

《现代交换原理》实验报告

实验名称 软件定义网络（SDN）实验

班 级 2022211305

学 号 2022211683、2022211124、2022211130

姓 名 张晨阳、梁维熙、金建名

指导教师 赵学达

目录

实验一 SDN 基本原理与 OpenFlow 协议分析.....	1
一、 实验目的.....	1
二、实验内容与实验步骤.....	1
2.1. 基础环境搭建.....	1
2.2. 建立 Mininet 与 Ryu 连接并抓包分析	1
1. 启动 Ryu 控制器	1
2. 利用 Mininet 工具构建网络拓扑.....	2
3. 利用抓包工具 Wireshark 分析 OpenFlow 协议	2
三、实验结果分析.....	4
(1) HELLO	4
(2) FEATURES_REQUEST	5
(3) FEATURES_REPLY	5
(4) MULTIPART_REQUEST.....	6
(5) MULTIPART_REPLY	6
(6) FLOW_MOD.....	6
(7) PACKET_IN	7
(8) PACKET_OUT	8
(9) PORT_STATUS	8
四、实验心得.....	10
实验二 OpenFlow 流表操作实践.....	11
一、 实验目的.....	11
二、 实验内容与实验步骤.....	11
2.1. 启动实验环境	11
1. 首先通过执行命令.....	11
2. 打开两个终端，在终端一中通过执行命令启动 ryu 控制器	11
2.2. 实现二层交换	13
2.3. 下发流表、流表匹配与数据包处理	14

1. 删除 s1 流表	14
2. 连通 $h1 \leftrightarrow h3$	14
三、 实验结果分析	17
四、 实验心得	19

实验一 SDN 基本原理与 OpenFlow 协议分析

一、实验目的

1. 了解软件定义网络控制平面和数据平面分离的原理。
2. 了解 mininet 网络仿真平台以及开源 SDN 控制器 Ryu 的基本原理和功能。
3. 掌握使用 mininet 平台自主构建一个较为复杂的网络拓扑，并建立与 Ryu 控制器的连接。
4. 掌握使用 Wireshark 工具抓取 OpenFlow 协议报文并进行分析。

二、实验内容与实验步骤

2.1. 基础环境搭建

1. 下载 VMware Workstation 虚拟机并安装。
2. 下载本实验提供的压缩包并根据实验指导书进行虚拟机环境的安装。

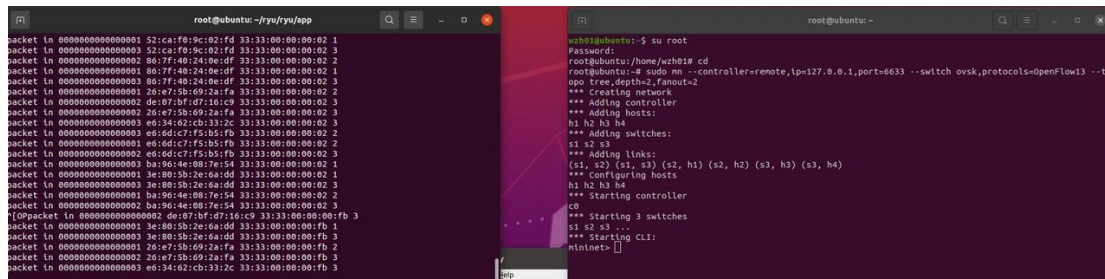
2.2. 建立 Mininet 与 Ryu 连接并抓包分析

1. 启动 Ryu 控制器

在本实验提供的虚拟机环境中内置了多种控制器应用程序，在我们的实验中使用的是 `simple_switch_13.py`，在 `ryu` 控制器对应的目录（本实验中的路径为 `root/ryu/ryu/app`）中使用命令：

```
ryu-manager simple_switch_13.py
```

即可启动控制器。

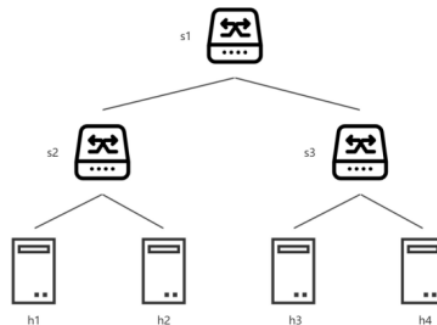


2. 利用 Mininet 工具构建网络拓扑

该工具为模拟网络中的设备，在本实验中我们使用命令：

```
sudo mn --controller=remote,ip=127.0.0.1,port=6633 --switch ovsk,protocols=OpenFlow13 --topo tree,depth=2,fanout=2
```

其中，--controller=remote,ip=127.0.0.1,port=6633 的含义为连接到本地运行的 Ryu 控制器，即步骤一中创建的控制器。--switch ovsk,protocols=OpenFlow13 的含义为使用 Open vSwitch，并指定使用 OpenFlow 1.3 协议。--topo tree,depth=2,fanout=2 的含义为创建一个树形拓扑网络，深度为 2，每个叶子交换机连接两台设备，创建的拓扑网络如图所示。

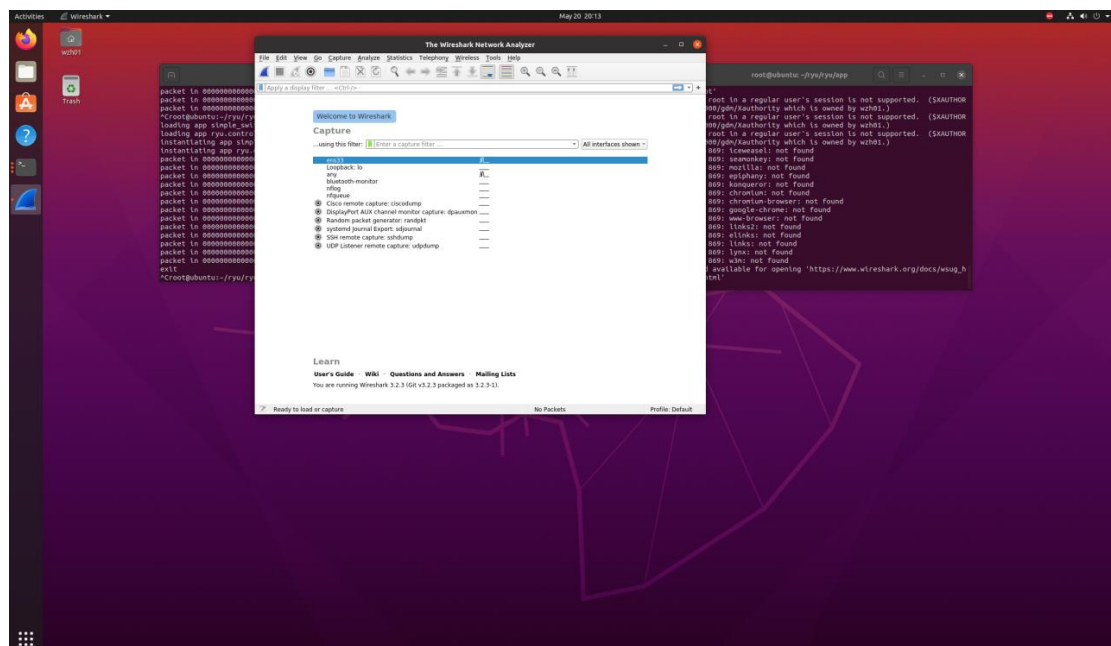


3. 利用抓包工具 Wireshark 分析 OpenFlow 协议

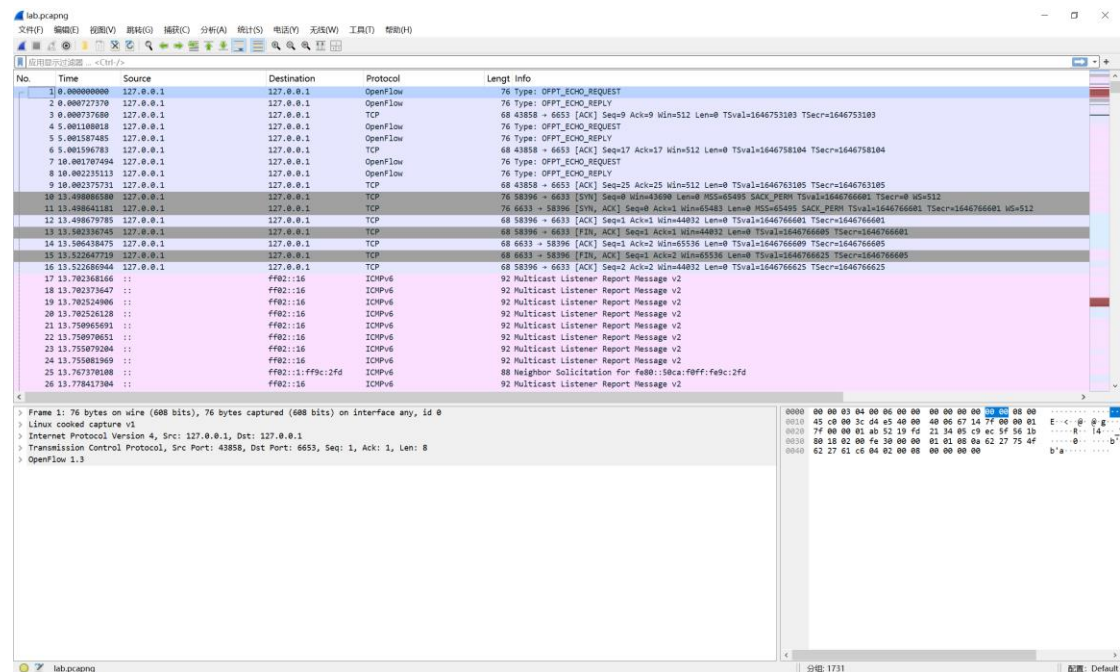
Wireshark 为 Linux 自带的抓包工具，在系统中我们可以通过命令：

```
sudo wireshark
```

启动 Wireshark 进行抓包，启动界面如下图所示：



在 Mininet 与 Ryu 建立连接之前，选择 Wireshark 的 any 模式进行抓包，抓包得到的结果如下：



三、实验结果分析

由于实验中存在三台路由器，一台作为三层交换机，两台作为二层交换机，二者仅存在一处不同，因此接下来先对三层交换机抓包进行说明：

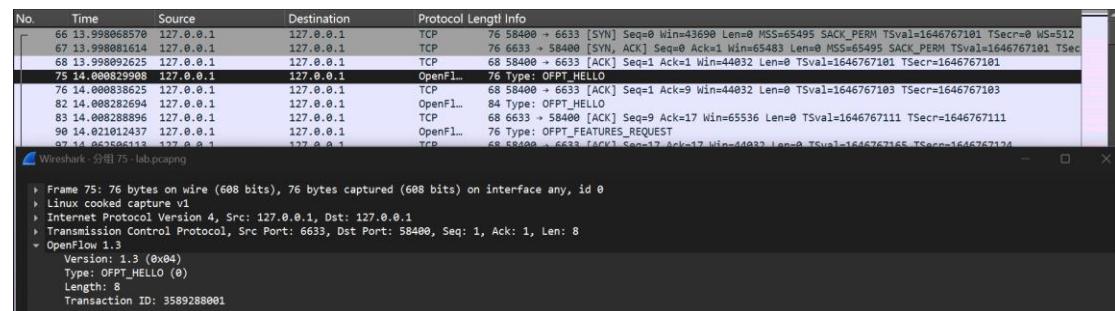
在实验中，三层交换机的端口为 58400，因此在 wireshark 中查找：

```
tcp.dstport == 58400 || tcp.srcport == 58400
```

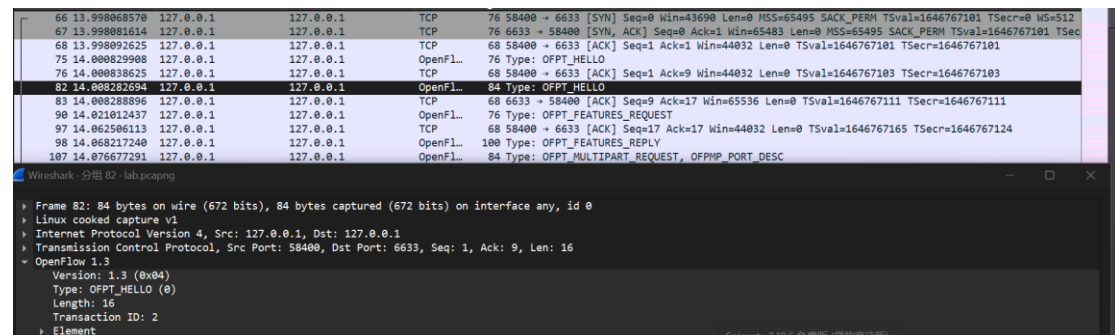
进行筛选。

(1) HELLO

控制器端口：6633 -> 交换机端口：58400



交换机端口：58400 -> 控制器端口：6633



这两个消息用于使交换机和控制器建立连接。

(2) FEATURES_REQUEST

控制器端口：6633 -> 交换机端口：58400

```
66 13.998066570 127.0.0.1 127.0.0.1 TCP 76 58400 -> 6633 [SYN] Seq=0 Win=43690 Len=0 MSS=65495 SACK_PERM TSval=1646767101 TSecr=0 WS=512
67 13.998061614 127.0.0.1 127.0.0.1 TCP 76 6633 -> 58400 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=1646767101 TSecr=0
68 13.998092625 127.0.0.1 127.0.0.1 TCP 68 58400 -> 6633 [ACK] Seq=1 Ack=1 Win=44032 Len=0 TSval=1646767101 TSecr=1646767103
75 14.000829908 127.0.0.1 127.0.0.1 OpenFl... 76 Type: OFPT_HELLO
76 14.000838625 127.0.0.1 127.0.0.1 TCP 68 58400 -> 6633 [ACK] Seq=1 Ack=9 Win=44032 Len=0 TSval=1646767103 TSecr=1646767103
82 14.008282694 127.0.0.1 127.0.0.1 OpenFl... 84 Type: OFPT_HELLO
83 14.008288896 127.0.0.1 127.0.0.1 TCP 68 6633 -> 58400 [ACK] Seq=9 Ack=17 Win=65536 Len=0 TSval=1646767111 TSecr=1646767111
90 14.021012437 127.0.0.1 127.0.0.1 OpenFl... 76 Type: OFPT_FEATURES_REQUEST
97 14.062506113 127.0.0.1 127.0.0.1 TCP 68 58400 -> 6633 [ACK] Seq=17 Ack=17 Win=44032 Len=0 TSval=1646767165 TSecr=1646767124
98 14.068217240 127.0.0.1 127.0.0.1 OpenFl... 100 Type: OFPT_FEATURES_REPLY
107 14.076677291 127.0.0.1 127.0.0.1 OpenFl... 84 Type: OFPT_MULTIPART_REQUEST, OFPMP_PORT_DESC
```

Wireshark - 分组 82 - lab.pcapng

```
> Frame 82: 84 bytes on wire (672 bits), 84 bytes captured (672 bits) on interface any, id 0
> Linux cooked capture v1
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 58400, Dst Port: 6633, Seq: 1, Ack: 9, Len: 16
  > OpenFlow 1.3
    Version: 1.3 (0x04)
    Type: OFPT_HELLO (0)
    Length: 16
    Transaction ID: 2
    Element
```

此消息表示控制器需要了解路由器的一些基本资源和配置，因此发送请求。

(3) FEATURES_REPLY

交换机端口：58400 -> 控制器端口：6633

```
83 14.008288896 127.0.0.1 127.0.0.1 TCP 68 6633 -> 58400 [ACK] Seq=9 Ack=17 Win=65536 Len=0 TSval=1646767111 TSecr=1646767111
90 14.021012437 127.0.0.1 127.0.0.1 OpenFl... 76 Type: OFPT_FEATURES_REQUEST
97 14.062506113 127.0.0.1 127.0.0.1 TCP 68 58400 -> 6633 [ACK] Seq=17 Ack=17 Win=44032 Len=0 TSval=1646767165 TSecr=1646767124
98 14.068217240 127.0.0.1 127.0.0.1 OpenFl... 100 Type: OFPT_FEATURES_REPLY
107 14.076677291 127.0.0.1 127.0.0.1 OpenFl... 84 Type: OFPT_MULTIPART_REQUEST, OFPMP_PORT_DESC
108 14.076688072 127.0.0.1 127.0.0.1 TCP 68 58400 -> 6633 [ACK] Seq=49 Ack=33 Win=44032 Len=0 TSval=1646767179 TSecr=1646767179
109 14.076706397 127.0.0.1 127.0.0.1 OpenFl... 148 Type: OFPT_FLOW_MOD
110 14.076710966 127.0.0.1 127.0.0.1 TCP 68 58400 -> 6633 [ACK] Seq=49 Ack=113 Win=44032 Len=0 TSval=1646767179 TSecr=1646767179
```

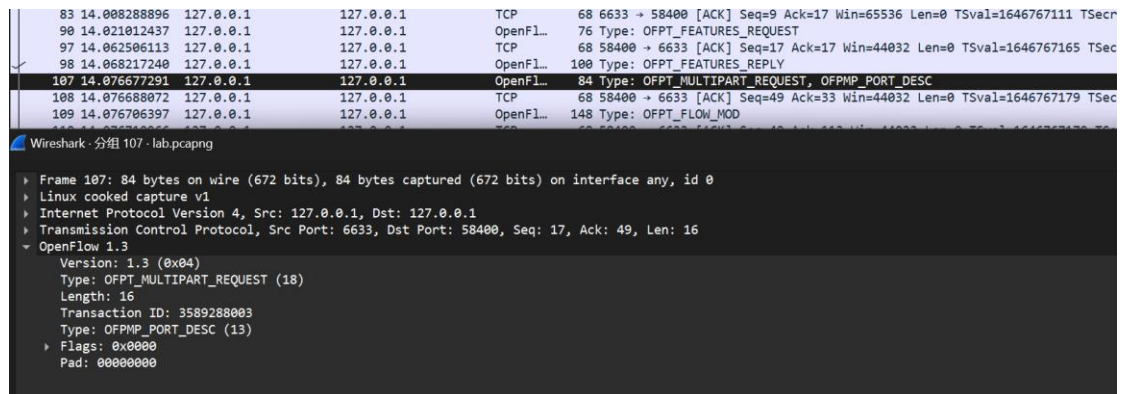
Wireshark - 分组 98 - lab.pcapng

```
> Frame 98: 100 bytes on wire (800 bits), 100 bytes captured (800 bits) on interface any, id 0
> Linux cooked capture v1
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 58400, Dst Port: 6633, Seq: 17, Ack: 17, Len: 32
  > OpenFlow 1.3
    Version: 1.3 (0x04)
    Type: OFPT_FEATURES_REPLY (6)
    Length: 32
    Transaction ID: 3589288002
    datapath_id: 0x0000000000000001
    n_buffers: 0
    n_tables: 254
    auxiliary_id: 0
    Pad: 0
    capabilities: 0x00000004f
    Reserved: 0x00000000
```

此消息是对消息 FEATURES_REPLY 的回复，也就是交换机向控制器发送控制器请求了解的信息。

(4) MULTIPART_REQUEST

控制器端口: 6633 -> 交换机端口: 58400



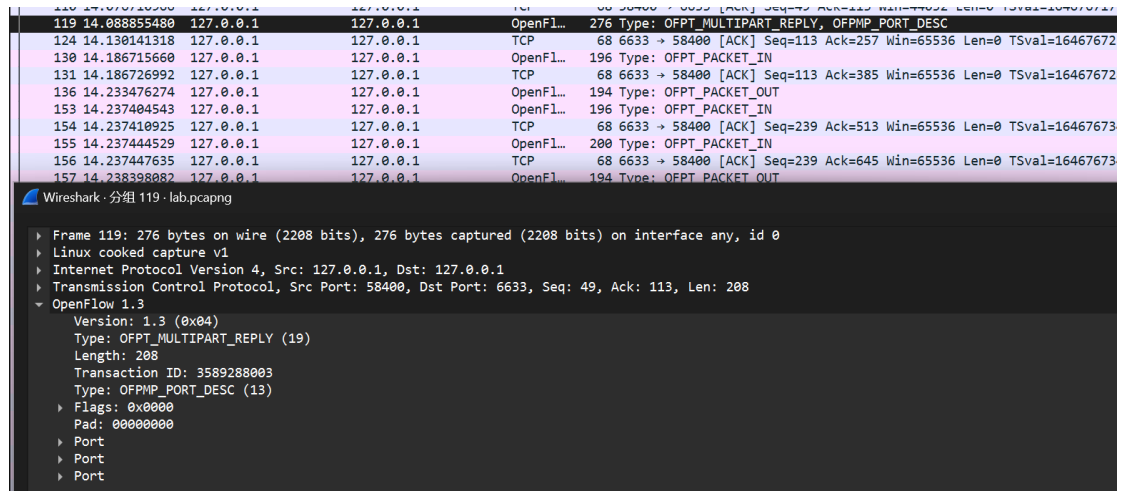
The image shows a Wireshark packet capture of a network traffic between a controller and a switch. The packet list shows several TCP and OpenFlow packets. The selected packet is a MULTIPART_REQUEST (OFPT_MULTIPART_REQUEST) with sequence number 17, sent from the controller (127.0.0.1) to the switch (127.0.0.1) on port 6633. The packet details pane shows the following information:

- Version: 1.3 (0x04)
- Type: OFPT_MULTIPART_REQUEST (18)
- Length: 16
- Transaction ID: 3589288003
- Type: OFPMP_PORT_DESC (13)
- Flags: 0x0000
- Pad: 00000000

此消息表示控制器需要进一步了解交换机的各种详细信息, 因此发送的请求。

(5) MULTIPART_REPLY

交换机端口: 58400 -> 控制器端口: 6633



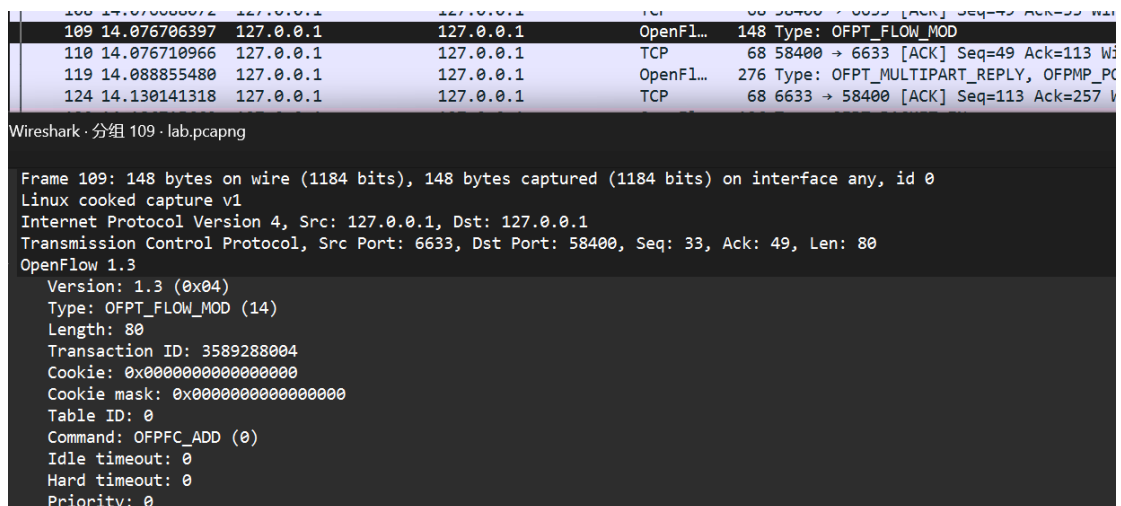
The image shows a Wireshark packet capture of a network traffic between a switch and a controller. The packet list shows several TCP and OpenFlow packets. The selected packet is a MULTIPART_REPLY (OFPT_MULTIPART_REPLY) with sequence number 49, sent from the switch (127.0.0.1) to the controller (127.0.0.1) on port 58400. The packet details pane shows the following information:

- Version: 1.3 (0x04)
- Type: OFPT_MULTIPART_REPLY (19)
- Length: 208
- Transaction ID: 3589288003
- Type: OFPMP_PORT_DESC (13)
- Flags: 0x0000
- Pad: 00000000
- Port
- Port
- Port

对消息 MULTIPART_REQUEST 的回复。告知相关信息。

(6) FLOW_MOD

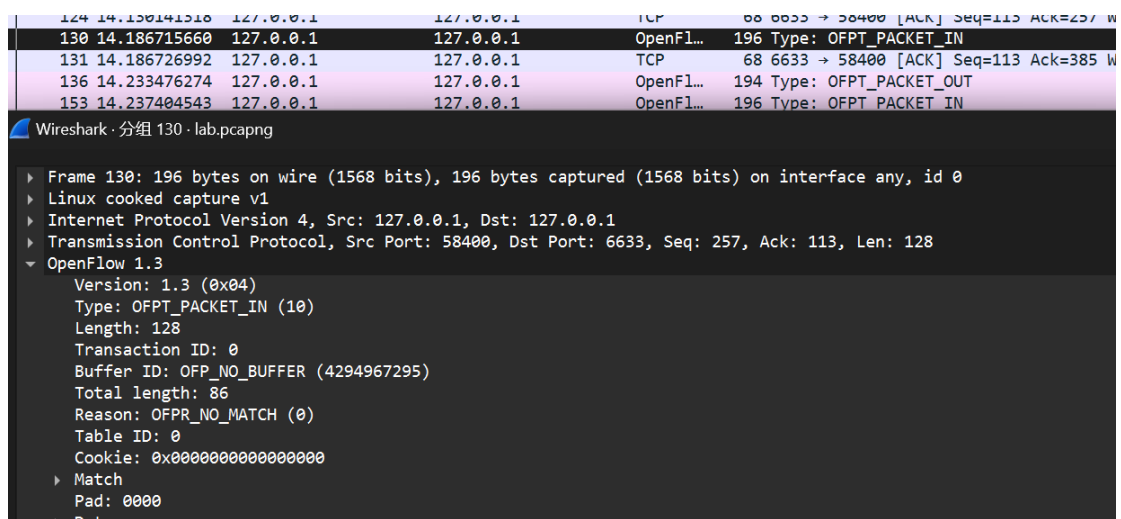
控制器端口: 6633 -> 交换机端口: 58400



控制器通过此消息指导交换机进行数据转发。

(7) PACKET_IN

交换机端口：58400 -> 控制器端口：6633



有两种情况：

- I：交换机查找流表，发现没有匹配条目时，向控制器询问如何转发；
- II：有匹配条目但对应的 action 是 OUTPUT=CONTROLLER 时。

(8) PACKET_OUT

控制器端口：6633 -> 交换机端口：58400

124	14.130141318	127.0.0.1	127.0.0.1	TCP	68	6633 → 58400 [ACK] Seq=113 Ack=2
130	14.186715660	127.0.0.1	127.0.0.1	OpenFl...	196	Type: OFPT_PACKET_IN
131	14.186726992	127.0.0.1	127.0.0.1	TCP	68	6633 → 58400 [ACK] Seq=113 Ack=3
136	14.233476274	127.0.0.1	127.0.0.1	OpenFl...	194	Type: OFPT_PACKET_OUT
153	14.237404543	127.0.0.1	127.0.0.1	OpenFl...	196	Type: OFPT_PACKET_IN
154	14.237410925	127.0.0.1	127.0.0.1	TCP	68	6633 → 58400 [ACK] Seq=239 Ack=5
155	14.237444529	127.0.0.1	127.0.0.1	OpenFl...	200	Type: OFPT_PACKET_IN
156	14.237447635	127.0.0.1	127.0.0.1	TCP	68	6633 → 58400 [ACK] Seq=239 Ack=6

Wireshark · 分组 136 · lab.pcapng

- ▶ Frame 136: 194 bytes on wire (1552 bits), 194 bytes captured (1552 bits) on interface any, id 0
- ▶ Linux cooked capture v1
- ▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
- ▶ Transmission Control Protocol, Src Port: 6633, Dst Port: 58400, Seq: 113, Ack: 385, Len: 126
- ▼ OpenFlow 1.3
 - Version: 1.3 (0x04)
 - Type: OFPT_PACKET_OUT (13)
 - Length: 126
 - Transaction ID: 3589288005
 - Buffer ID: OFP_NO_BUFFER (4294967295)
 - In port: 1
 - Actions length: 16
 - Pad: 000000000000
 - ▶ Action
 - ▶ Data

用于控制器指导交换机对数据包处理。

(9) PORT_STATUS

在实验过程中，我们在端口为 58400 的交换机没有发现存在这种包，但在其他两个交换机（端口分别为：58416 和 58429）的包中发现存在。

58416:

98	14.068217240	127.0.0.1	127.0.0.1	OpenFl...	100	Type: OFPT_FEATURES_REPLY
99	14.068890757	127.0.0.1	127.0.0.1	OpenFl...	148	Type: OFPT_PORT_STATUS
100	14.068945873	127.0.0.1	127.0.0.1	OpenFl...	100	Type: OFPT_FEATURES_REPLY
101	14.069446540	127.0.0.1	127.0.0.1	OpenFl...	148	Type: OFPT_PORT_STATUS
102	14.069602758	127.0.0.1	127.0.0.1	OpenFl...	100	Type: OFPT_FEATURES_REPLY

Wireshark · 分组 99 · lab.pcapng

- ▶ Frame 99: 148 bytes on wire (1184 bits), 148 bytes captured (1184 bits) on interface any, id 0
- ▶ Linux cooked capture v1
- ▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
- ▶ Transmission Control Protocol, Src Port: 58416, Dst Port: 6633, Seq: 17, Ack: 17, Len: 80
- ▼ OpenFlow 1.3
 - Version: 1.3 (0x04)
 - Type: OFPT_PORT_STATUS (12)
 - Length: 80
 - Transaction ID: 0
 - Reason: OFPPR_MODIFY (2)
 - Pad: 000000000000
 - ▶ Port

58426:

No.	Time	Source	Destination	Protocol	Length	Info
100	14.068945873	127.0.0.1	127.0.0.1	OpenFl...	100	Type: OFPT_FEATURES_REPLY
101	14.069446540	127.0.0.1	127.0.0.1	OpenFl...	148	Type: OFPT_PORT_STATUS
102	14.069602758	127.0.0.1	127.0.0.1	OpenFl...	100	Type: OFPT_FEATURES_REPLY
103	14.074075468	127.0.0.1	127.0.0.1	TCP	100	[TCP Retransmission] 58426 → 6633 [PSH, A
104	14.074079365	127.0.0.1	127.0.0.1	TCP	100	[TCP Retransmission] 58416 → 6633 [PSH, A
105	14.074100005	127.0.0.1	127.0.0.1	TCP	80	6633 → 58426 [ACK] Seq=17 Ack=129 Win=655:
106	14.074103822	127.0.0.1	127.0.0.1	TCP	80	6633 → 58416 [ACK] Seq=17 Ack=129 Win=655:
107	14.076677291	127.0.0.1	127.0.0.1	OpenFl...	84	Type: OFPT_MULTIPART_REQUEST, OFPMP_PORT_I
108	14.076680073	127.0.0.1	127.0.0.1	TCP	68	58426 → 6633 [ACK] Seq=40 Ack=33 Win=4193

Wireshark · 分组 101 · lab.pcapng

▶ Frame 101: 148 bytes on wire (1184 bits), 148 bytes captured (1184 bits) on interface any, id 0

▶ Linux cooked capture v1

▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

▶ Transmission Control Protocol, Src Port: 58426, Dst Port: 6633, Seq: 17, Ack: 17, Len: 80

▼ OpenFlow 1.3

Version: 1.3 (0x04)

Type: OFPT_PORT_STATUS (12)

Length: 80

Transaction ID: 0

Reason: OFPPR_MODIFY (2)

Pad: 00000000000000

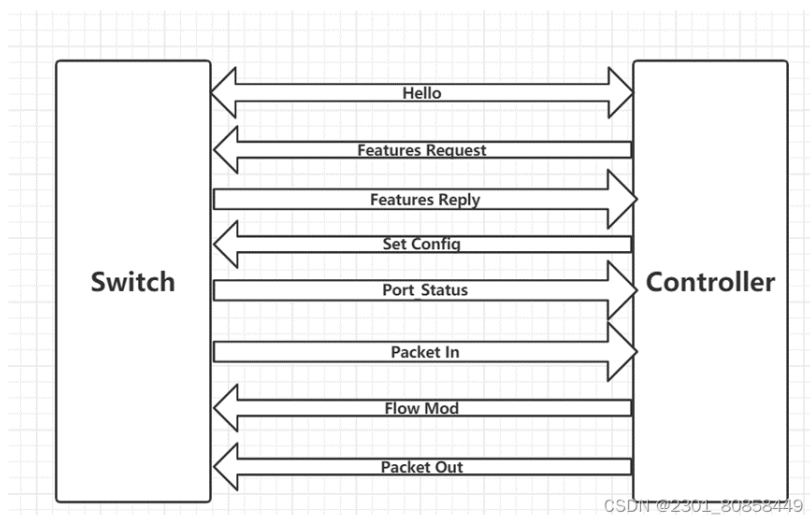
▶ Port

这种包的作用为当交换机端口发生改变时，告知控制器相应的端口状态。

在我们分析之后，发现只有两个二层交换机才存在这种包的原因是：

控制器需要直接与三台交换机连接，但我们的拓扑结构中还需要两个二层交换机各自选出一个空闲端口与三层交换机相连，这就导致每个二层交换机中各有一个端口的状态发生变化，因此向控制器发送这种包。

交互图：



四、实验心得

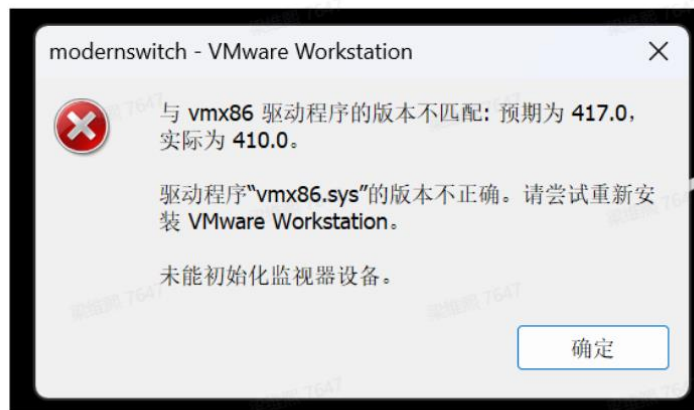
我们在使用 VMware Workstation 时遇到了以下问题：

1

Could not probe: failed to launch helper process.

卸载，装最新版本

2



重启电脑

但经过资料查阅等方法，都成功解决了。

本次实验我们学习了 SDN 和 OpenFlow 协议的基本原理，通过使用 Ryu 和 mininet 这些与网络编程相关的 python 包构建出一个虚拟网络拓扑，再使用 wireshark 对实验网络包进行全程追踪。

实验二 OpenFlow 流表操作实践

一、实验目的

1. 了解 OpenFlow 协议的基本原理及流表结构。
2. 熟悉流表项的增删改查操作及其对数据转发的影响。
3. 通过主机间连通状况验证流表行为。

二、实验内容与实验步骤

2.1. 启动实验环境

1. 首先通过执行命令

```
sudo apt install curl
```

安装开源命令行工具 curl，该工具通过各类网络协议传输数据，支持 get、post、delete 等 http 方法，本实验中用于进行下发、查看、删除流表。

2. 打开两个终端，在终端一中通过执行命令启动 ryu 控制器

```
ryu-manager ryu.app.ofctl_rest ryu.app.simple_switch_13
```

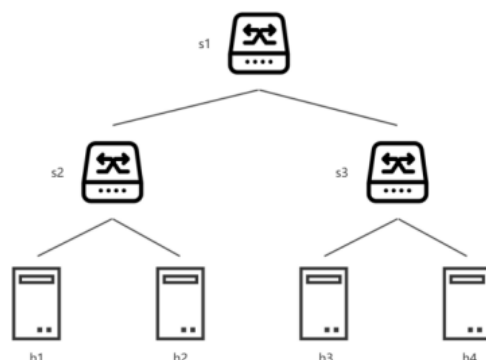
```
root@ubuntu:~# sudo apt install curl
Reading package lists... Done
Building dependency tree
Reading state information... Done
curl is already the newest version (7.68.0-1ubuntu2.25).
0 upgraded, 0 newly installed, 0 to remove and 62 not upgraded.
root@ubuntu:~# cd ryu/ryu/app
root@ubuntu:~/ryu/ryu/app# ryu-manager ryu.app.ofctl_rest ryu.app.simple_switch_13
loading app ryu.app.ofctl_rest
loading app ryu.app.simple_switch_13
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
creating context wsgi
instantiating app ryu.app.ofctl_rest of RestStatsApi
instantiating app ryu.app.simple_switch_13 of SimpleSwitch13
instantiating app ryu.controller.ofp_handler of OFPHandler
(3338) wsgi starting up on http://0.0.0.0:8080
```

在终端二中执行以下命令创建一个二层高的、每个叶子节点有两台主机的树形拓扑网络：

```
sudo mn --controller remote --topo tree,depth=2
```

其中，--topo 中的参数 tree 表示创建的网络为树形，depth=2 表示树形网络

的高度为 2，其网络拓扑结构如图所示。



然后，我们使用以下命令检查拓扑网络中各个交换机与主机的连接情况：

```
net
```

得到的结果如下：

```
root@ubuntu: ~  
wzh01@ubuntu:~$ su root  
Password:  
root@ubuntu:/home/wzh01# cd  
root@ubuntu:~# sudo mn --controller remote --topo tree,depth=2  
*** Creating network  
*** Adding controller  
Connecting to remote controller at 127.0.0.1:6653  
*** Adding hosts:  
h1 h2 h3 h4  
*** Adding switches:  
s1 s2 s3  
*** Adding links:  
(s1, s2) (s1, s3) (s2, h1) (s2, h2) (s3, h3) (s3, h4)  
*** Configuring hosts  
h1 h2 h3 h4  
*** Starting controller  
c0  
*** Starting 3 switches  
s1 s2 s3 ...  
*** Starting CLI:  
mininet> net  
h1 h1-eth0:s2-eth1  
h2 h2-eth0:s2-eth2  
h3 h3-eth0:s3-eth1  
h4 h4-eth0:s3-eth2  
s1 lo: s1-eth1:s2-eth3 s1-eth2:s3-eth3  
s2 lo: s2-eth1:h1-eth0 s2-eth2:h2-eth0 s2-eth3:s1-eth1  
s3 lo: s3-eth1:h3-eth0 s3-eth2:h4-eth0 s3-eth3:s1-eth2  
c0  
mininet> |
```

输出表明 h1、h2 与 s2 相连，h3、h4 与 s3 相连，s2、s3 与 s1 相连，与预期树形拓扑网络相同。

2.2. 实现二层交换

首先，我们使用以下命令进行各个节点之间的通信测试

pingall

得到如下结果:

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
mininet>
```

结果说明所有主机之间可以正常通信，这是因为 Ryu 的 `simple_switch_13` 中已经实现了二层自学习交换功能，无需额外配置。在新建的终端三中分别使用以下两种命令查看 s1 的流表，得到结果：

```
curl http://127.0.0.1:8080/stats/flow/1
```

[illegible]

```
sudo ovs-ofctl -O OpenFlow13 dump-flows s1
```

[illegible]

我们可以发现 s1 已经自动学习了 MAC 地址。

2.3. 下发流表、流表匹配与数据包处理

1. 删除 s1 流表

我们在终端中执行以下命令删除 s1 中的流表：

```
curl -X POST -d '{"dpid":1,"match":{}}' http://127.0.0.1:8080/stats/flowentry/delete
```

然后使用以下命令再次查看 s1 的流表：

```
root@ubuntu:~# curl -X POST -d '{"dpid":1,"match":{}}' http://127.0.0.1:8080/stats/flowentry/delete
root@ubuntu:~# curl http://127.0.0.1:8080/stats/flow/1
{"1": []}root@ubuntu:~#
```

输出表明 s1 的流表成功删除。然后我们再次使用 pingall 命令进行主机之间的通信测试，得到结果如下：

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 X X
h2 -> h1 X X
h3 -> X X h4
h4 -> X X h3
*** Results: 66% dropped (4/12 received)
mininet> 
```

结果说明此时 h1 与 h2 相通，h3 与 h4 相通。

2. 连通 h1<->h3

首先，我们在终端二中使用以下两条命令查看 h1 和 h3 的 MAC 地址：

```
h1 ifconfig
```

```
mininet> h1 ifconfig
h1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.0.0.1 netmask 255.0.0.0 broadcast 10.255.255.255
    inet6 fe80::5cd9:d3ff:fef9:6dc7 prefixlen 64 scopeid 0x20<link>
    ether 5e:d9:d3:f9:6d:c7 txqueuelen 1000 (Ethernet)
    RX packets 123 bytes 12104 (12.1 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 36 bytes 2504 (2.5 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

mininet> 
```

h3 ifconfig

```
mininet> h3 ifconfig
h3-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.0.0.3 netmask 255.0.0.0 broadcast 10.255.255.255
    inet6 fe80::543d:87ff:fe26:3e2d prefixlen 64 scopeid 0x20<link>
    ether 56:3d:87:26:3e:2d txqueuelen 1000 (Ethernet)
    RX packets 132 bytes 13153 (13.1 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 37 bytes 2574 (2.5 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

然后，我们使用 curl 工具给 s1 添加一个流表，流表的源地址为 h1，目的地址为 h3:

```
curl -X POST -d '{
  "dpid": 1,
  "priority": 100,
  "match": {
    "in_port": 1,
    "dl_src": "5e:d9:d3:f9:6d:c7",
    "dl_dst": "56:3d:87:26:3e:2d"
  },
  "actions": [{"type": "OUTPUT", "port": 2}]
}' http://127.0.0.1:8080/stats/flowentry/add
```

终端一中的 ryu 终端输出如下:

```
127.0.0.1 - - [19/May/2025 13:03:34] "POST /stats/flowentry/add HTTP/1.1" 200 115 0.000768
packet in 0000000000000002 5e:d9:d3:f9:6d:c7 ff:ff:ff:ff:ff:ff 1
packet in 0000000000000002 5e:d9:d3:f9:6d:c7 ff:ff:ff:ff:ff:ff 1
packet in 0000000000000002 5e:d9:d3:f9:6d:c7 ff:ff:ff:ff:ff:ff 1
packet in 0000000000000002 5e:d9:d3:f9:6d:c7 ff:ff:ff:ff:ff:ff 1
packet in 0000000000000002 5e:d9:d3:f9:6d:c7 ff:ff:ff:ff:ff:ff 1
packet in 0000000000000002 e6:aa:93:25:8e:d8 ff:ff:ff:ff:ff:ff 2
packet in 0000000000000002 e6:aa:93:25:8e:d8 ff:ff:ff:ff:ff:ff 2
packet in 0000000000000002 f6:0a:bc:7b:13:15 33:33:00:00:00:02 3
packet in 0000000000000002 e6:aa:93:25:8e:d8 ff:ff:ff:ff:ff:ff 2
packet in 0000000000000002 e6:aa:93:25:8e:d8 ff:ff:ff:ff:ff:ff 2
packet in 0000000000000002 e6:aa:93:25:8e:d8 ff:ff:ff:ff:ff:ff 2
packet in 0000000000000003 56:3d:87:26:3e:2d ff:ff:ff:ff:ff:ff 1
packet in 0000000000000003 56:3d:87:26:3e:2d ff:ff:ff:ff:ff:ff 1
packet in 0000000000000003 56:3d:87:26:3e:2d ff:ff:ff:ff:ff:ff 1
packet in 0000000000000003 56:3d:87:26:3e:2d ff:ff:ff:ff:ff:ff 1
packet in 0000000000000003 56:3d:87:26:3e:2d ff:ff:ff:ff:ff:ff 1
packet in 0000000000000003 d6:df:4a:9f:46:e8 ff:ff:ff:ff:ff:ff 2
packet in 0000000000000003 d6:df:4a:9f:46:e8 ff:ff:ff:ff:ff:ff 2
packet in 0000000000000003 d6:df:4a:9f:46:e8 ff:ff:ff:ff:ff:ff 2
packet in 0000000000000003 d6:df:4a:9f:46:e8 ff:ff:ff:ff:ff:ff 2
packet in 0000000000000003 d6:df:4a:9f:46:e8 ff:ff:ff:ff:ff:ff 2
packet in 0000000000000002 e6:aa:93:25:8e:d8 33:33:00:00:00:02 2
```

然后，我们使用以下命令给 s1 分别添加 h1 广播与 h3->h1 的流表：

```
curl -X POST -d '{
  "dpid": 1,
  "priority": 100,
  "match": {
    "in_port": 1,
    "dl_src": "5e:d9:d3:f9:6d:c7",
    "dl_dst": "ff:ff:ff:ff:ff:ff"
  },
  "actions": [{"type": "OUTPUT", "port": 2}]
}' http://127.0.0.1:8080/stats/flowentry/add

curl -X POST -d '{
  "dpid": 1,
  "priority": 100,
  "match": {
    "in_port": 2,
    "dl_src": "56:3d:87:26:3e:2d",
    "dl_dst": "5e:d9:d3:f9:6d:c7"
  },
  "actions": [{"type": "OUTPUT", "port": 1}]
}' http://127.0.0.1:8080/stats/flowentry/add
```

此时，我们再次进行 pingall 操作，可以发现 h1 与 h3 之间能够通信。

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 X
h2 -> h1 X X
h3 -> h1 X h4
h4 -> X X h3
*** Results: 50% dropped (6/12 received)
mininet>
```

再次检查 s1 中的流表项：

```
root@ubuntu:~# sudo ovs-ofctl -O OpenFlow13 dump-flows s1
cookie=0x0, duration=855.028s, table=0, n_packets=3, n_bytes=238, priority=100,in_port="s1-eth1",dl_src=5e:d9:d3:f9:6d:c7,dl_dst=56:3d:87:26:3e:2d actions=output:"s1-eth2"
cookie=0x0, duration=547.796s, table=0, n_packets=0, n_bytes=0, priority=100,in_port="s1-eth1",dl_src=ff:ff:ff:ff:ff:ff,dl_dst=56:3d:87:26:3e:2d actions=output:"s1-eth2"
cookie=0x0, duration=460.094s, table=0, n_packets=0, n_bytes=0, priority=100,in_port="s1-eth1",dl_src=56:3d:87:26:3e:2d,dl_dst=ff:ff:ff:ff:ff:ff actions=output:"s1-eth2"
cookie=0x0, duration=395.511s, table=0, n_packets=0, n_bytes=0, priority=100,in_port="s1-eth1",dl_src=56:3d:87:26:3e:2d,dl_dst=5e:d9:d3:f9:6d:c7 actions=output:"s1-eth2"
cookie=0x0, duration=210.028s, table=0, n_packets=34, n_bytes=1428, priority=100,in_port="s1-eth1",dl_src=5e:d9:d3:f9:6d:c7,dl_dst=ff:ff:ff:ff:ff:ff actions=output:"s1-eth2"
cookie=0x0, duration=179.253s, table=0, n_packets=0, n_bytes=0, priority=100,in_port=3,dl_src=56:3d:87:26:3e:2d,dl_dst=5e:d9:d3:f9:6d:c7 actions=output:"s1-eth1"
cookie=0x0, duration=55.468s, table=0, n_packets=4, n_bytes=280, priority=100,in_port="s1-eth2",dl_src=56:3d:87:26:3e:2d,dl_dst=5e:d9:d3:f9:6d:c7 actions=output:"s1-eth1"
root@ubuntu:~#
```

三、实验结果分析

两种查看流表的方式有什么不同？这些流表表示了什么信息？

1. 输出格式方面

```
sudo ovs-ofctl -O OpenFlow13 dump-flows s1
```

输出格式较为直观和简洁，通常以文本形式直接显示流表的详细信息，包括流表的优先级、匹配字段（如源地址、目的地址、协议类型等）、动作（如转发到某个端口、丢弃等）等，这些信息是按照 OpenFlow 协议规定的格式直接从交换机中读取并展示出来的，方便网络管理员快速查看和理解流表的配置情况。

```
curl http://127.0.0.1:8080/stats/flow/1
```

输出格式是 JSON 格式。JSON 格式是一种轻量级的数据交换格式，它以键值对的形式组织数据。在 Mininet 中，当使用 Ryu 控制器等基于 RESTAPI 的控制器时，通过该命令获取的流表信息会以 JSON 格式返回。例如，它可能会包含多个键，如“flows”键下包含了流表的详细信息，其中包括流表的 id、cookie、priority 等字段，以及每个流表的 match 条件和 actions 等，这种格式便于程序进行解析和处理，适合用于开发自动化脚本或与其他基于 JSON 的系统进行交互。

2. 获取流表的途径方面

```
sudo ovs-ofctl -O OpenFlow13 dump-flows s1
```

通过 OpenFlow 协议直接与交换机进行通信来获取流表信息的。它利用了 OpenFlow 协议中的消息机制，向交换机发送请求消息，交换机接收到请求后，会将流表信息以 OpenFlow 消息的形式返回，这种方式是基于 OpenFlow 协议的直接交互，不依赖于其他中间件或控制器。

```
curl http://127.0.0.1:8080/stats/flow/1
```

通过与控制器的交互来获取流表信息的。控制器会维护一个流表的视图，当收到该 curl 请求时，控制器会从自己的内部数据结构中获取流表信息，并将其转换为 JSON 格式返回给请求方。这种方式依赖于控制器的正常运行以及控制器与交换机之间的通信，控制器起到了中间代

理的作用，它从交换机获取流表信息后进行存储和管理，然后通过 RESTAPI 向外部提供这些信息。

请解释这种现象的原因。

由于创建的拓扑网络为树形结构，所以与 s2 相连的主机要与 s3 相连的主机需要经过 s1 进行转发，但是我们通过命令将 s1 中的流表删除，此时 s1 不知道如何进行通信包的转发，所以两侧的主机不能够进行相连。但是与 s2 直接相连的两台主机只需要经过 s2 进行转发即可进行通信，所以二者之间能够 ping 通；与 s3 直接相连的两台主机能够 ping 通原因相同。

此时尝试 ping，应 ping 不通。为什么？

ping 命令通过发送 ICMP 包到目标地址并接受从目标地址发回的 ICMP 包进行分析判断两个地址之间能否连通，我们在 s1 中配置了 h1->h3 的流表，但是并未配置 h3->h1 的流表，即使 h3 通过 ARP 的方式获取到了 h1 的 MAC 地址，s1 交换机不会将 h3 回复的 ICMP 包转发给 h1。

为什么仅设置了 h1 广播的流表项，未设置 h3 广播的流表项，h3->h1 仍能 ping 通？

h3 首先使用 ARP 广播包请求 h1 的 MAC 地址，虽然我们在 s1 中没有配置 h3 广播的流表项，但是在 SDN 环境中，交换机的默认行为通常是由控制器配置的流表决定的。如果流表中没有明确禁止某种类型的帧（如广播帧）的转发，交换机会根据帧的类型（如广播帧）进行默认处理。对于广播帧，交换机会将其转发到所有其他端口，以确保所有设备都能接收到该帧。所以该 ARP 包能够正确转发至 h1，使 h1 能够构建 ARP 回复包告诉 h3 自己的 MAC 地址。

四、实验心得

遇到的问题与解决方案

1. 在配置流表时尝试使用其他端口（例如端口 3）作为目标端口连通失败：

```
root@ubuntu:~# sudo ovs-ofctl -O OpenFlow3 dump-flows s1
cookie=0x0, duration=55.928s, table=0, n_packets=3, n_bytes=238, priority=100,in_port="s1-eth1",dl_src=5e:d9:d3:f9:6d:c7,dl_dst=56:3d:87:26:3e:2d actions=output:"s1-eth2"
cookie=0x0, duration=547.796s, table=0, n_packets=0, n_bytes=0, priority=100,in_port="s1-eth1",dl_src=ff:ff:ff:ff:ff:ff,dl_dst=56:3d:87:26:3e:2d actions=output:"s1-eth2"
cookie=0x0, duration=460.094s, table=0, n_packets=0, n_bytes=0, priority=100,in_port="s1-eth1",dl_src=56:3d:87:26:3e:2d,dl_dst=ff:ff:ff:ff:ff:ff actions=output:"s1-eth2"
cookie=0x0, duration=305.513s, table=0, n_packets=0, n_bytes=0, priority=100,in_port="s1-eth1",dl_src=56:3d:87:26:3e:2d,dl_dst=5e:d9:d3:f9:6d:c7 actions=output:"s1-eth2"
cookie=0x0, duration=219.928s, table=0, n_packets=34, n_bytes=1428, priority=100,in_port="s1-eth1",dl_src=5e:d9:d3:f9:6d:c7,dl_dst=ff:ff:ff:ff:ff:ff actions=output:"s1-eth2"
cookie=0x0, duration=179.253s, table=0, n_packets=0, n_bytes=0, priority=100,in_port=3,dl_src=56:3d:87:26:3e:2d,dl_dst=5e:d9:d3:f9:6d:c7 actions=output:"s1-eth1"
cookie=0x0, duration=55.468s, table=0, n_packets=4, n_bytes=280, priority=100,in_port="s1-eth2",dl_src=56:3d:87:26:3e:2d,dl_dst=5e:d9:d3:f9:6d:c7 actions=output:"s1-eth1"
root@ubuntu:~#
```

经过排查，我们发现在 mininet 自动生成树形拓扑结构时，s1 中给 s2、s3 分配的端口分别为 1 和 2，所以在配置流表时使用其他端口会导致配置失败。

通过这次实验，我深刻理解了如何利用 Ryu 控制器、Mininet 以及 curl 工具来构建软件定义网络（SDN）。这次实践不仅加深了我对网络交换机运作机制的认识，还让我掌握了搭建 SDN 环境的基本技能。

在实验过程中，我们遇到了一些挑战，但通过查阅相关资料和进行故障排查，我们最终成功解决了这些问题。这一过程不仅增强了我们的问题解决能力，还提高了我们的故障排查技能。这些经验对于我们未来在网络领域的学习和工作都具有重要的意义。

通过这次实验，我更加坚信软件定义网络是未来网络发展的重要方向。它为我们提供了更灵活、更高效的网络管理方式，使我们能够更好地应对不断变化的网络需求。我期待在未来的学习和工作中，能够继续探索和应用 SDN 技术，为网络的发展做出自己的贡献。