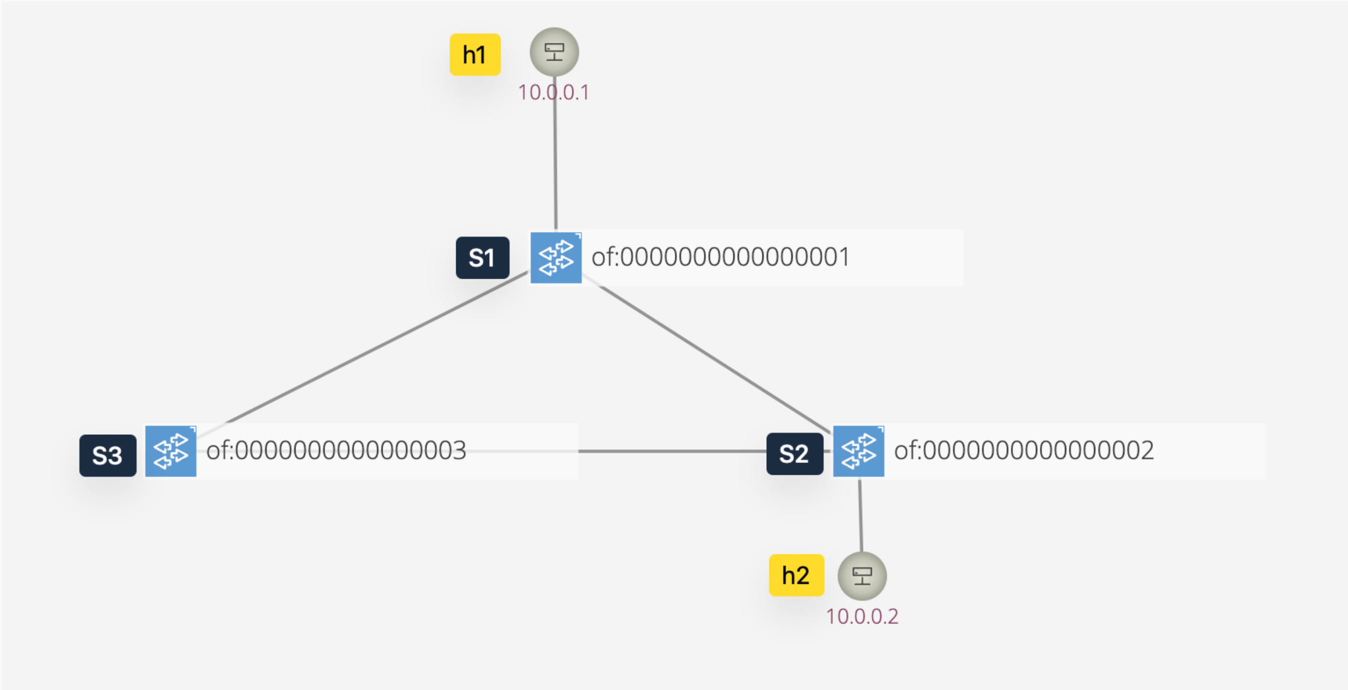

Figure 3: Broadcast storm topology

Install flow rule Also, I create flow rules to forward ARP packets to all ports on each switch, which will lead to a broadcast storm.

Scenario of Broadcast Storm

- When `h1` sends an ARP request (a broadcast message), `S1` will forward this ARP packet to both `S2` and `S3`.
- `S2` will receive the packet from `S1` and forward it to both `S3` and `h2`, and `S3` will also forward it back to `S1`.
- The packet keeps circulating around the loop (`S1`, `S2`, `S3`), and because the packet is a broadcast, each switch forwards it to its neighbors.

Result of Broadcast Storm

The broadcast storm can be observed in the following screenshot:

Figure 4: (Before) CPU Usage

Figure 5: (After) CPU Usage

Part4: Trace ReactiveForwarding

When `h1` pings `h2`, the following events occur (D: Date Plane, C: Control Plane):

1. D: `h1` sends an ARP request to `h2`. (Who has 10.0.0.2? Tell 10.0.0.1)
2. D → C: `S1` receives the ARP request and sends it to the controller because there is no corresponding flow rule. (PACKET_IN: IN_PORT=1)
3. C → D: The controller instructs `S1` to forward the request to all other ports, effectively broadcasting the ARP request in hopes of finding `h2`. (PACKET_OUT: OUTPUT=FLOOD)
4. D: `S1` forwards the ARP request to all other ports except the incoming port.
5. D: `h2` receives the ARP request and sends an ARP reply. (10.0.0.2 is at 72:69:b6:3c:cb:12)
6. D → C: `S1` receives the ARP reply and sends it to the controller because there is no corresponding flow rule. (PACKET_IN: IN_PORT=2)
7. C → D: The controller instructs `S1` to forward the reply to `h1`. (PACKET_OUT: OUTPUT=1)
8. D: `S1` forwards the ARP reply to `h1`.
9. D: `h1` receives the ARP reply and sends an ICMP echo request to `h2`.
10. D → C: `S1` receives the ICMP echo request and sends it to the controller because there is no corresponding flow rule. (PACKET_IN: IN_PORT=1)
11. C → D: The controller instructs `S1` to forward the ICMP echo request to `h2`. (PACKET_OUT: OUTPUT=2)
12. D: `S1` forwards the ICMP echo request to `h2`.
13. D: `h2` receives the first ICMP request.

What you've learned or solved

Through the Lab 2 exercises, I gained a deeper understanding of how flow rules work. In particular, the process of using Wireshark to observe packets in Part 4 gave me a clearer view of the interaction between the Data Plane and the Control Plane. Starting from Part 1, I learned how to capture packets with Wireshark, followed by learning how to install flow rules in Part 2. In Part 3, I gained insight into the broadcast storm issue, and finally, in Part 4, I observed the reactive forwarding process through Wireshark. This step-by-step learning approach helped me better understand how the OpenFlow Protocol operates.

Additionally, in the Part 4 exercise, I initially encountered some difficulties in understanding the packet flow. Later, I realized that I was only capturing packets on the Loopback: lo interface and missed the packets on the s1-eth1 and s1-eth2 interfaces, which prevented me from observing the packet exchange between h1 and h2. This experience taught me that packets may traverse different interfaces, so it is important to select the correct interface when using Wireshark for packet analysis.