实验名称：Lab7 网络层控制平面流表下发
实验人：谢志康
学号：22307110187
时间：24.12.7 – 24.12.13

**环境准备：**
VMWare Linux 虚拟机
Mininet　mn --version: 2.3.0
Git 方式下载 ryu　git clone https://github.com/faucetsdn/ryu.git; cd ryu; pip install .
或 git clone https://gitclone.com/github.com/faucetsdn/ryu.git
直接 install 错误，开虚拟环境 install：
--sudo apt install python3-venv
--python3 -m venv myenv
--source myenv/bin/activate
又出问题：
Can not Install Ryu: AttributeError: module 'setuptools.command.easy_install' has no attribute 'get_script_args'
Python 版本改为 3.10 即可（3.12 太新了，不兼容）。
先后遇见两个错误：
TypeError: cannot set 'is_timeout' attribute of immutable type 'TimeoutError'
--git clone https://github.com/eventlet/eventlet.git
（或--git clone https://gitclone.com/github.com/eventlet/eventlet.git）
--cd eventlet
--pip install .
module 'collections' has no attribute 'MutableMapping'
参考安装 ryu_ryu 安装-CSDN 博客解决。
ImportError ： cannot import name 'ALREADY_HANDLLED':
参考实验文档链接解决：Update wsgi.py by isaac2077 · Pull Request #166 · faucetsdn/ryu

```
class _AlreadyHandledResponse(Response):
    # XXX: Eventlet API should not be used directly.
    #from eventlet.wsgi import ALREADY_HANDLED
    #_ALREADY_HANDLED = ALREADY_HANDLED
    import eventlet.wsgi
    _ALREADY_HANDLED = getattr(getattr(eventlet.wsgi, "WSGI_LOCAL", None), "already_handled", None)

    def __call__(self, environ, start_response):
        return self._ALREADY_HANDLED
```

最终 ryu-manager 正常运行：
root@kurumi:~# ryu-manager --version
ryu-manager 4.34

```
(myenv) root@ubuntu:~/myenv/lib/python3.10/site-packages/ryu/app# nano wsgi.py
(myenv) root@ubuntu:~/myenv/lib/python3.10/site-packages/ryu/app# ryu-manager --
version
ryu-manager 4.34
```

环境试运行：

```
2.3.0
root@ubuntu:~# sudo mn --topo single,3 --mac --switch ovsk --controller remote
*** Creating network
*** Adding controller
Connecting to remote controller at 127.0.0.1:6653
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1)
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet>
```

Mininet 生成网络

```
root@ubuntu: ~/myenv/lib/python3.10/site-packages/ryu/app
0,supported=0,peer=0), 2: OFPPhyPort(port_no=2,hw_addr='92:c9:71:ac:e1:66',nam
b's1-eth2',config=0,state=0,curr=192,advertised=0,supported=0,peer=0), 3: OFPP
Port(port_no=3,hw_addr='76:aa:73:a0:90:6b',name=b's1-eth3',config=0,state=0,cu
=192,advertised=0,supported=0,peer=0)})
move onto main mode
EVENT ofp_event->SimpleSwitch EventOFPPacketIn
packet in 1 00:00:00:00:00:03 33:33:ff:00:00:03 3
EVENT ofp_event->SimpleSwitch EventOFPPacketIn
packet in 1 00:00:00:00:00:01 33:33:ff:00:00:01 1
EVENT ofp_event->SimpleSwitch EventOFPPacketIn
packet in 1 00:00:00:00:00:02 33:33:00:00:00:16 2
EVENT ofp_event->SimpleSwitch EventOFPPacketIn
EVENT ofp_event->SimpleSwitch EventOFPPacketIn
packet in 1 00:00:00:00:00:02 33:33:00:00:00:16 2
packet in 1 00:00:00:00:00:02 33:33:00:00:00:02 2
EVENT ofp_event->SimpleSwitch EventOFPPacketIn
packet in 1 00:00:00:00:00:02 33:33:00:00:00:16 2
EVENT ofp_event->SimpleSwitch EventOFPPacketIn
EVENT ofp_event->SimpleSwitch EventOFPPacketIn
packet in 1 00:00:00:00:00:03 33:33:00:00:00:16 3
packet in 1 00:00:00:00:00:03 33:33:00:00:00:02 3
EVENT ofp_event->SimpleSwitch EventOFPPacketIn
EVENT ofp_event->SimpleSwitch EventOFPPacketIn
packet in 1 00:00:00:00:00:01 33:33:00:00:00:16 1
```

Openflow 控制器端开始显示包的行为。

```
*** Starting CLI:
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=19.9 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.563 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.094 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.089 ms
```

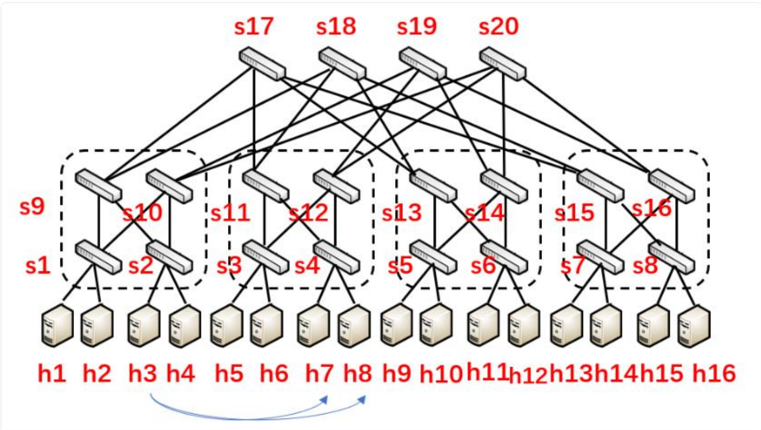
H1 ping h2 可以 ping 通

```
ubuntu@ubuntu:~/Desktop$ sudo -i
root@ubuntu:~# source myenv/bin/activate
(myenv) root@ubuntu:~# ls
myenv  packages.chroot  ryu  snap
(myenv) root@ubuntu:~# ovs-ofctl dump-flows s1
 cookie=0x0, duration=67.885s, table=0, n_packets=70, n_bytes=6692, in_port="s1-eth2",dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:01 actions=output:"s1-eth1"
 cookie=0x0, duration=67.880s, table=0, n_packets=69, n_bytes=6594, in_port="s1-eth1",dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02 actions=output:"s1-eth2"
(myenv) root@ubuntu:~#
```
观察流表，行为正常。

**ryu 组件介绍及代码分析：**

讲的挺全了的，我们的网络拓扑其实就是改一下初始化，把拓扑结构定义好，然后处理方式写出那三种即可。

**实验内容：**




**LPR.py：**

仿照 fattree_routing.py 文件，主要是理解 ryu 框架下的一些特殊语法，算法实现上其实比较简单。初始化部分：我的学号尾号 87，最后要求是 7→11 和 7→12

```python
self.hosts = {'10.0.0.1' : (1, 1), '10.0.0.2' : (1, 2), '10.0.0.3' : (2, 1), '10.0.0.4' : (2, 2),
              '10.0.0.5' : (3, 1), '10.0.0.6' : (3, 2), '10.0.0.7' : (4, 1), '10.0.0.8' : (4, 2),
              '10.0.0.9' : (5, 1), '10.0.0.10': (5, 2), '10.0.0.11': (6, 1), '10.0.0.12': (6, 2),
              '10.0.0.13': (7, 1), '10.0.0.14': (7, 2), '10.0.0.15': (8, 1), '10.0.0.16': (8, 2)}
self.parent = {1:(9,10),  2:(9,10),   3:(11,12),  4:(11,12),  5:(13,14),  6:(13,14),  7:(15,16),  8:(15,16),
               9:(17,18), 10:(19,20), 11:(17,18), 12:(19,20), 13:(17,18), 14:(19,20), 15:(17,18), 16:(19,20)}
self.son = {
    9 :{'10.0.0.1':1,'10.0.0.2':1,'10.0.0.3':2,'10.0.0.4':2},
    10:{'10.0.0.1':1,'10.0.0.2':1,'10.0.0.3':2,'10.0.0.4':2},
    11:{'10.0.0.5':3,'10.0.0.6':3,'10.0.0.7':4,'10.0.0.8':4},
    12:{'10.0.0.5':3,'10.0.0.6':3,'10.0.0.7':4,'10.0.0.8':4},
    13:{'10.0.0.9':5,'10.0.0.10':5,'10.0.0.11':6,'10.0.0.12':6},
    14:{'10.0.0.9':5,'10.0.0.10':5,'10.0.0.11':6,'10.0.0.12':6},
    15:{'10.0.0.13':7,'10.0.0.14':7,'10.0.0.15':8,'10.0.0.16':8},
    16:{'10.0.0.13':7,'10.0.0.14':7,'10.0.0.15':8,'10.0.0.16':8},
    17:{'10.0.0.1':9,'10.0.0.2':9,'10.0.0.3':9,'10.0.0.4':9,'10.0.0.5':11,'10.0.0.6':11,'10.0.0.7':11,'10.0.0.8':11,
    '10.0.0.9':13,'10.0.0.10':13,'10.0.0.11':13,'10.0.0.12':13,'10.0.0.13':15,'10.0.0.14':15,'10.0.0.15':15,'10.0.0.16':15}
    18:{'10.0.0.1':9,'10.0.0.2':9,'10.0.0.3':9,'10.0.0.4':9,'10.0.0.5':11,'10.0.0.6':11,'10.0.0.7':11,'10.0.0.8':11,
    '10.0.0.9':13,'10.0.0.10':13,'10.0.0.11':13,'10.0.0.12':13,'10.0.0.13':15,'10.0.0.14':15,'10.0.0.15':15,'10.0.0.16':15}
    19:{'10.0.0.1':10,'10.0.0.2':10,'10.0.0.3':10,'10.0.0.4':10,'10.0.0.5':12,'10.0.0.6':12,
    '10.0.0.7':12,'10.0.0.8':12,'10.0.0.9':14,'10.0.0.10':14,'10.0.0.11':14,'10.0.0.12':14,
    '10.0.0.13':16,'10.0.0.14':16,'10.0.0.15':16,'10.0.0.16':16},
    20:{'10.0.0.1':10,'10.0.0.2':10,'10.0.0.3':10,'10.0.0.4':10,'10.0.0.5':12,'10.0.0.6':12,
    '10.0.0.7':12,'10.0.0.8':12,'10.0.0.9':14,'10.0.0.10':14,'10.0.0.11':14,'10.0.0.12':14,
    '10.0.0.13':16,'10.0.0.14':16,'10.0.0.15':16,'10.0.0.16':16},
}

# LPR/RSR下输出H{x%16}→ H{(x+4)%16}及H{x%16} → H{(x+5)%16}的路径，其中x为学号的后两位 (22307110187)
self.key = [('10.0.0.7','10.0.0.11'), ('10.0.0.7','10.0.0.12')]
```

因为注意到每个 host 其实只有一个父亲交换机，所以 self.hosts 直接按框架代码所提供即可。然后所有的交换机可以按高度分为三层，s1—s8, s9—s16, s17—s20，然后 s1—s16 都是有两个父亲交换机，其实就相当于建一张图，将他们的父亲交换机都初始化链接起来。最后 s17—s20 是顶层，只能往下了，并且按照虚线方框分为四组的话，每个 s17—s20 都能链接这 4 组，也就是其实 s17 到 s20 往下走的话能走到任意一个 host。
对于两个 host 之间的路径规划，我最开始想的其实是类似“寻找公共祖先”算法，将整个图看作一个树，引入一个虚拟的 root 节点连接 s17—s20，然后分别从 src_host 和 dst_host 出发寻找最近公共祖先，这个应该是能做的。但其实后面想了想直接从 src 动态的往 dst 路上找也很简单，而且在最后 LLR 情况下要计算所有路径的最小 max_cost，还是这样动态的从源点到终点，类似 dfs 这样好搞一点，要是从两路往上汇聚，最后路径上 max_cost 还有些不好说。
所以最终我的算法思路就是从 src 往 dst 找，由于看成树结构来说，这是一个深度先减小后增加的过程，所以有一段路是从上往下，就要记录一下后代节点比较好搞，所以初始化部分也加上了 self.son，记录 8 以上每个交换机所能到达的 host，以及他们的下一跳（第三层，s1--s8）是谁。

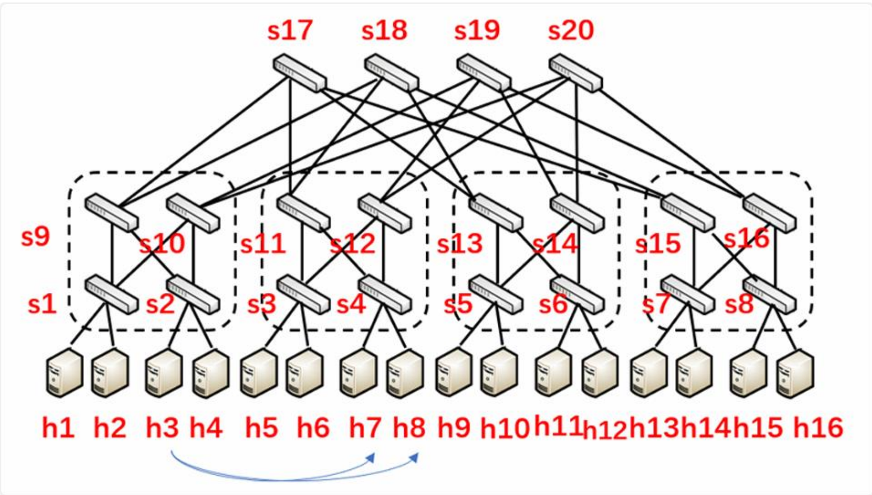
算法核心思路如上。

```python
def calculate(self, src, dst):
    dpid = self.hosts[src][0]  # 直接获取src host的父亲交换机
    while True:
        self.path[(src,dst)].append(dpid)
        if(1 <= dpid and dpid <= 8):  # 对于第三层的switches，往上走/往下走
            if (self.hosts[dst][0] == dpid):  # src和dst在一起，有同样的父亲交换机 or 从上往下走到dst父亲时，结束循环
                break
            else:
                dpid=self.parent[dpid][0]  # 每次都优先往左边走
        else:
            if (dst in self.son[dpid]):  # 往下走的部分，寻找dst在不在当前dpid节点的后辈节点中
                dpid=self.son[dpid][dst]  # 有dst的ip地址映射到下一个dpid
            else:  # 走到第一层了，依旧是优先往左走
                dpid=self.parent[dpid][0]
    # 找到任务要求的src--dst的路径，打印出来路径信息
    if (src,dst) in self.key:
        print("h%d ->" % (int(src.split('.')[-1])), end=" ")
        for i in self.path[(src, dst)]:
            print("s%d ->" % (i), end=" ")
        print("h%d" % (int(dst.split('.')[-1])))
```

从 src 的父亲交换机开始，作为第一个 dpid，往上走，且每一步都优先往左边走——parent[dpid][0]，0 是左边节点 1 是右边节点，直到走到某一个交换机 s，s 的 son 集合中包含了 dst host 的 key，则可以开始往下走了，在 son 集合中取 key 对应的 value，value 在 son 的初始化中定义为当前 s 能达到 dst host 的下一跳，直到最终又跳下第三层。
（12.10 更新了更简单的逻辑:）

```python
while True:
    self.path[(src, dst)].append(dpid)
    if self.hosts[dst][0] == dpid:  # 如果当前dpid已经是目标host的所属交换机，则退出
        break
    if dst not in self.son[dpid]:  # 如果dst不在当前dpid的子节点中
        dpid = self.parent[dpid][0]  # 往上走，选择父亲交换机的左节点
    else:
        dpid = self.son[dpid][dst]  # dst在子节点中，往下走
```

因为考虑到，如果顶层交换机和 dst host 已经确定，则下来的路仅有唯一一条，所以无需分类，只用分是在往上走还是往下走即可。
简单举例:




Src 为 h7，dst 为 h11（我的 key 其中之一）
第一个 dpid：h7 的直接父亲交换机 s4
第二个 dpid：由于 h11 不在 s4 的后辈集合中，继续往上，选择左边，s11
第三个 dpid：由于 h11 不在 s11 后辈中，再上，选左边，s17
验证改版后代码思路：s17 确定，dst 为 h11 确定，仅有一条路：
s17—s13—s6—h11
于是规划出完整路径。
代码的核心思路已经结束，接下来还有若干函数需要补充，就是要理解掌握 ryu 框架下相关的语法即可。
引入了一个 totalpath 字典作为缓存机制，所以计算过的 path 就直接留在里面就好，每次 get_outport 的时候要是查询的路径有再 totalpath 中，就不用再重复计算了。

```
def get_outport(self, dpid, src, dst):
    if (src, dst) not in self.totalpath:  # 开始规划路径
        self.calculate(src, dst)
    pth = self.totalpath[(src,dst)]
    for i in range(len(pth)):
        if pth[i] == dpid:  # 找到要求的dpid, 如果下一条是dst host, 返回dst host, 否则按adjacency矩阵返回
            return (i == len(pth)-1) ? self.hosts[dst][1] : self.adjacency[dpid][pth[i+1]]
```

在路径 src---dst 中寻找 dpid 的 outport
_packet_in_handler 函数就是核心功能函数，大体框架上按照实验文档来写，就是 ip 包和 arp 包要区分一下，解析方式不一样。
文档定义的是 mac_to_port 二维表，我直接定义字典类 totalpath，作用是相同的，记录计算路径和路径上节点关系，在这里获取到每个需要的 dpid 的 outport，其余部分完全参考实验文档即可，理解 ryu 框架代码：

```
if eth.ethertype == ether_types.ETH_TYPE_IP:
    ipv4_pkt = pkt.get_protocol(ipv4.ipv4)
    src = ipv4_pkt.src
    dst = ipv4_pkt.dst
    match = parser.OFPMatch(eth_type=ether_types.ETH_TYPE_IP, in_port=in_port, ipv4_src=src, ipv4_dst=dst)
elif eth.ethertype == ether_types.ETH_TYPE_ARP:
    arp_pkt = pkt.get_protocol(arp.arp)
    src = arp_pkt.src_ip
    dst = arp_pkt.dst_ip
    match = parser.OFPMatch(eth_type=ether_types.ETH_TYPE_ARP, in_port=in_port, arp_spa=src, arp_tpa=dst)
else:  # 包括lldp情况, 忽略lldp packet
    return

out_port = self.get_outport(dpid, src, dst)  # 获得dpid在src-dst路径上的的outport
```

Sudo -I 模式下

ryu-manager LPR.py –observe-links

python3.10 parallel_traffic_generator.py

(mn -c　清理上一轮 mininet 未结束的拓扑)

启动 ryu-manager：

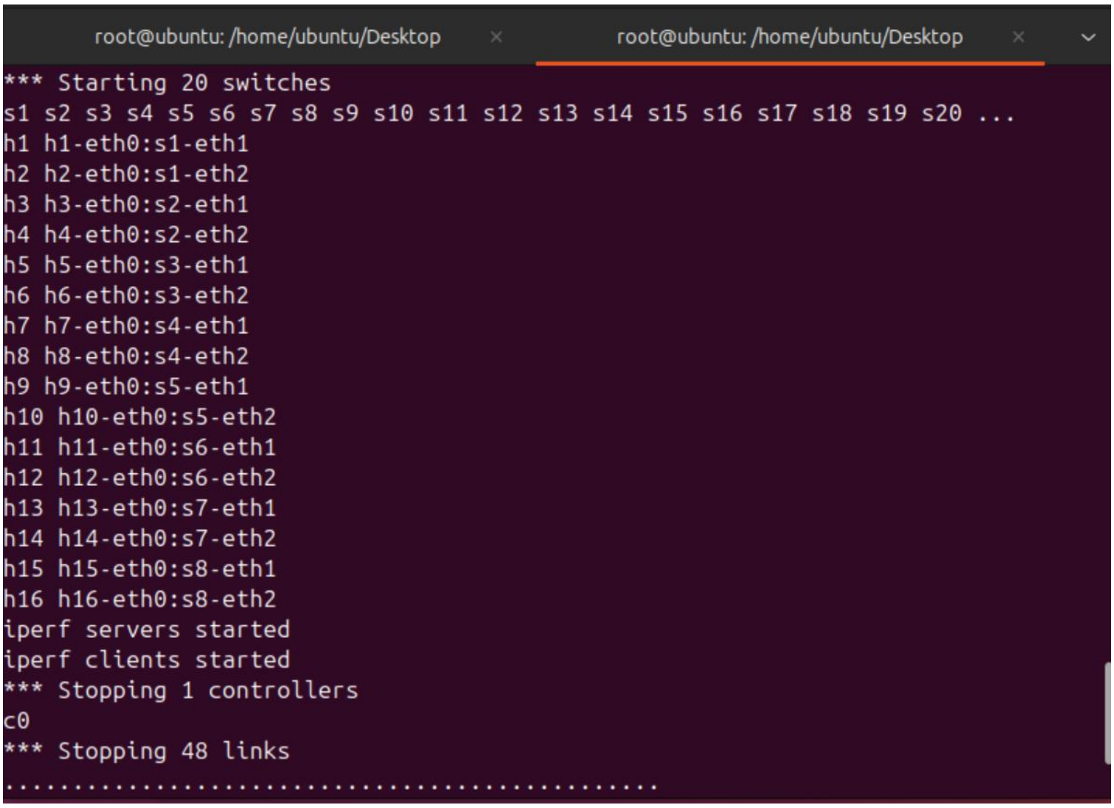
```
(myenv) root@ubuntu:/home/ubuntu/Desktop# ryu-manager LPR.py --observe-links
9 RLock(s) were not greened, to fix this error make sure you run eventlet.monkey
_patch() before importing any other modules.
loading app LPR.py
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
instantiating app LPR.py of ProjectController
instantiating app ryu.topology.switches of Switches
instantiating app ryu.controller.ofp_handler of OFPHandler
```

在另一个终端运行 parallel.py 文件（使用 parallel_traffic_generator.py 测试 LPR.py 任务）：

构建 hosts，links，开启交换机。

持续 100s 结束

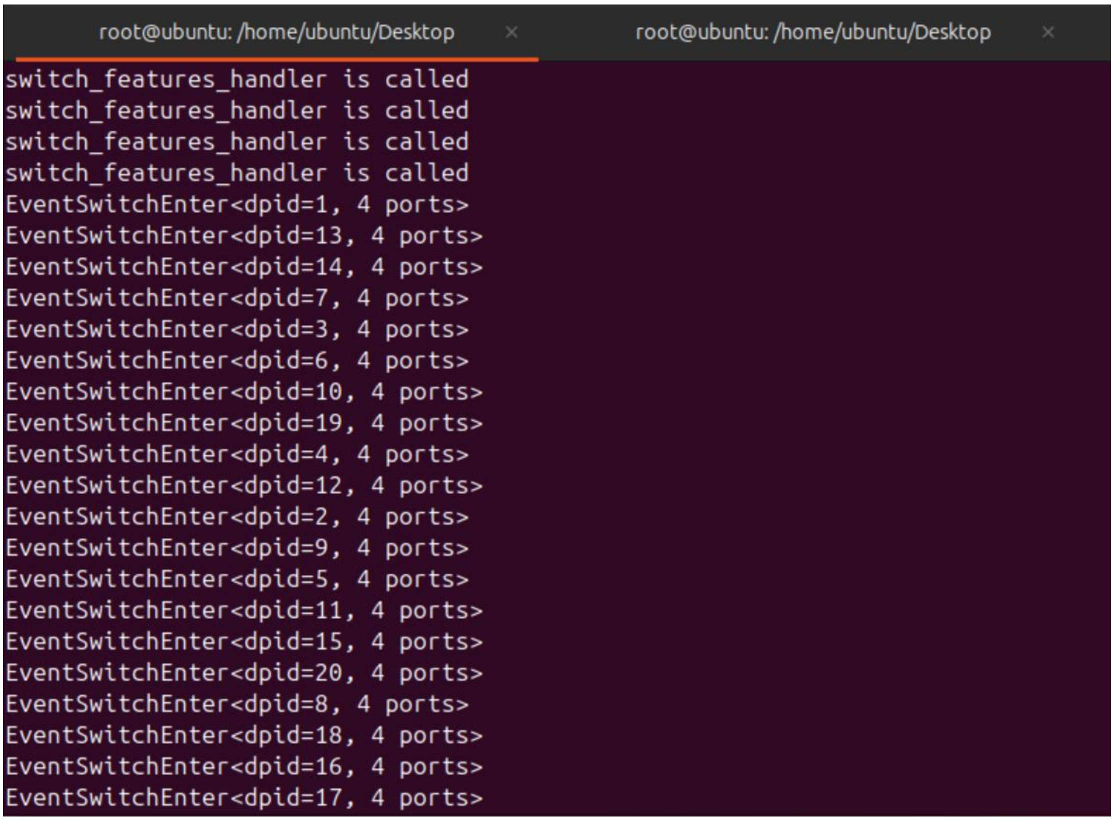
```
rm -f ~/.ssh/mn/*
*** Cleanup complete.
(myenv) root@ubuntu:/home/ubuntu/Desktop# python3.10 parallel.py
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
*** Adding switches:
s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 s16 s17 s18 s19 s20
*** Adding links:
(h1, s1) (h2, s1) (h3, s2) (h4, s2) (h5, s3) (h6, s3) (h7, s4) (h8, s4) (h9, s5)
 (h10, s5) (h11, s6) (h12, s6) (h13, s7) (h14, s7) (h15, s8) (h16, s8) (s9, s1)
(s9, s2) (s9, s17) (s9, s18) (s10, s1) (s10, s2) (s10, s19) (s10, s20) (s11, s3)
 (s11, s4) (s11, s17) (s11, s18) (s12, s3) (s12, s4) (s12, s19) (s12, s20) (s13,
 s5) (s13, s6) (s13, s17) (s13, s18) (s14, s5) (s14, s6) (s14, s19) (s14, s20) (
s15, s7) (s15, s8) (s15, s17) (s15, s18) (s16, s7) (s16, s8) (s16, s19) (s16, s2
0)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
*** Starting controller
c0
*** Starting 20 switches
s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 s16 s17 s18 s19 s20 ...
h1 h1-eth0:s1-eth1
```




按照要求：# LPR/RSR 下输出 H{x%16}→ H{(x+4)%16}及 H{x%16} → H{(x+5)%16}的路径，其中 x 为学号的后两位（22307110187）

所以我的任务：[('10.0.0.7','10.0.0.11'), ('10.0.0.7','10.0.0.12')]

另一个终端 ryu-manager 运行我的 LPR.py 文件结果如下：



```
EventSwitchEnter<dpid=17, 4 ports>
h7 -> s4 -> s11 -> s17 -> s13 -> s6 -> h11
h7 -> s4 -> s11 -> s17 -> s13 -> s6 -> h12
EventSwitchLeave<dpid=19, 0 ports>
EventSwitchLeave<dpid=1, 0 ports>
EventSwitchLeave<dpid=6, 0 ports>
EventSwitchLeave<dpid=14, 0 ports>
EventSwitchLeave<dpid=10, 0 ports>
EventSwitchLeave<dpid=13, 0 ports>
EventSwitchLeave<dpid=3, 0 ports>
EventSwitchLeave<dpid=8, 0 ports>
EventSwitchLeave<dpid=15, 0 ports>
EventSwitchLeave<dpid=7, 0 ports>
EventSwitchLeave<dpid=4, 0 ports>
EventSwitchLeave<dpid=20, 0 ports>
EventSwitchLeave<dpid=12, 0 ports>
EventSwitchLeave<dpid=5, 0 ports>
EventSwitchLeave<dpid=9, 0 ports>
EventSwitchLeave<dpid=11, 0 ports>
EventSwitchLeave<dpid=18, 0 ports>
EventSwitchLeave<dpid=16, 0 ports>
EventSwitchLeave<dpid=2, 0 ports>
EventSwitchLeave<dpid=17, 4 ports>
```

计算结果如上所示:

h7—s4—s11—s17—s13—s6—h11

h7—s4—s11—s17—s13—s6—h12

由原图比对可知完全正确，所有的流都从最左边的路径到达目的地。

**RSR.py:**

代码仅一处改动即可:

```python
def calculate(self, src, dst):
    dpid = self.hosts[src][0]  # 由图可知作为host的src的父亲只有一个
    while True:
        self.totalpath[(src, dst)].append(dpid)
        if self.hosts[dst][0] == dpid:  # 如果当前dpid已经是目标host的所属交换机，则退出
            break
        if dst not in self.son[dpid]:  # 如果dst不在当前dpid的子节点中
            dpid = self.parent[dpid][random.randint(0,1)]  # 往上走，随机选择父亲交换机节点
        else:
            dpid = self.son[dpid][dst]  # dst在子节点中，往下走
```

随机选择 0 或 1，向上走。

实验过程与上述同理，最终 parallel.py 程序运行 100s 结束

```
h5 h5-eth0:s3-eth1
h6 h6-eth0:s3-eth2
h7 h7-eth0:s4-eth1
h8 h8-eth0:s4-eth2
h9 h9-eth0:s5-eth1
h10 h10-eth0:s5-eth2
h11 h11-eth0:s6-eth1
h12 h12-eth0:s6-eth2
h13 h13-eth0:s7-eth1
h14 h14-eth0:s7-eth2
h15 h15-eth0:s8-eth1
h16 h16-eth0:s8-eth2
iperf servers started
iperf clients started
*** Stopping 1 controllers
c0
*** Stopping 48 links
................................................
*** Stopping 20 switches
s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 s16 s17 s18 s19 s20
*** Stopping 16 hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
*** Done
(myenv) root@ubuntu:/home/ubuntu/Desktop#
```

在 ryu-manager 的界面显示出 enter 和 leave 的信息，以及打印的随机通路：

```
switch_features_handler is called
switch_features_handler is called
switch_features_handler is called
switch_features_handler is called
switch_features_handler is called
EventSwitchEnter<dpid=7, 4 ports>
EventSwitchEnter<dpid=15, 4 ports>
EventSwitchEnter<dpid=12, 4 ports>
EventSwitchEnter<dpid=3, 4 ports>
EventSwitchEnter<dpid=6, 4 ports>
EventSwitchEnter<dpid=1, 4 ports>
EventSwitchEnter<dpid=4, 4 ports>
EventSwitchEnter<dpid=13, 4 ports>
EventSwitchEnter<dpid=19, 4 ports>
EventSwitchEnter<dpid=14, 4 ports>
EventSwitchEnter<dpid=8, 4 ports>
EventSwitchEnter<dpid=10, 4 ports>
EventSwitchEnter<dpid=5, 4 ports>
EventSwitchEnter<dpid=17, 4 ports>
EventSwitchEnter<dpid=16, 4 ports>
EventSwitchEnter<dpid=20, 4 ports>
EventSwitchEnter<dpid=2, 4 ports>
EventSwitchEnter<dpid=18, 4 ports>
EventSwitchEnter<dpid=11, 4 ports>
```

```
EventSwitchEnter<dpid=9, 4 ports>
h7 -> s4 -> s11 -> s18 -> s13 -> s6 -> h11
h7 -> s4 -> s11 -> s17 -> s13 -> s6 -> h12
EventSwitchLeave<dpid=17, 4 ports>
EventSwitchLeave<dpid=13, 4 ports>
switch_features_handler is called
switch_features_handler is called
EventSwitchEnter<dpid=17, 4 ports>
EventSwitchEnter<dpid=13, 4 ports>
EventSwitchLeave<dpid=13, 0 ports>
EventSwitchLeave<dpid=17, 0 ports>
EventSwitchLeave<dpid=11, 4 ports>
EventSwitchLeave<dpid=1, 4 ports>
EventSwitchLeave<dpid=20, 4 ports>
EventSwitchLeave<dpid=19, 4 ports>
EventSwitchLeave<dpid=12, 4 ports>
EventSwitchLeave<dpid=14, 4 ports>
EventSwitchLeave<dpid=8, 4 ports>
EventSwitchLeave<dpid=18, 4 ports>
EventSwitchLeave<dpid=5, 4 ports>
EventSwitchLeave<dpid=2, 4 ports>
EventSwitchLeave<dpid=4, 4 ports>
EventSwitchLeave<dpid=7, 4 ports>
EventSwitchLeave<dpid=15, 4 ports>
```

路径分别为

H7—s4—s11—s18—s13—s6—h11

H7—s4—s11—s17—s13—s6—h12

由原图比对可知完全正确，以随机路径到达目的地。

**LLR.py：**

寻找路径的最大 cost 的边，如前所说，考虑要要是知道是某个交换机和某个 host，则从该交换机到 host 的路径只有唯一一条，所以，对于确定了某个路径来说，我们只需要找到其 top 交换机，然后分别沿着两条路下降到 src 和 dst，计算最大 cost。

```python
def cal1(self, dpid:int, ip:str):
    res = 0
    while self.hosts[ip][0] != dpid:
        son = self.son[dpid][ip]
        res = max(res, self.costs[(dpid, son)])
        dpid = son
    return res

def cal2(self, dpid:int, ip1:str, ip2:str):
    return max(self.cal1(dpid, ip1), self.cal1(dpid, ip2))
```

计算路径分三类，对应三层交换机（其实这里是由于实验拓扑结构只有三层取巧了，更一般的做法应该写一个寻找公共祖先算法，但是这里就直接简单实现算了）

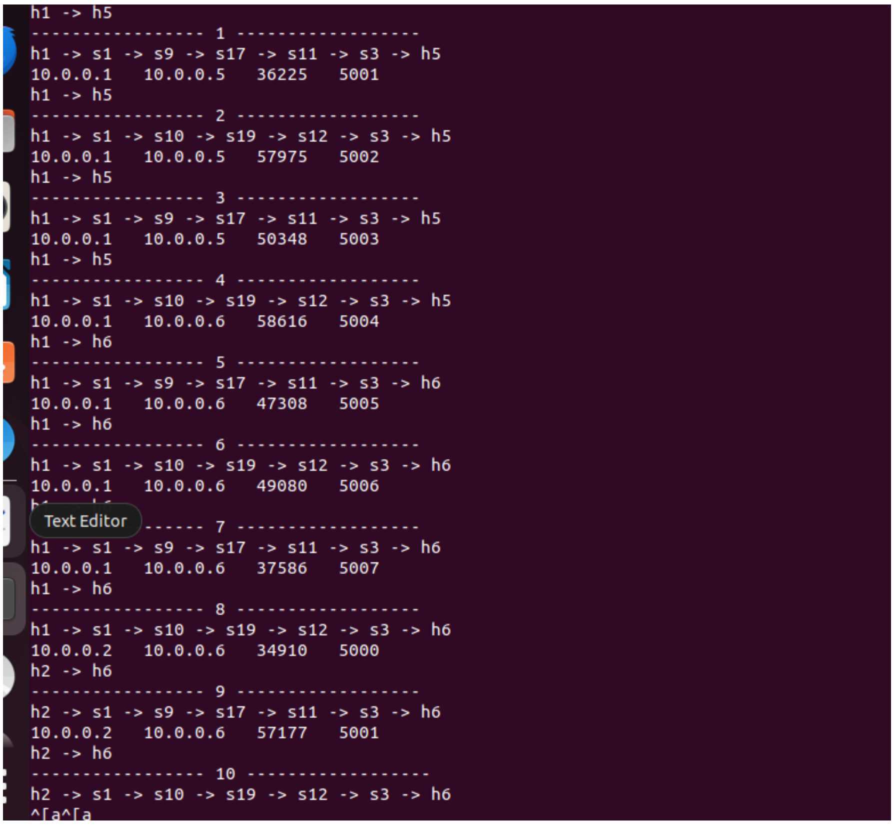
```python
def cal3(self, src, dst, src_port, dst_port):
    dpid_src = self.hosts[src][0]
    dpid_dst = self.hosts[dst][0]
    if dpid_src == dpid_dst:
        self.path[(src, dst, src_port, dst_port)].append(dpid_src)
    elif self.parent[dpid_src]== self.parent[dpid_dst]:
        pair_parent = self.parent[dpid_src]
        p0 = pair_parent[0]
        p1 = pair_parent[1]
        self.path[(src, dst, src_port, dst_port)].append(dpid_src)
        if self.cal2(p0, src, dst) < self.cal2(p1, src, dst):
            self.path[(src, dst, src_port, dst_port)].append(p0)
        else:
            self.path[(src, dst, src_port, dst_port)].append(p1)
        self.path[(src, dst, src_port, dst_port)].append(dpid_dst)
    else:
        min_cost = 114514
        top_dpid = 17
        self.path[(src, dst, src_port, dst_port)].append(dpid_src)
        for i in range(17,21):
            current_cost = self.cal2(i, src, dst)
            if current_cost < min_cost:
                min_cost = current_cost
                top_dpid = i
        self.path[(src, dst, src_port, dst_port)].append(self.son[top_dpid][src])
        self.path[(src, dst, src_port, dst_port)].append(top_dpid)
        self.path[(src, dst, src_port, dst_port)].append(self.son[top_dpid][dst])
        self.path[(src, dst, src_port, dst_port)].append(dpid_dst)
```

以最后 else 情况作为例子讲一下：就是它 src 和 dst 的公共祖先到最顶层的交换机了，最顶层的四个交换机是可以到任意 host 的，所以遍历四个交换机，分别计算路径最大 cost（前面 cal2 函数已有介绍，因为路径唯一）然后在这四个最大 cost 中选择最小的作为目标路径。而还有一个要求就是最大 cost 相同的时候要以左边原则来走，我们遍历是从 17 到 20，且严格小于才更新，也就保证了相等时从左边走的原则。最终更新路径。

```python
if eth.ethertype == ether_types.ETH_TYPE_IP:
    ipv4_pkt = pkt.get_protocol(ipv4.ipv4)
    src = ipv4_pkt.src
    dst = ipv4_pkt.dst
    if ipv4_pkt.proto == 6:
        _tcp = pkt.get_protocol(tcp.tcp)
        match = parser.OFPMatch(eth_type = ether_types.ETH_TYPE_IP, in_port = in_port,
            ipv4_src = src, ipv4_dst = dst, tcp_src = _tcp.src_port, tcp_dst = _tcp.dst_port)
        src_port = _tcp.src_port
        dst_port = _tcp.dst_port
    if ipv4_pkt.proto == 17:
        _udp = pkt.get_protocol(udp.udp)
        match = parser.OFPMatch(eth_type = ether_types.ETH_TYPE_IP, in_port = in_port,
            ipv4_src = src, ipv4_dst = dst, udp_src = _udp.src_port, udp_dst = _udp.dst_port)
        src_port = _udp.src_port
        dst_port = _udp.dst_port
    if ipv4_pkt.proto == 132:
        _sctp = pkt.get_protocol(sctp.sctp)
        match = parser.OFPMatch(eth_type = ether_types.ETH_TYPE_IP, in_port = in_port,
            ipv4_src = src, ipv4_dst = dst, sctp_src = _sctp.src_port, sctp_dst = _sctp.dst_port)
        src_port = _sctp.src_port
        dst_port = _sctp.dst_port
```

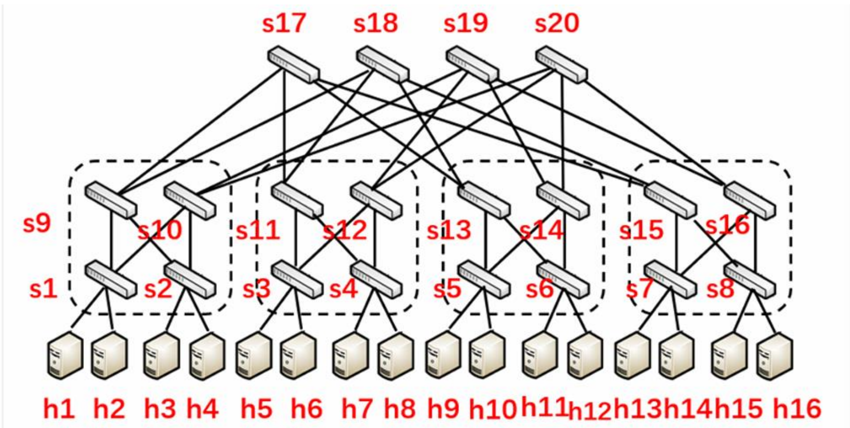
类型区分分别获取 match 和源目的端口号。

使用 sequential_traffic_generator.py 测试 LLR.py
Sequential 同样打开 mininet 建立拓扑，运行 100s 结束。
Ryu-manager 测试 LLR.py 结果如下：



```
h1 -> h5
----------------- 1 ------------------
h1 -> s1 -> s9 -> s17 -> s11 -> s3 -> h5
10.0.0.1   10.0.0.5   36225   5001
h1 -> h5
----------------- 2 ------------------
h1 -> s1 -> s10 -> s19 -> s12 -> s3 -> h5
10.0.0.1   10.0.0.5   57975   5002
h1 -> h5
----------------- 3 ------------------
h1 -> s1 -> s9 -> s17 -> s11 -> s3 -> h5
10.0.0.1   10.0.0.5   50348   5003
h1 -> h5
----------------- 4 ------------------
h1 -> s1 -> s10 -> s19 -> s12 -> s3 -> h5
10.0.0.1   10.0.0.6   58616   5004
h1 -> h6
----------------- 5 ------------------
h1 -> s1 -> s9 -> s17 -> s11 -> s3 -> h6
10.0.0.1   10.0.0.6   47308   5005
h1 -> h6
----------------- 6 ------------------
h1 -> s1 -> s10 -> s19 -> s12 -> s3 -> h6
10.0.0.1   10.0.0.6   49080   5006
Text Editor ------ 7 ------------------
h1 -> s1 -> s9 -> s17 -> s11 -> s3 -> h6
10.0.0.1   10.0.0.6   37586   5007
h1 -> h6
----------------- 8 ------------------
h1 -> s1 -> s10 -> s19 -> s12 -> s3 -> h6
10.0.0.2   10.0.0.6   34910   5000
h2 -> h6
----------------- 9 ------------------
h2 -> s1 -> s9 -> s17 -> s11 -> s3 -> h6
10.0.0.2   10.0.0.6   57177   5001
h2 -> h6
----------------- 10 ------------------
h2 -> s1 -> s10 -> s19 -> s12 -> s3 -> h6
^[a^[a
```

如上所示，跑了多次，前十个流 src 的 dst 相同的情况下，结果应该是相同的。




1—5：最开始都 0，走左边优先：1—9—17—11—3

（1 9）（9 17）（17 11）（11 3）的 cost 为 1 了

1—5：1—10—19—12—3 （因为这四个边 cost 都是 0，比 1 小）

（3 12）（12 19）（19 10）(10 1) cost 为 1

1—5：因为 1 往上走不管咋样要么 1 9 要么 1 10，cost 都是 1 了，所以还是遵循左边优先原则：1—9—17—11—3

然后（1 9）（9 17）（17 11）（11 3）的 cost 为 2 了

1—5：与第二个流情况相同，1<2：1—10—19—12—3

（3 12）（12 19）（19 10）(10 1) cost 为 2

……

后面同理。