

SDN Experiment 2

实验环境

与第一次实验相同

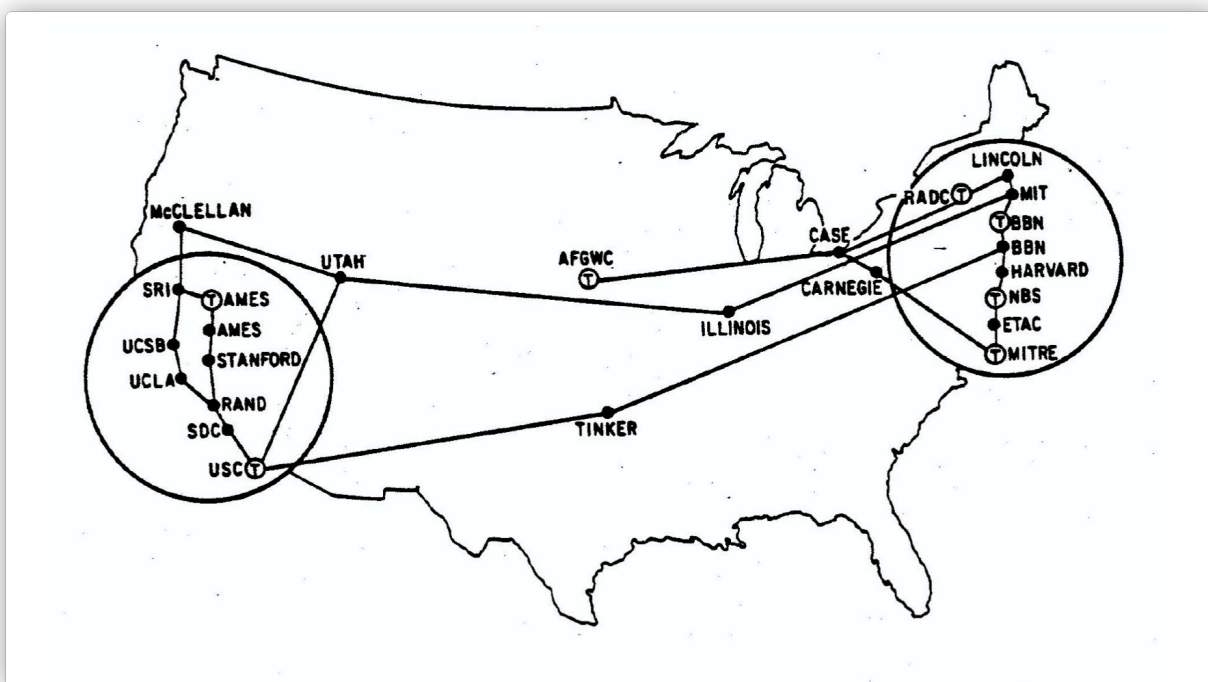
实验内容

第二次实验主要为设计性实验，要求各位在熟悉SDN的基本原理和RYU API的基础上解决下面问题

题目

假如你有一个笔友遍天下爱写信的朋友叫李华，她生活在1972年的UCLA，希望通过ARPAnet（世界第一个数据包交换网络，互联网的鼻祖，接入了25个研究机构，共计55条链路。具体拓扑见下图）发送一封Email给位于MIT的李明同学，现在需要你借助Ryu控制器编写Ryu APP帮助她

1. 为减少网络中节点的中转，希望找到一条从UCLA到MIT跳数最少的连接，输出经过的路线
2. 为了尽快发送Email，希望能找到一条从UCLA到MIT时延最短的连接，输出经过的路线及总的时延，利用Ping包的RTT验证你的结果（此问题选做）



说明

- 上述拓扑为ARPAnet1972.3，源自[The Internet Zoo](#)，借助[assessing-mininet](#)转化成Mininet拓扑，做了一些修改（加入时延，修改名称等）作为实验拓扑
- 上述拓扑中存在环路，你需要解决ARP包的洪泛问题，你可以选择参考生成树协议（STP），但更推荐通过控制器的逻辑来解决，两种参考的思路如下：

一种思路是：我们通过Ryu的API可以发现全局的拓扑信息，将交换机的端口信息记录下来，当控制器收到一个未学习的Arp Request时，直接发给所有交换机连接主机的那些端口，这样我们可以减少数据包在网络中的无意义的洪泛；

另一种思路是利用(dpid, mac, dstination ip)作为键值记录对应的port,每个交换机第一次收到广播的Arp Request时记录下来, 下一次收到键值相同但是 port 不同的 Arp Request直接将包丢弃, 从而避免了洪泛 (由19级同学提出)

- Ryu通过LLDP报文发现拓扑中的交换机，主机发现则需要主机主动发包，相关API的使用参考如下：

```

from ryu.base import app_manager
from ryu.ofproto import ofproto_v1_3
from ryu.controller.handler import set_ev_cls
from ryu.controller.handler import MAIN_DISPATCHER, CONFIG_DISPATCHER
from ryu.controller import ofp_event
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib import hub
from ryu.topology.api import get_all_host, get_all_link, get_all_switch

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

    def __init__(self, *args, **kwargs):
        super(NetworkAwareness, self).__init__(*args, **kwargs)
        self.dpid_mac_port = {}
        self.topo_thread = hub.spawn(self._get_topology)

    def add_flow(self, datapath, priority, match, actions):
        dp = datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        inst = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
        mod = parser.OFPFlowMod(datapath=dp, priority=priority, 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)

def _get_topology(self):
    while True:
        self.logger.info('\n\n\n')

        hosts = get_all_host(self)
        switches = get_all_switch(self)
        links = get_all_link(self)

        self.logger.info('hosts:')
        for hosts in hosts:
            self.logger.info(hosts.to_dict())

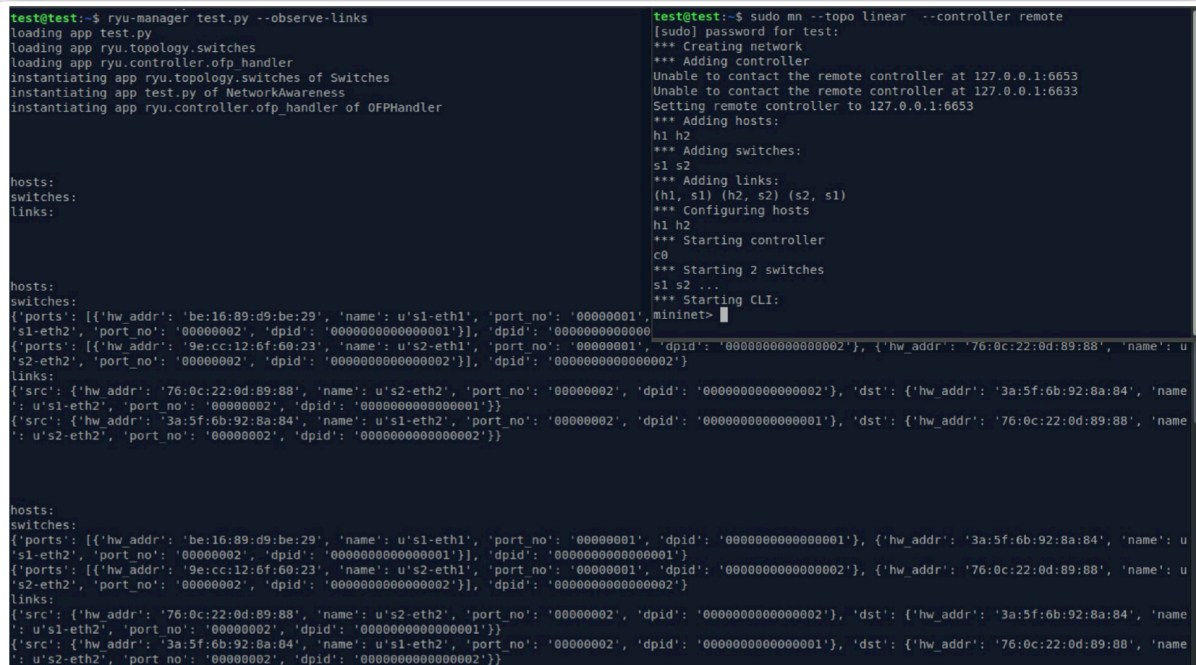
        self.logger.info('switches:')
        for switch in switches:
            self.logger.info(switch.to_dict())

        self.logger.info('links:')
        for link in links:
            self.logger.info(link.to_dict())

        hub.sleep(2)

```

测试结果如下图（提取API打印的信息可查看Ryu源码 `ryu/topology/switches.py` 中类的定义）



```

test@test:~$ ryu-manager test.py --observe-links
loading app test.py
loading app ryu.topology.switches
loading app ryu.controller.ofp.handler
instantiating app ryu.topology.switches of Switches
instantiating app test.py of NetworkAwareness
instantiating app ryu.controller.ofp_handler of OFPHandler

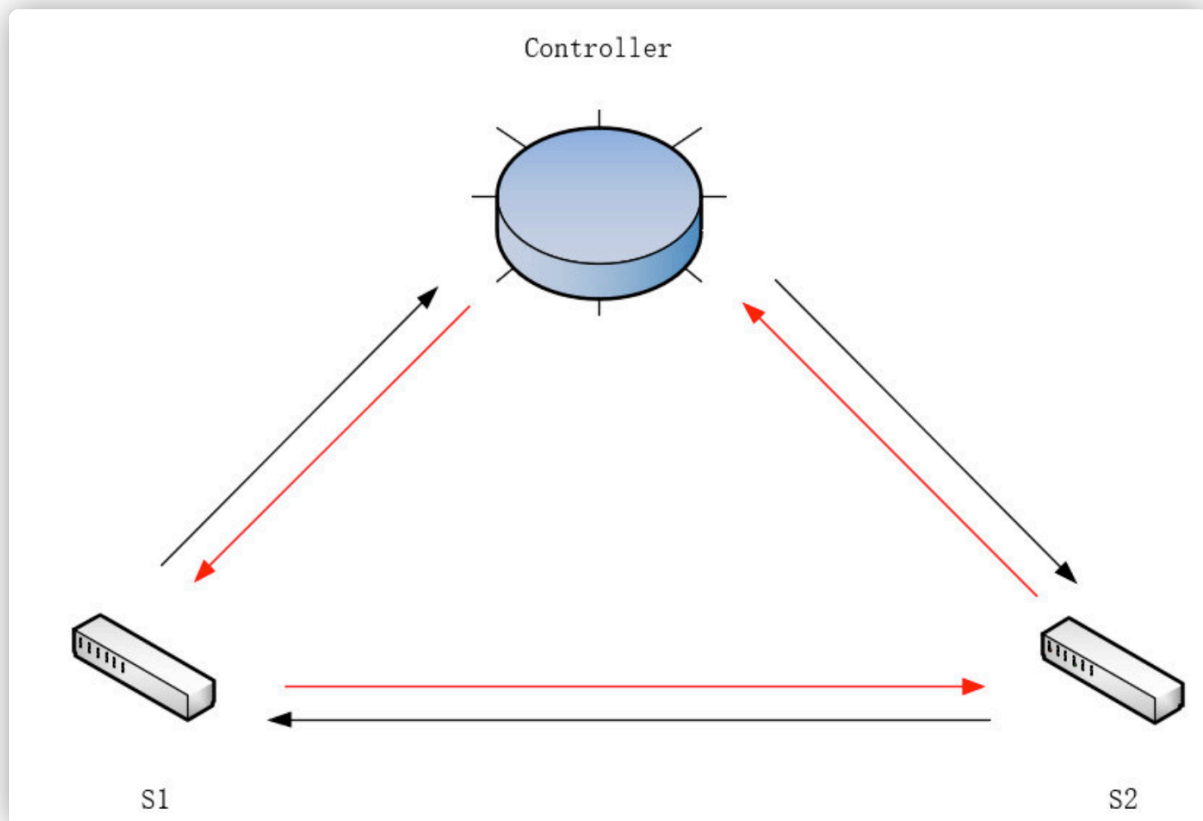
hosts:
switches:
links:

hosts:
switches:
[{'ports': [{'hw_addr': 'be:16:89:d9:be:29', 'name': 'u's1-eth1', 'port_no': '00000001', 'dpid': '0000000000000001'}, {'hw_addr': '3a:5f:6b:92:8a:84', 'name': 'u's1-eth2', 'port_no': '00000002', 'dpid': '0000000000000001'}], 'dpid': '0000000000000000'}, {'ports': [{'hw_addr': '9e:cc:12:6f:60:23', 'name': 'u's2-eth1', 'port_no': '00000001', 'dpid': '0000000000000002'}, {'hw_addr': '76:0c:22:0d:89:88', 'name': 'u's2-eth2', 'port_no': '00000002', 'dpid': '0000000000000002'}], 'dpid': '0000000000000002'}], 'links': [{'src': {'hw_addr': '76:0c:22:0d:89:88', 'name': 'u's2-eth2', 'port_no': '00000002', 'dpid': '0000000000000002'}, 'dst': {'hw_addr': '3a:5f:6b:92:8a:84', 'name': 'u's1-eth2', 'port_no': '00000002', 'dpid': '0000000000000001'}}, {'src': {'hw_addr': '3a:5f:6b:92:8a:84', 'name': 'u's1-eth2', 'port_no': '00000002', 'dpid': '0000000000000001'}, 'dst': {'hw_addr': '76:0c:22:0d:89:88', 'name': 'u's2-eth2', 'port_no': '00000002', 'dpid': '0000000000000002'}}]

hosts:
switches:
[{'ports': [{'hw_addr': 'be:16:89:d9:be:29', 'name': 'u's1-eth1', 'port_no': '00000001', 'dpid': '0000000000000001'}, {'hw_addr': '3a:5f:6b:92:8a:84', 'name': 'u's1-eth2', 'port_no': '00000002', 'dpid': '0000000000000001'}], 'dpid': '0000000000000000'}, {'ports': [{'hw_addr': '9e:cc:12:6f:60:23', 'name': 'u's2-eth1', 'port_no': '00000001', 'dpid': '0000000000000002'}, {'hw_addr': '76:0c:22:0d:89:88', 'name': 'u's2-eth2', 'port_no': '00000002', 'dpid': '0000000000000002'}], 'dpid': '0000000000000002'}], 'links': [{'src': {'hw_addr': '76:0c:22:0d:89:88', 'name': 'u's2-eth2', 'port_no': '00000002', 'dpid': '0000000000000002'}, 'dst': {'hw_addr': '3a:5f:6b:92:8a:84', 'name': 'u's1-eth2', 'port_no': '00000002', 'dpid': '0000000000000001'}}, {'src': {'hw_addr': '3a:5f:6b:92:8a:84', 'name': 'u's1-eth2', 'port_no': '00000002', 'dpid': '0000000000000001'}, 'dst': {'hw_addr': '76:0c:22:0d:89:88', 'name': 'u's2-eth2', 'port_no': '00000002', 'dpid': '0000000000000002'}}]

```

- 对于图的存储及最短路径算法，可自行实现，可使用现有的库（如[networkx](#)）
- 测量链路时延的思路可参考下图（建议先完成基于跳数的最短路径转发后再做下面的部分）



控制器将带有时间戳LLDP报文下发给S1，S1转发给S2，S2上传回控制器（即内圈红色箭头的路径），根据收到的时间和发送时间即可计算出控制器经S1到S2再返回控制器的时延，记为 `lldp_delay_s12`

反之，控制器经S2到S1再返回控制器的时延，记为 `lldp_delay_s21`

我们可以利用Echo Request/Reply报文求出控制器到S1、S2的往返时延，记为 `echo_delay_s1`，`echo_delay_s2`

则S1到S2的时延

$$delay = (lldp_delay_s12 + lldp_delay_s21 - echo_delay_s1 - echo_delay_s2) / 2$$

为此，我们需要对Ryu做如下修改：

1. `ryu/topology/Switches.py` 的 `PortData/__init__()`

`PortData` 记录交换机的端口信息，我们需要增加 `self.delay` 属性记录上述的 `lldp_delay`

`self.timestamp` 为LLDP包在发送时被打上的时间戳，具体发送的逻辑查看源码

```
class PortData(object):
    def __init__(self, is_down, lldp_data):
        super(PortData, self).__init__()
        self.is_down = is_down
        self.lldp_data = lldp_data
        self.timestamp = None
        self.sent = 0
        self.delay = 0
```

2. `ryu/topology/Switches/lldp_packet_in_handler()`

`lldp_packet_in_handler()` 处理接收到的LLDP包，在这里我们用收到LLDP报文的时间戳减去发送时的时间戳即为 `lldp_delay`，由于LLDP报文被设计为经一跳后转给控制器，我们可将 `lldp_delay` 存入发送LLDP包对应的交换机端口

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def lldp_packet_in_handler(self, ev):
    # add receive timestamp
    recv_timestamp = time.time()
    if not self.link_discovery:
        return

    msg = ev.msg
    try:
        src_dpid, src_port_no = LLDPpacket.lldp_parse(msg.data)
    except LLDPpacket.LLDPUnknownFormat:
        # This handler can receive all the packets which can be
        # not-LLDP packet. Ignore it silently
        return

    # calc the delay of lldp packet
    for port, port_data in self.ports.items():
        if src_dpid == port.dpid and src_port_no == port.port_no:
            send_timestamp = port_data.timestamp
            if send_timestamp:
                port_data.delay = recv_timestamp - send_timestamp

    ...
```

完成上述修改后需重新编译安装Ryu，在安装目录下运行 `sudo python setup.py install`

3. 获取 `lldp_delay`

在你们需要完成的计算时延的APP中，利用 `lookup_service_brick` 获取到正在运行的 `switches` 的实例（即步骤12中被我们修改的类），按如下的方式即可获取相应的 `lldp_delay`

```
from ryu.base.app_manager import lookup_service_brick

...

@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def packet_in_handler(self, ev):
    msg = ev.msg
    dpid = msg.datapath.id
    try:
        src_dpid, src_port_no = LLDPpacket.lldp_parse(msg.data)

        if self.switches is None:
            self.switches = lookup_service_brick('switches')
```

```

        for port in self.switches.ports.keys():
            if src_dpid == port.dpid and src_port_no == port.port_no:
                lldp_delay[(src_dpid, dpid)] =
self.switches.ports[port].delay
    except:
        return

```

示例

- 跳数最少的连接

The screenshot shows a terminal window with two main sections. The left section displays a network topology map with nodes and their connections. The right section shows the output of a ping command from node 10.0.0.22 to node 10.0.0.12.

```

host not find/no path
host not find/no path
topo map:
node    ->    node
1       25
1       23
2       3
2       5
3       4
4       11
4       6
5       8
7       16
7       25
8       17
9       16
9       22
10      18
10      14
11      24
12      18
12      23
13      20
13      21
13      15
14      21
15      22
16      17
18      19
19      20
20      10.0.0.22
22      23
24      25
25      10.0.0.12

path: 10.0.0.22 -> 10.0.0.12
10.0.0.22 -> 1:s20:3 -> 3:s19:2 -> 4:s18:3 -> 2:s12:3 -> 3:s23:2 -> 3:s1:2 -> 2:s25:1 -> 10.0.0.12

```

```

Terminal - test@test:~/sdn/assessing-mininet/parser
controller to create all the routes, then do 'pingall' on the mininet c
onsole.

*** edited by zys for xjtu sdn_exp_2019

*** Starting CLI:
mininet> UCLA ping MIT
PING 10.0.0.12 (10.0.0.12) 56(84) bytes of data:
64 bytes from 10.0.0.12: icmp_seq=3 ttl=64 time=277 ms
64 bytes from 10.0.0.12: icmp_seq=4 ttl=64 time=527 ms
64 bytes from 10.0.0.12: icmp_seq=5 ttl=64 time=523 ms
64 bytes from 10.0.0.12: icmp_seq=6 ttl=64 time=523 ms
64 bytes from 10.0.0.12: icmp_seq=7 ttl=64 time=525 ms
64 bytes from 10.0.0.12: icmp_seq=8 ttl=64 time=526 ms
64 bytes from 10.0.0.12: icmp_seq=9 ttl=64 time=526 ms
64 bytes from 10.0.0.12: icmp_seq=10 ttl=64 time=525 ms
64 bytes from 10.0.0.12: icmp_seq=11 ttl=64 time=524 ms
^C
--- 10.0.0.12 ping statistics ---
11 packets transmitted, 9 received, 18% packet loss, time 10059ms
rtt min/avg/max/mdev = 277.046/497.723/527.087/78.036 ms
mininet>

```

拓扑的打印不做要求，打印出经过的交换机即可

因为沉默主机的原因，前几次Ping会丢包为正常现象

- 时延最短的连接

The screenshot shows a terminal window with two main sections. The left section displays a network delay map with nodes, their connections, and the delay in milliseconds. The right section shows the output of a ping command from node 10.0.0.22 to node 10.0.0.12.

```

total path delay: 117.178201675
path: 10.0.0.22 -> 10.0.0.12
10.0.0.22 -> 1:s20:2 -> 2:s13:4 -> 2:s15:3 -> 3:s22:2 -> 2:s9:3 -> 3:s16:2 -> 3:s7:2 -> 3:s25:1 -> 10.0.0.12

delay map:
node    ->    node    delay
1       25          51.60 ms
1       23          36.67 ms
2       3           14.73 ms
2       5           15.61 ms
3       4           16.52 ms
4       11          17.69 ms
4       6           19.25 ms
5       8           11.16 ms
7       16          18.57 ms
7       25          20.25 ms
8       17          17.25 ms
9       16          20.12 ms
9       22          14.57 ms
10      18          15.37 ms
10      14          16.91 ms
11      24          19.47 ms
12      18          42.89 ms
12      23          46.08 ms
13      20          16.36 ms
13      21          19.99 ms
13      15          18.83 ms
14      21          21.30 ms
15      22          17.05 ms
16      17          14.55 ms
18      19          47.06 ms
19      20          50.53 ms
20      10.0.0.22    0.00 ms
22      23          18.29 ms
24      25          14.72 ms
25      10.0.0.12    0.00 ms

```

```

Terminal - test@test:~/sdn/assessing-mininet/parser
controller to create all the routes, then do 'pingall' on the mininet c
onsole.

*** edited by zys for xjtu sdn_exp_2019

*** Starting CLI:
mininet> UCLA ping MIT
PING 10.0.0.12 (10.0.0.12) 56(84) bytes of data:
64 bytes from 10.0.0.12: icmp_seq=3 ttl=64 time=135 ms
64 bytes from 10.0.0.12: icmp_seq=4 ttl=64 time=235 ms
64 bytes from 10.0.0.12: icmp_seq=5 ttl=64 time=226 ms
64 bytes from 10.0.0.12: icmp_seq=6 ttl=64 time=226 ms
64 bytes from 10.0.0.12: icmp_seq=7 ttl=64 time=226 ms
64 bytes from 10.0.0.12: icmp_seq=8 ttl=64 time=226 ms
64 bytes from 10.0.0.12: icmp_seq=9 ttl=64 time=226 ms
64 bytes from 10.0.0.12: icmp_seq=10 ttl=64 time=226 ms
64 bytes from 10.0.0.12: icmp_seq=11 ttl=64 time=226 ms
^C
--- 10.0.0.12 ping statistics ---
11 packets transmitted, 9 received, 18% packet loss, time 10065ms
rtt min/avg/max/mdev = 135.017/217.177/235.072/29.177 ms
mininet>

```

同样拓扑的打印不做要求，打印出经过的交换机和总的路径时延

由图，总路径时延应约等于Ping包RTT的一半

参考

本次实验参考网络资料整理而成