Software Define Network   Lab2

实验准备：

–Construct a network using Mininet
–Use wireshark to capture OpenFlow messages
–Ping and use ovs-ofctl to check the flow tables

实验要求：

–Construct the ARPANET-1969 (4 nodes) using Mininet
–Run Ryu as the remote controller
–Let UCLA ping UTAH
–If not reachable, try to solve it by programming with Ryu

实验环境：

Windows 10, VMware Workstation Pro, Ubuntu

实验过程：

Warn-Up:

1. 构建一个简单拓扑的 Mininet，使用 Wireshark 捕获 OpenFLow 协议数据包以及查看交换机流表

我们在 mininet/custom 目录下打开命令行终端，输入 sudo mn --topo=tree,2,2 --mac --controller=remote

remote 表示不用’Mininet’自带的控制器，尝试使用’Ryu’等远端控制器。

```
sdn@ubuntu:~/Desktop/mininet/custom$ sudo mn --topo=tree,2,2 --mac --controller=
remote
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Unable to contact the remote controller at 127.0.0.1:6633
Setting remote controller to 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:
```

在 mininet 命令行中输入 links，查看连接情况。

```
mininet> links
s1-eth1<->s2-eth3 (OK OK)
s1-eth2<->s3-eth3 (OK OK)
s2-eth1<->h1-eth0 (OK OK)
s2-eth2<->h2-eth0 (OK OK)
s3-eth1<->h3-eth0 (OK OK)
s3-eth2<->h4-eth0 (OK OK)
mininet> 
```

可以看到，主机 h1 和 h2 连接交换机 s2，主机 h3 和 h4 连接交换机 s3，交换机 s2 和 s3 连接交换机 s1，构成一个树形拓扑结构。

输入 sudo ovs-ofctl dump-flows s1，查看 s1 的流表表项。

```
sdn@ubuntu:~/Desktop/mininet/custom$ sudo ovs-ofctl dump-flows s1
 cookie=0x0, duration=71.282s, table=0, n_packets=361, n_bytes=21660, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
```

输入 sudo wireshark，启动 Wireshark，选择 Loopback:lo 端口，开始捕获。

No.	Time	Source	Destination	Protocol	Length	Info
4	0.000039335	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_HELLO
9	0.000110380	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_HELLO
14	0.501509761	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_HELLO
25	16.030075091	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_HELLO
30	16.030171852	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_HELLO
35	16.531334092	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_HELLO
46	32.063249520	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_HELLO
51	32.063329534	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_HELLO
56	32.564544203	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_HELLO
67	48.092670948	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_HELLO
72	48.092751393	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_HELLO
77	48.594220839	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_HELLO
88	64.128793791	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_HELLO
93	64.128862495	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_HELLO
98	64.630440031	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_HELLO
109	80.154296445	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_HELLO
114	80.154462583	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_HELLO
119	80.655836773	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_HELLO

我们捕获到不少 OpenFlow 协议报文，选择一条查看详细内容。

```
Wireshark · Packet 4 · Loopback: lo

▸ Frame 4: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 45522, Dst Port: 6653, Seq: 1, Ack: 1, Len: 8
▾ OpenFlow 1.5
    Version: 1.5 (0x06)
    Type: OFPT_HELLO (0)
    Length: 8
    Transaction ID: 14259
```

这是序号为 4 的报文，协议为 OpenFlow，Version 为 1.5，类型为 OFPT_Hello，长度为 8，ID 为 14259。

通过查找相关资料我们得知，OpenFLow 协议报文类型有三种分类，即同步，异步和 Controller to Switch 类型，其中 OFPT_Hello 类型属于同步类型，当连接启动时交换机和控制器会发送 Hello 消息进行交互。资料链接如下：

https://www.h3c.com/cn/d_201811/1131080_30005_0.htm

2. 使用 Ryu APP 查看拓扑

我们在 mininet/custom 目录下打开命令行终端，输入 sudo mn --custom topo-2sw-2host.py --topo mytopo --controller remote

remote 表示不用’Mininet’自带的控制器，尝试使用’Ryu’等远端控制器。

```
sdn@ubuntu:~/Desktop/mininet/custom$ sudo mn --custom topo-2sw-2host.py --topo m
ytopo --controller remote
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Unable to contact the remote controller at 127.0.0.1:6633
Setting remote controller to 127.0.0.1:6653
*** Adding hosts:
h1 h2
*** Adding switches:
s3 s4
*** Adding links:
(h1, s3) (s3, s4) (s4, h2)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 2 switches
s3 s4 ...
*** Starting CLI:
mininet>
```

在 mininet 命令行中输入 links，查看连接情况。

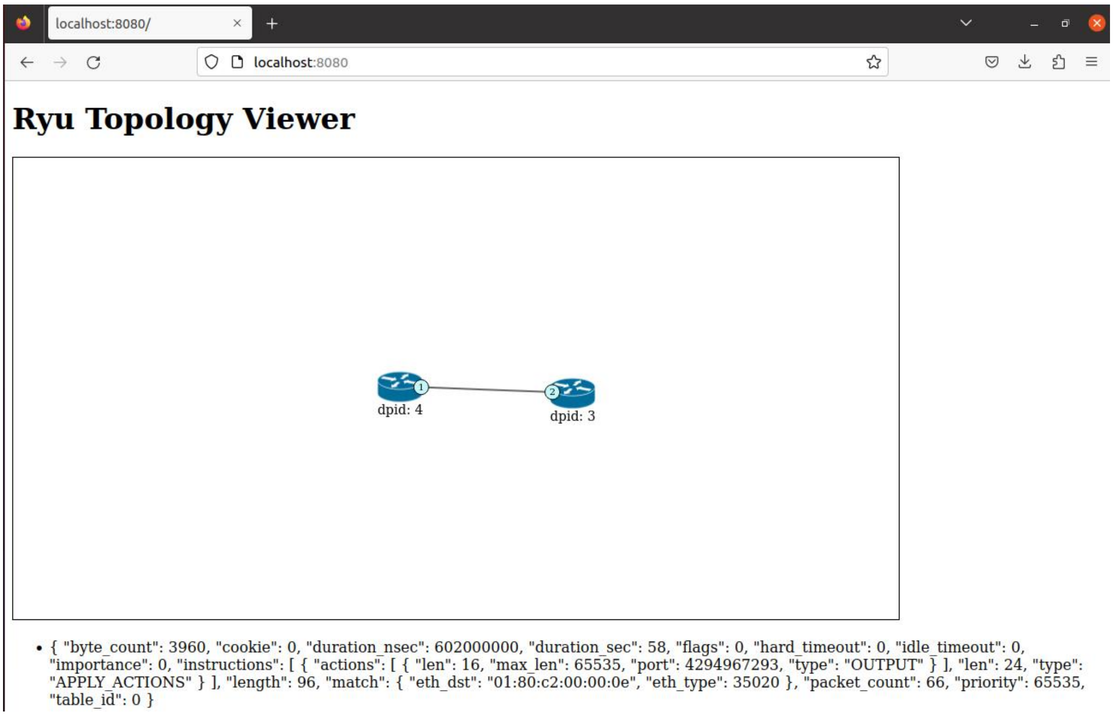
```
mininet> links
h1-eth0<->s3-eth1 (OK OK)
s3-eth2<->s4-eth1 (OK OK)
s4-eth2<->h2-eth0 (OK OK)
mininet>
```

可以看到，主机 h1 连接交换机 s3，交换机 s3 连接交换机 s4，交换机 s4 连接主机 h2，构成拓扑结构。

我们在 ryu/ryu/app/gui_topology 目录下，打开命令行终端，输入 ryu-manager --observe-links sdn/ryu/ryu/app/gui_topology/gui_topology.py

```
sdn@ubuntu:~/Desktop/ryu/ryu/app/gui_topology$ ryu-manager --observe-links gui_t
opology.py
loading app gui_topology.py
loading app ryu.app.ofctl_rest
loading app ryu.app.rest_topology
loading app ryu.app.ws_topology
loading app ryu.controller.ofp_handler
creating context wsgi
instantiating app None of DPSet
creating context dpset
instantiating app None of Switches
creating context switches
instantiating app gui_topology.py of GUIServerApp
instantiating app ryu.app.ofctl_rest of RestStatsApi
instantiating app ryu.app.rest_topology of TopologyAPI
instantiating app ryu.app.ws_topology of WebSocketTopology
instantiating app ryu.controller.ofp_handler of OFPHandler
(150898) wsgi starting up on http://0.0.0.0:8080
```

在浏览器中访问 localhost:8080 以查看拓扑，点击交换机即可查看流表。



3. 实现 Ryu 简单交换机

输入 sudo mn --controller remote --mac --topo=tree,2,2，创建树形拓扑。

```
sdn@ubuntu:~/Desktop/mininet/custom$ sudo mn --topo=tree,2,2 --mac --controller=
remote
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Unable to contact the remote controller at 127.0.0.1:6633
Setting remote controller to 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>
```

在 mininet 命令行中输入 pingall，测试各节点之间的连通性。

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X X
h2 -> X X X
h3 -> X X X
h4 -> X X X
*** Results: 100% dropped (0/12 received)
```

发现无法 ping 通，这是因为我们还没有启动 ryu-manager。

我们将实验指导书里的 ryu 简单交换机的参考代码保存，命名为 simple_switch.py，保存在 mininet/custom 目录下，代码详见实验指导书。

输入 sudo ryu-manager simple_switch.py，启动 Ryu 控制器。

```
sdn@ubuntu:~/Desktop/mininet/custom$ sudo ryu-manager simple_switch.py
[sudo] password for sdn:
loading app simple_switch.py
loading app ryu.controller.ofp_handler
instantiating app simple_switch.py of L2Switch
instantiating app ryu.controller.ofp_handler of OFPHandler
```

等待一段时间，直到 ryu-manager 光标不再闪烁之后，在 mininet 命令行中再次输入 pingall，测试各节点连通性。

```
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)
```

这次我们可以发现，各个节点之间都能互相 ping 通。

在 mininet 命令行中输入 xterm h4，打开 h4 的 xterm 终端，启动 Wireshark 捕获端口 h4-eth0。

```
mininet> xterm h4
mininet>
```

```
"Node: h4"
root@ubuntu:/home/sdn/Desktop/mininet/custom# sudo wireshark
 ** (wireshark:151460) 02:57:40.071724 [GUI WARNING] -- QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-root'
```

在 mininet 命令行中输入 h1 ping h3 -c3，使主机 h1 ping 主机 h3 三次，同时开始 h4-eth0 端口的捕获。

```
mininet> h1 ping h3 -c3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=3.67 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=2.93 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=2.71 ms

--- 10.0.0.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 2.711/3.104/3.669/0.409 ms
mininet>
```

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.0.1	10.0.0.3	ICMP	98	Echo (ping) request id=0x4fd3, seq=1/256, ttl=64 (reply in 2)
2	0.000498131	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping) reply id=0x4fd3, seq=1/256, ttl=64 (request in 1)
3	1.001213795	10.0.0.1	10.0.0.3	ICMP	98	Echo (ping) request id=0x4fd3, seq=2/512, ttl=64 (reply in 4)
4	1.001623691	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping) reply id=0x4fd3, seq=2/512, ttl=64 (request in 3)
5	2.002855011	10.0.0.1	10.0.0.3	ICMP	98	Echo (ping) request id=0x4fd3, seq=3/768, ttl=64 (reply in 6)
6	2.003312669	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping) reply id=0x4fd3, seq=3/768, ttl=64 (request in 5)
7	5.160158971	00:00:00_00:00:03	00:00:00_00:…	ARP	42	Who has 10.0.0.1? Tell 10.0.0.3
8	5.161588253	00:00:00_00:00:01	00:00:00_00:…	ARP	42	Who has 10.0.0.3? Tell 10.0.0.1
9	5.162229083	00:00:00_00:00:03	00:00:00_00:…	ARP	42	10.0.0.3 is at 00:00:00:00:00:03
10	5.163364703	00:00:00_00:00:01	00:00:00_00:…	ARP	42	10.0.0.1 is at 00:00:00:00:00:01

我们可以看到，抓包结果显示该交换机存在验证缺陷：packet_in_handler()函数会将数据包洪泛到交换机的所有端口，故 h1 和 h3 通讯时，h4 也会收到所有的包。

Task:

4. 实现 Ryu 自学习交换机

首先我们需要实现 1969 年的 Arpanet 拓扑结构，1969 年的 ARPANET 非常简单，仅由四个结点组成。假设每个结点都对应一个交换机，每个交换机都具有一个直连主机。前文给出的简单交换机洪泛数据包，虽然能初步实现主机间的通信，但会带来不必要的带宽消耗，并且会使通信内容泄露给第三者。需要在简单交换机的基础上实现二层自学习交换机，避免数据包的洪泛。

1969 年的 Arpanet 拓扑结构示意图如下：

Dezember 1969

自学习交换机的实现框架如下：

（1）控制器为每个交换机维护一个 mac-port 映射表。

（2）控制器收到 packet_in 消息后，解析其中携带的数据包。

（3）控制器学习 src_mac - in_port 映射。

（4）控制器查询 dst_mac，如果未学习，则洪泛数据包；如果已学习，则向指定端口转发数据包(packet_out)，并向交换机下发流表项(flow_mod)，指导交换机转发同类型的数据包。

拓扑结构和 mininet 构建实现于 topo_1969_1.py 文件之中，代码详见实验指导书。

基于实验指导书给出的自学习交换机框架，我们完善代码实现自学习交换机，代码如下：

```python
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 Switch(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

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

    def add_flow(self, datapath, priority, match, actions, idle_timeout=0,
hard_timeout=0):
        dp = datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser
        inst = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS,
actions)]
        mod = parser.OFPFlowMod(datapath=dp, priority=priority,
                                idle_timeout=idle_timeout,
                                hard_timeout=hard_timeout,
                                match=match, instructions=inst)
        dp.send_msg(mod)

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(
            ofp.OFPP_CONTROLLER, ofp.OFPCML_NO_BUFFER)]
        self.add_flow(dp, 0, match, actions)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath
```

```python
        ofp = dp.ofproto
        parser = dp.ofproto_parser
        # the identity of switch
        dpid = dp.id
        self.mac_to_port.setdefault(dpid, {})
        # the port that receive the packet
        in_port = msg.match['in_port']
        pkt = packet.Packet(msg.data)
        eth_pkt = pkt.get_protocol(ethernet.ethernet)
        # get the mac
        dst = eth_pkt.dst
        src = eth_pkt.src
        # we can use the logger to print some useful information
        self.logger.info('packet: %s %s %s %s', dpid, src, dst, in_port)
        # you need to code here to avoid the direct flooding
        # having fun
        # :)

        # Save dpid and src of in_port to dict mac_to_port, learning
        self.mac_to_port[dpid][src] = in_port
        if dst in self.mac_to_port[dpid]:
            # Setting direction according to the table
            out_port = self.mac_to_port[dpid][dst]
        else:
            out_port = ofp.OFPP_FLOOD  # Flood
        actions = [parser.OFPActionOutput(out_port)]
        if out_port != ofp.OFPP_FLOOD:  # Add flow
            match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
            self.add_flow(dp, 1, match, actions)
        out = parser.OFPPacketOut(datapath=dp, buffer_id=msg.buffer_id,
                                  in_port=in_port, actions=actions,
data=msg.data)
        dp.send_msg(out)
```

代码分析：

我们在 Switch()类型初始化函数__init__()函数中，选择初始化一个 mac_to_port 字典(Dictionary)，用于记录 MAC 地址和端口等信息，初始化为空。Python 的字典数据类型的资料如下：

https://www.runoob.com/python/python-dictionary.html

我们主要需要修改代码里的 packer_in_handler()函数，在得到了源 MAC 地址和目的 MAC 地址之后，我们将 in_port 对象内的信息记录到 mac_to_port 字典里的 dpid 和 src 键下，如果目的 MAC 地址已经记录在了字典里的 dpid 键下，我们可以根据字典确定 out_port 的转发方向，如果字典里没有已经存在的

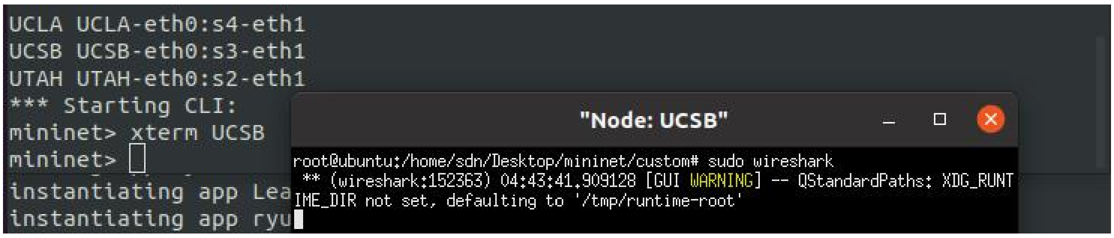
记录，则洪泛转发。根据选择的转发方式，确定 Output 时的动作(Action)。同时如果不是洪泛转发，则需要添加流表。最后在完成流表的添加后，再进行转发数据包的封装与实施转发。

修改后的代码命名为 Learning_Switch.py，在命令行终端输入 sudo python topo_1969_1.py，构建 mininet 网络拓扑；在另一个命令行终端输入 sudo ryu-manager Learning_Switch.py，启动 Ryu 控制器。

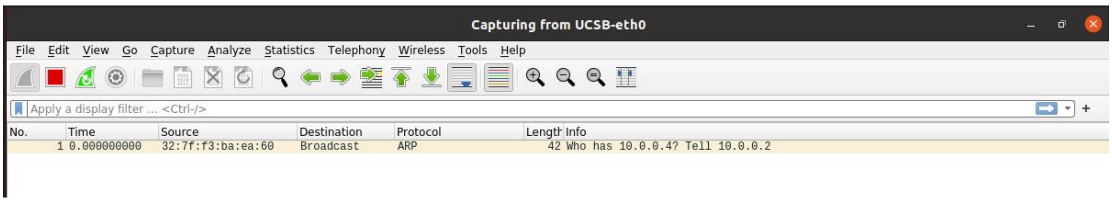
```
sdn@ubuntu:~/Desktop/mininet/custom$ sudo python topo_1969_1.py
[sudo] password for sdn:
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
SRI UCLA UCSB UTAH
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(s1, SRI) (10.00Mbit 50ms delay) (10.00Mbit 50ms delay) (s1, s2) (10.00Mbit 34ms
 delay) (10.00Mbit 34ms delay) (s1, s3) (10.00Mbit 13ms delay) (10.00Mbit 13ms d
elay) (s1, s4) (s2, UTAH) (s3, UCSB) (s4, UCLA)
*** Configuring hosts
SRI (cfs -1/100000us) UCLA (cfs -1/100000us) UCSB (cfs -1/100000us) UTAH (cfs -1
/100000us)
*** Starting controller
c0
*** Starting 4 switches
s1 s2 s3 s4 ...(10.00Mbit 50ms delay) (10.00Mbit 34ms delay) (10.00Mbit 13ms del
ay) (10.00Mbit 50ms delay) (10.00Mbit 34ms delay) (10.00Mbit 13ms delay)
```

```
sdn@ubuntu:~/Desktop/mininet/custom$ sudo ryu-manager Learning_Switch.py
[sudo] password for sdn:
loading app Learning_Switch.py
loading app ryu.controller.ofp_handler
instantiating app Learning_Switch.py of Switch
instantiating app ryu.controller.ofp_handler of OFPHandler
packet: 2 46:12:99:1e:7c:bc 33:33:00:00:00:02 2
packet: 1 6a:0c:11:aa:65:c0 33:33:00:00:00:02 1
packet: 4 6a:0c:11:aa:65:c0 33:33:00:00:00:02 2
packet: 1 aa:76:ad:e9:63:83 33:33:00:00:00:02 3
packet: 3 6a:0c:11:aa:65:c0 33:33:00:00:00:02 2
packet: 4 aa:76:ad:e9:63:83 33:33:00:00:00:02 2
packet: 2 6a:0c:11:aa:65:c0 33:33:00:00:00:02 2
```

在 mininet 命令行中输入 xterm UCSB，启动 UCSB 主机的 xterm 终端，在终端内输入 sudo wireshark，启动 Wireshark 进行数据包捕获。



在 Wireshark 中选择 UCSB-eth0 端口进行捕获，同时在 mininet 命令行中输入 UCLA ping UTAH -c3，测试两主机之间的连通性，查看捕获结果。



```
mininet> UCLA ping UTAH -c3
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=271 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=129 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=129 ms

--- 10.0.0.4 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 128.636/176.161/270.659/66.820 ms
mininet>
```

可以看到，UCLA 成功 ping 通 UTAH，而且 UCSB 没有收到相关数据包，成功实现了自学习交换机。

2	1.987073415	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REQUEST
3	1.988203580	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REPLY
5	1.999682226	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REQUEST
6	1.999769513	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REQUEST
7	2.001026950	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REPLY
9	2.001327258	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REPLY
11	2.037801976	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REQUEST
12	2.039285445	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REPLY
14	6.990361371	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REQUEST
15	6.991571903	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REPLY
17	7.002414367	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REQUEST
18	7.002496634	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REQUEST
19	7.003645559	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REPLY
21	7.004035831	127.0.0.1	127.0.0.1	OpenFlow	74	Type: OFPT_ECHO_REPLY

5. 处理环路广播的自学习交换机

UCLA 和 UCSB 通信频繁，两者间建立了一条直连链路。在新的拓扑 topo_1969_2.py 中运行自学习交换机，UCLA 和 UTAH 之间无法正常通信。分析流表发现，源主机虽然只发了很少的几个数据包，但流表项却匹配了上千次。

新的 Arpanet 拓扑结构示意图如下：

在命令行终端内输入 sudo python topo_1969_2.py，构建 mininet 网络拓扑；在另一个命令行终端内输入 sudo ryu-manager Learning_Switch.py，启动 Ryu 控制器。

```
sdn@ubuntu:~/Desktop/mininet/custom$ sudo python topo_1969_2.py
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
SRI UCLA UCSB UTAH
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(s1, SRI) (10.00Mbit) (10.00Mbit) (s1, s2) (10.00Mbit) (10.00Mbit) (s1, s3) (10.
00Mbit) (10.00Mbit) (s1, s4) (s2, UTAH) (s3, UCSB) (10.00Mbit) (10.00Mbit) (s3,
s4) (s4, UCLA)
*** Configuring hosts
SRI (cfs -1/100000us) UCLA (cfs -1/100000us) UCSB (cfs -1/100000us) UTAH (cfs -1
/100000us)
*** Starting controller
c0
*** Starting 4 switches
s1 s2 s3 s4 ...(10.00Mbit) (10.00Mbit) (10.00Mbit) (10.00Mbit) (10.00Mbit) (10.0
0Mbit) (10.00Mbit) (10.00Mbit)
```

```
sdn@ubuntu:~/Desktop/mininet/custom$ sudo ryu-manager Learning_Switch.py
loading app Learning_Switch.py
loading app ryu.controller.ofp_handler
instantiating app Learning_Switch.py of Switch
instantiating app ryu.controller.ofp_handler of OFPHandler
packet: 3 22:04:6f:a6:77:4e 33:33:00:00:00:02 1
packet: 3 72:ae:ba:ac:3e:e5 33:33:00:00:00:02 3
packet: 3 22:04:6f:a6:77:4e 33:33:00:00:00:02 2
packet: 3 22:04:6f:a6:77:4e 33:33:00:00:00:02 3
packet: 3 72:ae:ba:ac:3e:e5 33:33:00:00:00:02 3
packet: 3 22:04:6f:a6:77:4e 33:33:00:00:00:02 2
packet: 3 22:04:6f:a6:77:4e 33:33:00:00:00:02 3
packet: 3 72:ae:ba:ac:3e:e5 33:33:00:00:00:02 3
packet: 3 22:04:6f:a6:77:4e 33:33:00:00:00:02 2
packet: 3 22:04:6f:a6:77:4e 33:33:00:00:00:02 3
packet: 3 72:ae:ba:ac:3e:e5 33:33:00:00:00:02 3
packet: 3 22:04:6f:a6:77:4e 33:33:00:00:00:02 2
packet: 3 22:04:6f:a6:77:4e 33:33:00:00:00:02 3
packet: 3 72:ae:ba:ac:3e:e5 33:33:00:00:00:02 3
```

可以看到，在 ryu-manager 中出现了数目相当大的流表记录，且同一个数据包多次出现，只有 in_port 端口的键值发生了变化。

在 mininet 命令行中输入 UCLA ping UTAH -c3，测试两节点之间的连通性。

```
mininet> UCLA ping UTAH -c3
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
From 10.0.0.2 icmp_seq=1 Destination Host Unreachable
From 10.0.0.2 icmp_seq=2 Destination Host Unreachable
From 10.0.0.2 icmp_seq=3 Destination Host Unreachable

--- 10.0.0.4 ping statistics ---
3 packets transmitted, 0 received, +3 errors, 100% packet loss, time 2032ms
pipe 3
mininet>
```

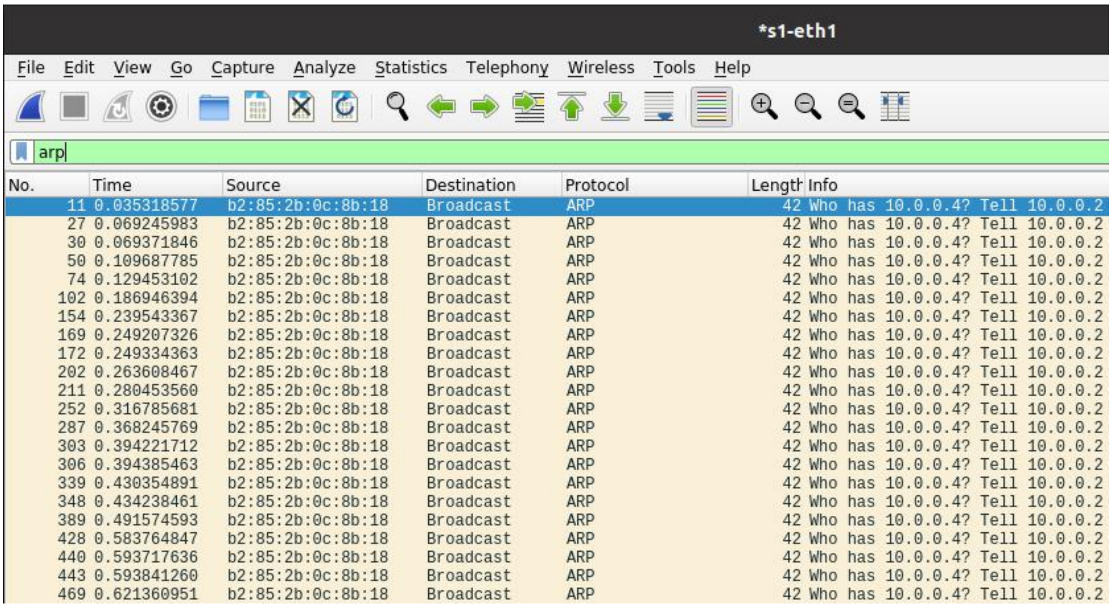
可以看到，两节点之间无法 ping 通，显示主机不可达。

在 mininet 命令行终端内输入 dpctl dump-flows，查看与各主机直接相连的交换机的流表表项。

```
mininet> dpctl dump-flows
*** s1 ------------------------------------------------------------------------
 cookie=0x0, duration=65.649s, table=0, n_packets=2602, n_bytes=109284, priority
=1,in_port="s1-eth2",dl_dst=b2:85:2b:0c:8b:18 actions=output:"s1-eth3"
 cookie=0x0, duration=192.218s, table=0, n_packets=139797, n_bytes=11716876, pri
ority=0 actions=CONTROLLER:65535
*** s2 ------------------------------------------------------------------------
 cookie=0x0, duration=65.717s, table=0, n_packets=2602, n_bytes=109284, priority
=1,in_port="s2-eth1",dl_dst=b2:85:2b:0c:8b:18 actions=output:"s2-eth2"
 cookie=0x0, duration=181.192s, table=0, n_packets=136292, n_bytes=11401739, pri
ority=0 actions=CONTROLLER:65535
*** s3 ------------------------------------------------------------------------
 cookie=0x0, duration=65.648s, table=0, n_packets=2601, n_bytes=109242, priority
=1,in_port="s3-eth2",dl_dst=b2:85:2b:0c:8b:18 actions=output:"s3-eth2"
 cookie=0x0, duration=187.129s, table=0, n_packets=139786, n_bytes=11712937, pri
ority=0 actions=CONTROLLER:65535
*** s4 ------------------------------------------------------------------------
 cookie=0x0, duration=192.231s, table=0, n_packets=139802, n_bytes=11717365, pri
ority=0 actions=CONTROLLER:65535
mininet>
```

可以看到，虽然 UCLA 与 UTAH 主机之间只 ping 了 3 次，但流表项却匹配了约 140000 次，出现了异常情况。

在命令行终端内输入 sudo wireshark，启动 Wireshark 进行数据包捕获。可以随意选择一个交换机端口，例如我选择了 s1-eth1 端口进行捕获。



No.	Time	Source	Destination	Protocol	Length	Info
11	0.035318577	b2:85:2b:0c:8b:18	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
27	0.069245983	b2:85:2b:0c:8b:18	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
30	0.069371846	b2:85:2b:0c:8b:18	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
50	0.109687785	b2:85:2b:0c:8b:18	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
74	0.129453102	b2:85:2b:0c:8b:18	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
102	0.186946394	b2:85:2b:0c:8b:18	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
154	0.239543367	b2:85:2b:0c:8b:18	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
169	0.249207326	b2:85:2b:0c:8b:18	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
172	0.249334363	b2:85:2b:0c:8b:18	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
202	0.263608467	b2:85:2b:0c:8b:18	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
211	0.280453560	b2:85:2b:0c:8b:18	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
252	0.316785681	b2:85:2b:0c:8b:18	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
287	0.368245769	b2:85:2b:0c:8b:18	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
303	0.394221712	b2:85:2b:0c:8b:18	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
306	0.394385463	b2:85:2b:0c:8b:18	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
339	0.430354891	b2:85:2b:0c:8b:18	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
348	0.434238461	b2:85:2b:0c:8b:18	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
389	0.491574593	b2:85:2b:0c:8b:18	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
428	0.583764847	b2:85:2b:0c:8b:18	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
440	0.593717636	b2:85:2b:0c:8b:18	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
443	0.593841260	b2:85:2b:0c:8b:18	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
469	0.621360951	b2:85:2b:0c:8b:18	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2

可以看到，WireShark 也截取到了数目异常大的相同报文，这些报文均为 ARP 请求报文，UCLA 主机（IP 地址为 10.0.0.2）发送 ARP 请求报文，请求 UTAH 主机（IP 地址为 10.0.0.4）的 MAC 地址，而这些 ARP 请求报文没有得到应答，所以 UCLA 主机持续发送请求报文，且由于拓扑结构中存在环路，形成了 ARP 广播风暴。这实际上是 ARP 广播数据包在环状拓扑中洪泛导致的，传统网络利用生成树协议解决这一问题。在 SDN 中，不必局限于生成树协议，可以通过多种新的策略解决这一问题。

处理环路广播的思路如下：

当序号为 dpid 的交换机从 in_port 第一次收到某个 src_mac 主机发出，询问 dst_ip 的广播 ARP Request 数据包时，控制器记录一个映射 (dpid, src_mac, dst_ip)->in_port。下一次该交换机收到同一(src_mac, dst_ip)但 in_port 不同的 ARP Request 数据包时直接丢弃，否则洪泛。

基于实验指导书给出的处理环路广播的自学习交换机框架，我们完善代码实现处理环路广播的自学习交换机，代码如下：

```
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
from ryu.lib.packet import arp
from ryu.lib.packet import ether_types

ETHERNET = ethernet.ethernet.__name__
ETHERNET_MULTICAST = "ff:ff:ff:ff:ff:ff"
ARP = arp.arp.__name__


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

    def __init__(self, *args, **kwargs):
        super(Switch_Dict, self).__init__(*args, **kwargs)
        # (dpid, src_mac, dst_ip)=>in_port, you may use it in mission 2
        # maybe you need a global data structure to save the mapping
        # just data structure in mission 1
        self.mac_to_port = {}
        self.arp_table = {}

    def add_flow(self, datapath, priority, match, actions, idle_timeout=0,
hard_timeout=0):
        dp = datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser
        inst = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS,
actions)]
        mod = parser.OFPFlowMod(datapath=dp, priority=priority,
                              idle_timeout=idle_timeout,
                              hard_timeout=hard_timeout,
                              match=match, instructions=inst)
        dp.send_msg(mod)

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath
```

```
        ofp = dp.ofproto
        parser = dp.ofproto_parser
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(
            ofp.OFPP_CONTROLLER, ofp.OFPCML_NO_BUFFER)]
        self.add_flow(dp, 0, match, actions)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser
        # the identity of switch
        dpid = dp.id
        self.mac_to_port.setdefault(dpid, {})
        # the port that receive the packet
        in_port = msg.match['in_port']
        pkt = packet.Packet(msg.data)
        eth_pkt = pkt.get_protocol(ethernet.ethernet)
        if eth_pkt.ethertype == ether_types.ETH_TYPE_LLDP:
            return
        if eth_pkt.ethertype == ether_types.ETH_TYPE_IPV6:
            return
        # get the mac
        dst = eth_pkt.dst
        src = eth_pkt.src
        # we can use the logger to print some useful information
        self.logger.info('packet: %s %s %s %s', dpid, src, dst, in_port)
        # get protocols
        header_list = dict((p.protocol_name, p)
                           for p in pkt.protocols if type(p) != str)
        if dst == ETHERNET_MULTICAST and ARP in header_list:
            # you need to code here to avoid broadcast loop to finish mission
2

            dst_ip = header_list[ARP].dst_ip
            # set logger to show useful information
            self.logger.info("ARP Learning: %s %s %s %s",
                             dpid, src, dst_ip, in_port)
            # If info is already in ARP table
            if (dpid, src, dst_ip) in self.arp_table:
                # The same info comes from another port, Just Drop it
                if self.arp_table[dpid, src, dst_ip] != in_port:
```

```
                    out = parser.OFPPacketOut(datapath=dp,
buffer_id=ofp.OFPCML_NO_BUFFER,
                                              in_port=in_port, actions=[],
data=None)  # Drop
                    dp.send_msg(out)
                    return
            # If info is not in ARP table, Learn and FLood it
            else:
                # Arp table learning
                self.arp_table[(dpid, src, dst_ip)] = in_port
                actions = [parser.OFPActionOutput(ofp.OFPP_FLOOD)]  #
Flood
                out = parser.OFPPacketOut(datapath=dp,
buffer_id=msg.buffer_id,
                                          in_port=in_port, actions=actions,
data=msg.data)
                dp.send_msg(out)

        # self-learning
        # you need to code here to avoid the direct flooding
        # having fun
        # :)
        # just code in mission 1

        # Save dpid and src of in_port to dict mac_to_port, learning
        self.mac_to_port[dpid][src] = in_port
        if dst in self.mac_to_port[dpid]:
            # Setting direction according to the table
            out_port = self.mac_to_port[dpid][dst]
        else:
            out_port = ofp.OFPP_FLOOD  # Flood
        actions = [parser.OFPActionOutput(out_port)]
        if out_port != ofp.OFPP_FLOOD:  # Add flow
            match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
            self.add_flow(dp, 1, match, actions)
        out = parser.OFPPacketOut(datapath=dp, buffer_id=msg.buffer_id,
                                  in_port=in_port, actions=actions,
data=msg.data)
        dp.send_msg(out)
```

代码分析：

在自学习交换机的基础上，我们在 Switch()类型初始化函数__init__()函数中，除了选择初始化一个 mac_to_port 字典，还初始化了一个 arp_table 字典，用于记录 dpid，源 MAC 地址，目的 IP 地址等信息。

我们还需要修改代码里的 packet_in_handler()函数，如果 header_list 里的键值符合 ARP 报文的特征，即目的 MAC 地址(dst)是以太网组播(ETHERNET_MULTICAST)并且 header_list 的键为 ARP 类型，则记录 header_list 对象里的 dst_ip，同时使用 logger 显示有帮助的信息。如果表项的内容已经存在于 arp_table 中并且 arp_table 内记录的信息与 in_port 对象不完全相同，即符合 src_mac, dst_ip 相同但 in_port 不同的条件，我们就丢弃这个数据包，否则若表项内容不存在于 arp_table 中，则洪泛转发数据包。随后就是自学习交换机中已经实现了的功能。

修改后的代码命名为 Broadcast_Loop.py，在命令行终端内输入 sudo python topo_1969_2.py，构建 mininet 网络拓扑；在另一个命令行终端内输入 sudo ryu-manager Broadcast_Loop.py，启动 Ryu 控制器。

```
sdn@ubuntu:~/Desktop/mininet/custom$ sudo python topo_1969_2.py
[sudo] password for sdn:
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
SRI UCLA UCSB UTAH
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(s1, SRI) (10.00Mbit) (10.00Mbit) (s1, s2) (10.00Mbit) (10.00Mbit) (s1, s3) (10.
00Mbit) (10.00Mbit) (s1, s4) (s2, UTAH) (s3, UCSB) (10.00Mbit) (10.00Mbit) (s3,
s4) (s4, UCLA)
*** Configuring hosts
SRI (cfs -1/100000us) UCLA (cfs -1/100000us) UCSB (cfs -1/100000us) UTAH (cfs -1
/100000us)
*** Starting controller
c0
*** Starting 4 switches
s1 s2 s3 s4 ...(10.00Mbit) (10.00Mbit) (10.00Mbit) (10.00Mbit) (10.00Mbit) (10.0
0Mbit) (10.00Mbit) (10.00Mbit)
```

```
sdn@ubuntu:~/Desktop/mininet/custom$ sudo ryu-manager Broadcast_Loop.py
[sudo] password for sdn:
loading app Broadcast_Loop.py
loading app ryu.controller.ofp_handler
instantiating app Broadcast_Loop.py of Switch_Dict
instantiating app ryu.controller.ofp_handler of OFPHandler
packet: 4 ee:54:6d:2f:cf:c1 ff:ff:ff:ff:ff:ff 1
ARP Learning: 4 ee:54:6d:2f:cf:c1 10.0.0.4 1
```

在 mininet 命令行中输入 UCLA ping UTAH -c3，测试两节点之间的连通性。

```
mininet> UCLA ping UTAH -c3
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=33.3 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=0.549 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=0.062 ms

--- 10.0.0.4 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2024ms
rtt min/avg/max/mdev = 0.062/11.301/33.292/15.551 ms
mininet>
```

```
packet: 4 ee:54:6d:2f:cf:c1 ff:ff:ff:ff:ff:ff 1
ARP Learning: 4 ee:54:6d:2f:cf:c1 10.0.0.4 1
packet: 1 ee:54:6d:2f:cf:c1 ff:ff:ff:ff:ff:ff 4
ARP Learning: 1 ee:54:6d:2f:cf:c1 10.0.0.4 4
packet: 1 ee:54:6d:2f:cf:c1 ff:ff:ff:ff:ff:ff 4
ARP Learning: 1 ee:54:6d:2f:cf:c1 10.0.0.4 4
packet: 3 ee:54:6d:2f:cf:c1 ff:ff:ff:ff:ff:ff 3
ARP Learning: 3 ee:54:6d:2f:cf:c1 10.0.0.4 3
packet: 3 ee:54:6d:2f:cf:c1 ff:ff:ff:ff:ff:ff 3
ARP Learning: 3 ee:54:6d:2f:cf:c1 10.0.0.4 3
packet: 2 ee:54:6d:2f:cf:c1 ff:ff:ff:ff:ff:ff 2
ARP Learning: 2 ee:54:6d:2f:cf:c1 10.0.0.4 2
packet: 3 ee:54:6d:2f:cf:c1 ff:ff:ff:ff:ff:ff 2
ARP Learning: 3 ee:54:6d:2f:cf:c1 10.0.0.4 2
packet: 1 ee:54:6d:2f:cf:c1 ff:ff:ff:ff:ff:ff 3
ARP Learning: 1 ee:54:6d:2f:cf:c1 10.0.0.4 3
```

可以看到，UCLA 主机成功 ping 通 UTAH 主机。

在 mininet 命令行终端内输入 dpctl dump-flows，查看与各主机直接相连的交换机的流表表项。

```
mininet> dpctl dump-flows
*** s1 ------------------------------------------------------------------------
 cookie=0x0, duration=433.825s, table=0, n_packets=12, n_bytes=840, priority=1,i
n_port="s1-eth2",dl_dst=ee:54:6d:2f:cf:c1 actions=output:"s1-eth4"
 cookie=0x0, duration=433.814s, table=0, n_packets=8, n_bytes=616, priority=1,in
_port="s1-eth4",dl_dst=e6:90:dd:d5:c5:05 actions=output:"s1-eth2"
 cookie=0x0, duration=472.713s, table=0, n_packets=30, n_bytes=2330, priority=0
actions=CONTROLLER:65535
*** s2 ------------------------------------------------------------------------
 cookie=0x0, duration=433.836s, table=0, n_packets=11, n_bytes=798, priority=1,i
n_port="s2-eth1",dl_dst=ee:54:6d:2f:cf:c1 actions=output:"s2-eth2"
 cookie=0x0, duration=433.818s, table=0, n_packets=8, n_bytes=616, priority=1,in
_port="s2-eth2",dl_dst=e6:90:dd:d5:c5:05 actions=output:"s2-eth1"
 cookie=0x0, duration=472.718s, table=0, n_packets=15, n_bytes=1049, priority=0
actions=CONTROLLER:65535
*** s3 ------------------------------------------------------------------------
 cookie=0x0, duration=472.724s, table=0, n_packets=22, n_bytes=1659, priority=0
actions=CONTROLLER:65535
*** s4 ------------------------------------------------------------------------
 cookie=0x0, duration=433.837s, table=0, n_packets=11, n_bytes=798, priority=1,i
n_port="s4-eth2",dl_dst=ee:54:6d:2f:cf:c1 actions=output:"s4-eth1"
 cookie=0x0, duration=433.835s, table=0, n_packets=8, n_bytes=616, priority=1,in
_port="s4-eth1",dl_dst=e6:90:dd:d5:c5:05 actions=output:"s4-eth2"
 cookie=0x0, duration=472.730s, table=0, n_packets=22, n_bytes=1780, priority=0
actions=CONTROLLER:65535
mininet> 
```

可以看到，流表表项的匹配次数明显减少。

实验结果：

1. 学会了使用 Ryu 远程控制器，以及使用 Ryu 控制器查看网络拓扑结构。
2. 学会了使用命令行指令查看流表表项以及使用 Wireshark 捕获控制平面报文信息。
3. 学会了使用 Ryu 远程控制器来配置网络拓扑中的交换机，学习了简单洪泛交换机的代码，功能与缺陷。
4. 学会了实现 Ryu 自学习交换机，熟悉了 Ryu 控制器中的数据结构。
5. 学会了基于 Ryu 自学习交换机处理环路广播，了解了交换机的各种操作。