Software Define Network   Lab3

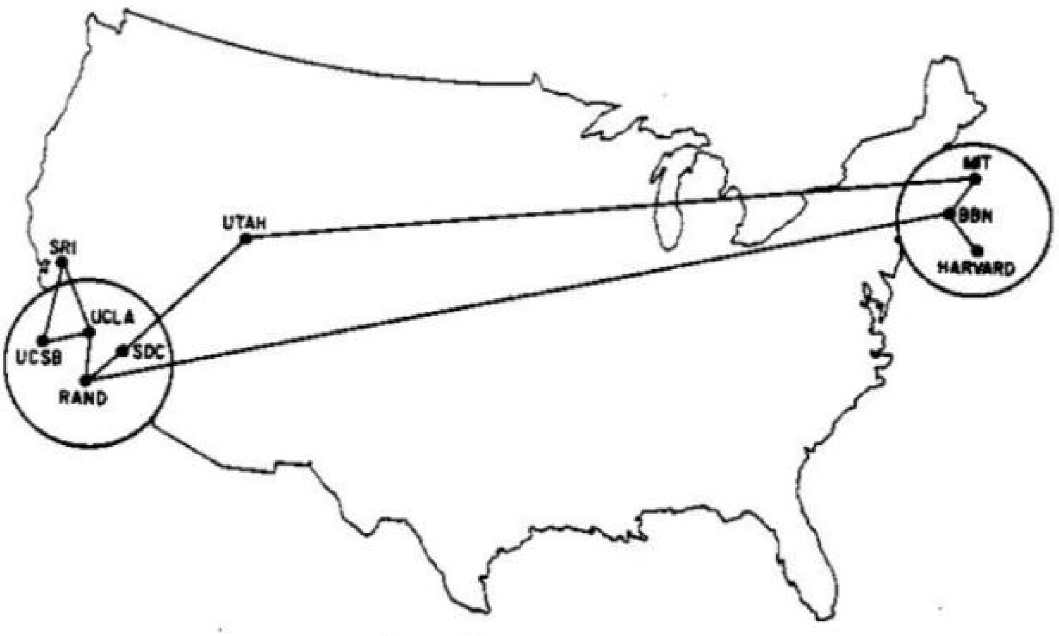
实验目的：

通过本次实验，希望大家掌握以下内容：
学习利用 ryu.topology.api 发现网络拓扑
学习利用 LLDP 和 Echo 数据包测量链路时延
学习计算基于跳数和基于时延的最短路由
学习设计能够容忍链路故障的路由策略
分析网络集中式控制与分布式控制的差异，思考 SDN 的得与失

实验环境：

Windows 10, VMware Workstation Pro, Ubuntu

问题背景：



Juni 1970

来到 1970 年，在你的建设下 ARPANET 飞速发展，在一年内从原来西部 4 个结点组成的简单网络逐渐发展为拥有 9 个结点，横跨东西海岸，初具规模的网络。

ARPANET 的拓展极大地便利了东西海岸之间的通信，但用户仍然十分关心网络服务的性能。一条时延较小的转发路由将显著提升用户体验，尤其是在一些实时性要求很高的应用场景下。另外，路由策略对网络故障的容忍能力也是影响用户体验的重要因素，好的路由策略能够向用户隐藏一定程度的链路故障，使得个别链路断开后用户间的通信不至于中断。

SDN 是一种集中式控制的网络架构，控制器可以方便地获取网络拓扑、各链路和交换机的性能指标、网络故障和拓扑变化等全局信息，这也是 SDN 的优势之一。在掌握全局信息的基础上，SDN 就能实现更高效、更健壮的路由策略。

在正式任务之前，为帮助同学们理解，本指导书直接给出了一个示例。请运行示例程序，理解怎样利用 ryu.topology.api 获取网络拓扑，并计算跳数最少的路由。

跳数最少的路由不一定是最快的路由，在实验任务一中，你将学习怎样利用 LLDP 和 Echo 数据包测量链路时延，并计算时延最小的路由。

1970 年的网络硬件发展尚不成熟，通信链路和交换机端口发生故障的概率较高。在实验任务二中，你将学习在链路不可靠的情况下，设计对链路故障有一定容忍能力的路由策略。

实验过程：

1. 示例：最少跳数路径

拓扑感知

控制器首先要获取网络的拓扑结构，才能够对网络进行各种测量分析，网络拓扑主要包括主机、链路和交换机的相关信息。

将实验指导书中的 NetworkAwareness.py 代码保存。在命令行终端内输入：

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

创建网络拓扑，再在另一个命令行终端内输入：

```
ryu-manager NetworkAwareness.py --observe-links
```

启动 Ryu 控制器

```
sdn@ubuntu:~/Desktop/mininet/custom$ sudo mn --topo=tree,2,2 --co
ntroller=remote
[sudo] password for sdn:
*** 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>
```

```
sdn@ubuntu:~/Desktop/mininet/custom$ ryu-manager NetworkAwareness.p
y --observe-links
loading app NetworkAwareness.py
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
instantiating app NetworkAwareness.py of NetworkAwareness
instantiating app ryu.topology.switches of Switches
instantiating app ryu.controller.ofp_handler of OFPHandler




hosts:
switches:
links:




hosts:
switches:
links:
```

可以看到，一开始 hosts, switches, links 等信息均为空。

```
hosts:
switches:
{'dpid': '0000000000000001', 'ports': [{'dpid': '0000000000000001',
 'port_no': '00000001', 'hw_addr': '2e:b8:e7:18:72:ef', 'name': 's1
-eth1'}, {'dpid': '0000000000000001', 'port_no': '00000002', 'hw_ad
dr': 'b2:1b:e3:b7:11:18', 'name': 's1-eth2'}]}
{'dpid': '0000000000000002', 'ports': [{'dpid': '0000000000000002',
 'port_no': '00000001', 'hw_addr': 'ea:1a:68:06:fa:82', 'name': 's2
-eth1'}, {'dpid': '0000000000000002', 'port_no': '00000002', 'hw_ad
dr': '86:32:8d:6a:f8:3f', 'name': 's2-eth2'}, {'dpid': '00000000000
00002', 'port_no': '00000003', 'hw_addr': 'be:37:bb:9d:15:fe', 'nam
e': 's2-eth3'}]}
{'dpid': '0000000000000003', 'ports': [{'dpid': '0000000000000003',
 'port_no': '00000001', 'hw_addr': '9e:20:29:aa:d4:08', 'name': 's3
-eth1'}, {'dpid': '0000000000000003', 'port_no': '00000002', 'hw_ad
dr': '72:0d:71:2f:0f:fb', 'name': 's3-eth2'}, {'dpid': '00000000000
00003', 'port_no': '00000003', 'hw_addr': 'fa:90:9d:9b:17:cd', 'nam
e': 's3-eth3'}]}
links:
{'src': {'dpid': '0000000000000001', 'port_no': '00000001', 'hw_add
r': '2e:b8:e7:18:72:ef', 'name': 's1-eth1'}, 'dst': {'dpid': '00000
00000000002', 'port_no': '00000003', 'hw_addr': 'be:37:bb:9d:15:fe'
, 'name': 's2-eth3'}}
{'src': {'dpid': '0000000000000001', 'port_no': '00000002', 'hw_add
r': 'b2:1b:e3:b7:11:18', 'name': 's1-eth2'}, 'dst': {'dpid': '00000
00000000003', 'port_no': '00000003', 'hw_addr': 'fa:90:9d:9b:17:cd'
, 'name': 's3-eth3'}}
```

随后 switches 与 links 有信息记录，而 hosts 仍没有信息记录。

沉默主机现象

主机如果没有主动发送过数据包，控制器就无法发现主机。运行前面的 NetworkAwareness.py 时，你可能会看到 host 输出为空，这就是沉默主机现象导致的。你可以在 mininet 中运行 pingall 指令，令每个主机发出 ICMP 数据包，这样控制器就能够发现主机。当然命令的结果是 ping 不通，因为程序中并没有下发路由的代码。

在 mininet 命令行内输入 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)
mininet>
```

可以看到，各节点之间无法 ping 通。


```
hosts:
{'mac': 'b6:eb:60:08:4b:bf', 'ipv4': [], 'ipv6': ['fe80::b4eb:60ff:
fe08:4bbf'], 'port': {'dpid': '0000000000000002', 'port_no': '00000
001', 'hw_addr': 'ea:1a:68:06:fa:82', 'name': 's2-eth1'}}
{'mac': '0e:80:3b:ac:61:09', 'ipv4': [], 'ipv6': ['fe80::c80:3bff:f
eac:6109'], 'port': {'dpid': '0000000000000003', 'port_no': '000000
02', 'hw_addr': '72:0d:71:2f:0f:fb', 'name': 's3-eth2'}}
{'mac': '5e:bd:2a:38:d1:ab', 'ipv4': [], 'ipv6': ['fe80::5cbd:2aff:
fe38:d1ab'], 'port': {'dpid': '0000000000000003', 'port_no': '00000
001', 'hw_addr': '9e:20:29:aa:d4:08', 'name': 's3-eth1'}}
{'mac': '0a:0c:3f:3a:d8:6c', 'ipv4': [], 'ipv6': ['fe80::80c:3fff:f
e3a:d86c'], 'port': {'dpid': '0000000000000002', 'port_no': '000000
02', 'hw_addr': '86:32:8d:6a:f8:3f', 'name': 's2-eth2'}}
switches:
{'dpid': '0000000000000001', 'ports': [{'dpid': '0000000000000001',
 'port_no': '00000001', 'hw_addr': '2e:b8:e7:18:72:ef', 'name': 's1
-eth1'}, {'dpid': '0000000000000001', 'port_no': '00000002', 'hw_ad
dr': 'b2:1b:e3:b7:11:18', 'name': 's1-eth2'}]}
{'dpid': '0000000000000002', 'ports': [{'dpid': '0000000000000002',
 'port_no': '00000001', 'hw_addr': 'ea:1a:68:06:fa:82', 'name': 's2
-eth1'}, {'dpid': '0000000000000002', 'port_no': '00000002', 'hw_ad
dr': '86:32:8d:6a:f8:3f', 'name': 's2-eth2'}, {'dpid': '00000000000
00002', 'port_no': '00000003', 'hw_addr': 'be:37:bb:9d:15:fe', 'nam
e': 's2-eth3'}]}
{'dpid': '0000000000000003', 'ports': [{'dpid': '0000000000000003',
 'port_no': '00000001', 'hw_addr': '9e:20:29:aa:d4:08', 'name': 's3
-eth1'}, {'dpid': '0000000000000003', 'port_no': '00000002', 'hw_ad
dr': '72:0d:71:2f:0f:fb', 'name': 's3-eth2'}, {'dpid': '00000000000
00003', 'port_no': '00000003', 'hw_addr': 'fa:90:9d:9b:17:cd', 'nam
e': 's3-eth3'}]}
links:
{'src': {'dpid': '0000000000000001', 'port_no': '00000001', 'hw_add
r': '2e:b8:e7:18:72:ef', 'name': 's1-eth1'}, 'dst': {'dpid': '00000
00000000002', 'port_no': '00000003', 'hw_addr': 'be:37:bb:9d:15:fe'
, 'name': 's2-eth3'}}
{'src': {'dpid': '0000000000000001', 'port_no': '00000002', 'hw_add
```

运行 pingall 指令后可以看到，hosts, switches, links 等均有信息记录。

2. 计算最少跳数路径

下面第一个函数位于我们给出的 network_awareness.py 文件中，第二个函数位于 shortest_forward.py。核心逻辑是，当控制器接收到携带 ipv4 报文的 Packet_In 消息时，调用 networkx 计算最短路（也可以自行实现，比如迪杰斯特拉算法），然后把相应的路由下发到沿途交换机，具体逻辑可以查看附件代码。shortest_forward.py 未处理环路，请根据你在实验一中处理环路的代码对 handle_arp 函数稍加补充即可。

根据实验指导书的提示，使用 Lab2 中 Broadcast_Loop.py 中的代码，实现自学习交换机与处理环路，修改 shortest_forward.py 中的 handle_arp()函数。

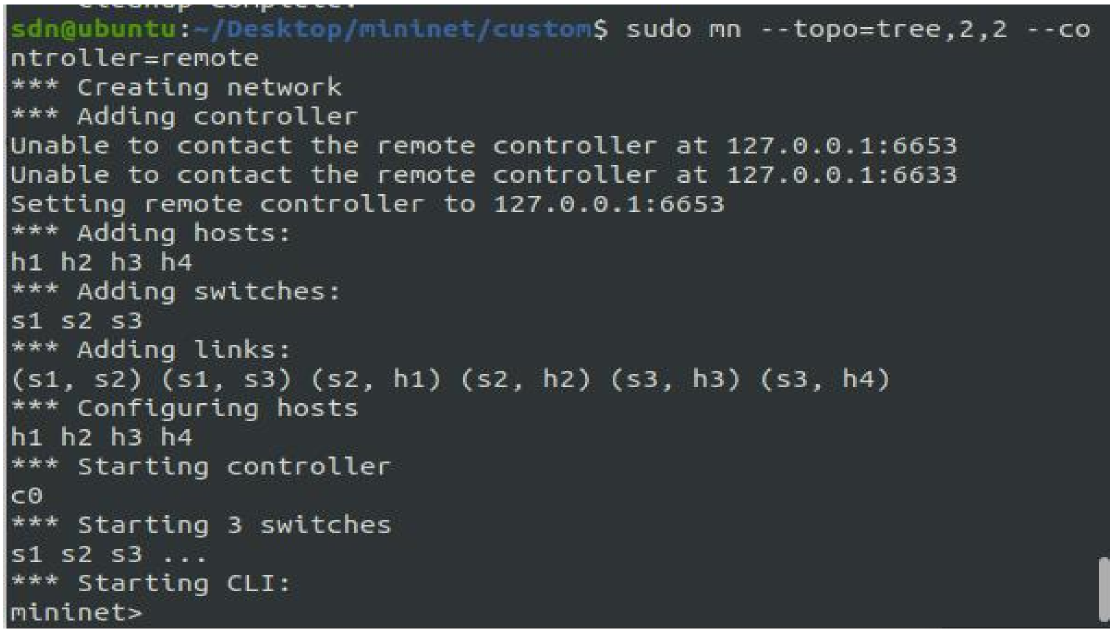
在命令行终端内输入：

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

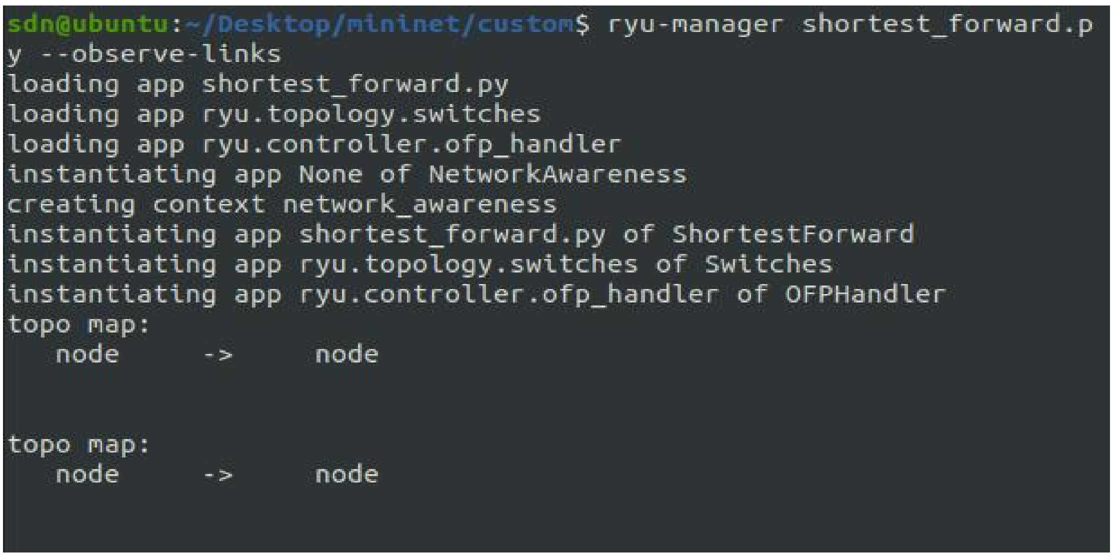
创建网络拓扑，再在另一个命令行终端内输入：

```
ryu-manager shortest_forward.py --observe-links
```

启动 Ryu 控制器



```
sdn@ubuntu:~/Desktop/mininet/custom$ sudo mn --topo=tree,2,2 --co
ntroller=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>
```



```
sdn@ubuntu:~/Desktop/mininet/custom$ ryu-manager shortest_forward.p
y --observe-links
loading app shortest_forward.py
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
instantiating app None of NetworkAwareness
creating context network_awareness
instantiating app shortest_forward.py of ShortestForward
instantiating app ryu.topology.switches of Switches
instantiating app ryu.controller.ofp_handler of OFPHandler
topo map:
   node      ->      node


topo map:
   node      ->      node
```

可以看到，在 mininet 命令行内输入指令前，Ryu 控制器没有信息记录。

在 mininet 命令行内输入 pingall，测试各节点之间的连通性。

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> X h3 X
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 16% dropped (10/12 received)
mininet> 
```

可以看到，只有部分节点能 ping 通。因为沉默主机现象，前几次 ping 可能都会输出 host not find/no path，这属于正常现象。

```
ARP Learning: 2 4e:b3:76:60:0e:62 10.0.0.2 1
topo map:
   node      ->     node
 10.0.0.1            2


host not find/no path
topo map:
   node      ->     node
 10.0.0.1            2
    2             10.0.0.2


topo map:
   node      ->     node
 10.0.0.1            2
    2             10.0.0.2


topo map:
   node      ->     node
 10.0.0.1            2
    2             10.0.0.2
    2                1
    3                1
```

可以看到，在 Ryu 控制器内显示了交换机 ARP 记录的信息，同时也显示了 host not find/no path。

再次在 mininet 命令行内输入 pingall，测试各节点之间的连通性。

```
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)
mininet> 
```

这一次所有 12 个节点能正常 ping 通。

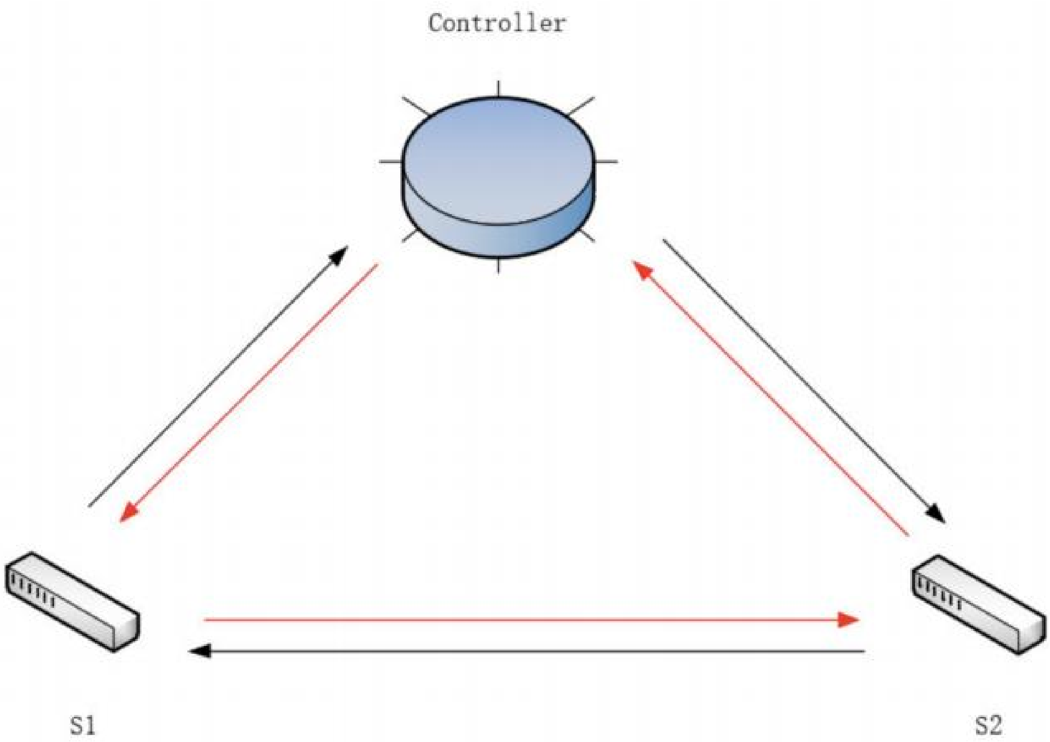
```
path: 10.0.0.1 -> 10.0.0.2
10.0.0.1 -> 1:s2:2 -> 10.0.0.2
ARP Learning: 2 8e:7c:c8:87:33:6f 10.0.0.3 2
ARP Learning: 1 8e:7c:c8:87:33:6f 10.0.0.3 1
ARP Learning: 1 8e:7c:c8:87:33:6f 10.0.0.3 1
ARP Learning: 3 8e:7c:c8:87:33:6f 10.0.0.3 3
ARP Learning: 3 8e:7c:c8:87:33:6f 10.0.0.3 3
ARP Learning: 3 8e:7c:c8:87:33:6f 10.0.0.3 3
path: 10.0.0.2 -> 10.0.0.3
10.0.0.2 -> 2:s2:3 -> 1:s1:2 -> 3:s3:1 -> 10.0.0.3
ARP Learning: 2 8e:7c:c8:87:33:6f 10.0.0.4 2
ARP Learning: 1 8e:7c:c8:87:33:6f 10.0.0.4 1
ARP Learning: 1 8e:7c:c8:87:33:6f 10.0.0.4 1
ARP Learning: 3 8e:7c:c8:87:33:6f 10.0.0.4 3
ARP Learning: 3 8e:7c:c8:87:33:6f 10.0.0.4 3
ARP Learning: 3 8e:7c:c8:87:33:6f 10.0.0.4 3
path: 10.0.0.2 -> 10.0.0.4
10.0.0.2 -> 2:s2:3 -> 1:s1:2 -> 3:s3:2 -> 10.0.0.4
ARP Learning: 3 ae:4b:bd:f9:95:f2 10.0.0.4 1
ARP Learning: 1 ae:4b:bd:f9:95:f2 10.0.0.4 2
ARP Learning: 1 ae:4b:bd:f9:95:f2 10.0.0.4 2
ARP Learning: 2 ae:4b:bd:f9:95:f2 10.0.0.4 3
ARP Learning: 2 ae:4b:bd:f9:95:f2 10.0.0.4 3
ARP Learning: 2 ae:4b:bd:f9:95:f2 10.0.0.4 3
path: 10.0.0.3 -> 10.0.0.4
10.0.0.3 -> 1:s3:2 -> 10.0.0.4
path: 10.0.0.1 -> 10.0.0.4
10.0.0.1 -> 1:s2:3 -> 1:s1:2 -> 3:s3:2 -> 10.0.0.4
path: 10.0.0.1 -> 10.0.0.4
10.0.0.1 -> 1:s2:3 -> 1:s1:2 -> 3:s3:2 -> 10.0.0.4
path: 10.0.0.2 -> 10.0.0.4
10.0.0.2 -> 2:s2:3 -> 1:s1:2 -> 3:s3:2 -> 10.0.0.4
```

同时 Ryu 控制器也显示了节点之间最短路径的记录。

3. 必做题：最小时延路径

跳数最少的路由不一定是最快的路由，链路时延也会对路由的快慢产生重要影响。请实时地（周期地）利用 LLDP 和 Echo 数据包测量各链路的时延，在网络拓扑的基础上构建一个有权图，然后基于此图计算最小时延路径。具体任务是，找出一条从 SDC 到 MIT 时延最短的路径，输出经过的路线及总的时延，利用 Ping 包的 RTT 验证你的结果。

测量原理：链路时延



控制器将带有时间戳的 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。

交换机收到控制器发来的 Echo 报文后会立即回复控制器，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

```
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。

switches.py NetworkAwareness.py shortest_forward.py network_awareness.py

```python
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def lldp_packet_in_handler(self, ev):
        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
        dst_dpid = msg.datapath.id
        if msg.datapath.ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION:
            dst_port_no = msg.in_port
        elif msg.datapath.ofproto.OFP_VERSION >= ofproto_v1_2.OFP_VERSION:
            dst_port_no = msg.match['in_port']
        else:
            LOG.error('cannot accept LLDP. unsupported version. %x',
                      msg.datapath.ofproto.OFP_VERSION)

        src = self._get_port(src_dpid, src_port_no)
        if not src or src.dpid == dst_dpid:
            return
        try:
            self.ports.lldp_received(src)
        except KeyError:
            # There are races between EventOFPPacketIn and
            # EventDPPortAdd. So packet-in event can happend before
```

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

```
sdn@ubuntu:~/Desktop/ryu$ sudo python setup.py install
[sudo] password for sdn:
running install
[pbr] Writing ChangeLog
[pbr] Generating ChangeLog
[pbr] ChangeLog complete (0.1s)
[pbr] Generating AUTHORS
[pbr] AUTHORS complete (0.3s)
running build
running build_py
running egg_info
writing pbr to ryu.egg-info/pbr.json
writing ryu.egg-info/PKG-INFO
writing dependency_links to ryu.egg-info/dependency_links.txt
writing entry points to ryu.egg-info/entry_points.txt
writing requirements to ryu.egg-info/requires.txt
writing top-level names to ryu.egg-info/top_level.txt
[pbr] Processing SOURCES.txt
[pbr] In git context, generating filelist from git
warning: no previously-included files found matching '.gitreview'
warning: no previously-included files matching '*.pyc' found anywhere in distrib
ution
reading manifest template 'MANIFEST.in'
warning: no previously-included files matching '*' found under directory 'doc/bu
```

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 1、2 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

    ......

    def __init__(self, *args, **kwargs):
        super(NetworkAwareness, self).__init__(*args, **kwargs)
        self.switch_info = {}  # dpid: datapath
        self.link_info = {}  # (s1, s2): s1.port
        self.port_link = {}  # s1,port:s1,s2
        self.port_info = {}  # dpid: (ports linked hosts)
        self.topo_map = nx.Graph()
        self.topo_thread = hub.spawn(self._get_topology)
        self.weight = 'delay' # change the weight from hop to delay
        self.lldp_delay = {}  # save the lldp_delay
        self.echo_delay = {}  # save the echo_delay
        self.delay = {}       # save the total delay
        self.switches = None  # the instance of running switches

    ......

    @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
        dpid = dp.id
        # try to get lldp_delay through switches
        try:
            src_dpid, src_port_no = LLDPPacket.lldp_parse(msg.data)
            if self.switches is None:
                self.switches = lookup_service_brick('switches') # look up
running switch instance
            for port in self.switches.ports.keys():
                if src_dpid == port.dpid and src_port_no == port.port_no:
                    self.lldp_delay[(src_dpid, dpid)] =
self.switches.ports[port].delay * 1000
                    # save the lldp_delay to the dictionary
                    # the delay in Python is calc with the unit "second",
in order to change to "ms", need to multiply 1000
        except:
            return
```

代码分析：
（1）import 与变量设置

首先我们需要 from ryu.base.app_manager import lookup_service_brick，然后在__init__()函数中定义字典 self.lldp_delay = {}用来存储 lldp_delay，定义字典 self.echo_delay = {}用来存储 echo_delay，定义字典 self.delay = {}用来存储总时延 delay，最后定义 self.switches = None 用来记录运行的 switches 实例。同时通过 self.weight = 'delay'将 weight 从 hop 改成 delay。

（2）packet_in_handler()函数

根据实验指导书的提示，我们在 packet_in_handler()函数中，如果当前记录运行的 switches 实例为 None，则使用 lookup_service_brick('switches')获取运行的 switches 实例。随后如果源 dpid 与端口 dpid 相符合，且源端口号与端口号相符合，则从 switches 中获取 lldp_delay 并记录到对应的字典中。同时注意，由于 Python 计算时 delay 的单位是秒(s)，为了与 topo 中的延迟设置相对应，需要改成 ms，所以得到的结果应该乘以 1000 再进行记录。

4. 获取 Echo_delay

我们需要为控制器设置发送 Echo_Request 报文的函数与处理交换机发来的 Echo_Reply 报文的函数，同时根据 Echo_Reply 报文的信息得到 echo_delay。

```
# send echo_request to switches
    def send_echo_request(self, switch):
        datapath = switch.dp
        parser = datapath.ofproto_parser
        echo_req = parser.OFPEchoRequest(datapath, data=bytes(("%.12f" %
time.time()).encode())) # need to encode
        datapath.send_msg(echo_req)

    # handle the echo_reply send by switches
    @set_ev_cls(ofp_event.EventOFPEchoReply, [MAIN_DISPATCHER,
CONFIG_DISPATCHER, HANDSHAKE_DISPATCHER])
    def echo_reply_handler(self, ev):
        now_timestamp = time.time() # record the time
        try:
            echo_delay = now_timestamp - eval(ev.msg.data) # calc the
echo_delay
            self.echo_delay[ev.msg.datapath.id] = echo_delay * 1000 # save
the echo_delay to the dictionary, also * 1000
        except:
            return
```

代码分析：

在 send_echo_request()函数中，我们通过 time.time()函数获取当前的时间戳，用特定的格式保存为字符串后转换为 bytes 类型，同时要注意，转换后的字符串要进行 encode()，否则会报错。将时间信息包装成 OFPEchoRequest 类型的报文，进行转发。

在 echo_reply_handler()函数中，首先通过 time.time()函数获取当前的时间，然后尝试用当前的时间减去 OFPEchoReply 类型报文数据内包含的时间，将这个时间差即 echo_delay 保存到对应的字典当中。同时与 lldp_delay 保存时要注意的地方一样，需要乘以 1000 来保持单位一致。

5. 计算总时延 delay 并打印

_get_topology() 函数本身是一个不断运行的线程，所以我们在这里获取时延，来得到不断的时延更新。

```
def _get_topology(self):
        _hosts, _switches, _links = None, None, None
        while True:
            hosts = get_host(self)
            switches = get_switch(self)
            links = get_link(self)
            # update topo_map when topology change
            if [str(x) for x in hosts] == _hosts and [str(x) for x in switches]
== _switches and [str(x) for x in links] == _links:
                continue
            _hosts, _switches, _links = [str(x) for x in hosts], [str(x) for
x in switches], [str(x) for x in links]
            for switch in switches:
                self.port_info.setdefault(switch.dp.id, set())
                # record all ports
                for port in switch.ports:
                    self.port_info[switch.dp.id].add(port.port_no)
                self.send_echo_request(switch)
                hub.sleep(0.5)
            for host in hosts:
                # take one ipv4 address as host id
                if host.ipv4:
                    self.link_info[(host.port.dpid, host.ipv4[0])] =
host.port.port_no
                    self.topo_map.add_edge(host.ipv4[0], host.port.dpid,
hop=1, delay=0, is_host=True)
            for link in links:
                # delete ports linked switches
                self.port_info[link.src.dpid].discard(link.src.port_no)
                self.port_info[link.dst.dpid].discard(link.dst.port_no)
                # s1 -> s2: s1.port, s2 -> s1: s2.port
                self.port_link[(link.src.dpid, link.src.port_no)] =
(link.src.dpid, link.dst.dpid)
```

```
                self.port_link[(link.dst.dpid, link.dst.port_no)] =
(link.dst.dpid, link.src.dpid)
                self.link_info[(link.src.dpid, link.dst.dpid)] =
link.src.port_no
                self.link_info[(link.dst.dpid, link.src.dpid)] =
link.dst.port_no
                # define values to calc the entire delay
                lldp_delay1 = 0
                lldp_delay2 = 0
                echo_delay1 = 0
                echo_delay2 = 0
                if (link.src.dpid, link.dst.dpid) in self.lldp_delay:
                    lldp_delay1 = self.lldp_delay[(link.src.dpid,
link.dst.dpid)]
                if (link.dst.dpid, link.src.dpid) in self.lldp_delay:
                    lldp_delay2 = self.lldp_delay[(link.dst.dpid,
link.src.dpid)]
                if link.src.dpid in self.echo_delay:
                    echo_delay1 = self.echo_delay[(link.src.dpid)]
                if link.dst.dpid in self.echo_delay:
                    echo_delay2 = self.echo_delay[(link.dst.dpid)]
                # calc to whole delay
                delay = (lldp_delay1 + lldp_delay2 -echo_delay1 -
echo_delay2) / 2
                # delay is supposed to be bigger than 0, if less than 0, set
it to 0
                if delay < 0:
                    delay = 0
                # save the whole delay to the dictionary
                self.delay[(link.src.dpid, link.dst.dpid)] = delay
                # add edge to topo map
                self.topo_map.add_edge(link.src.dpid, link.dst.dpid,
hop=1, delay=delay, is_host=False)
            if self.weight == 'delay':
                self.show_topo_map()
            hub.sleep(GET_TOPOLOGY_INTERVAL)

def show_topo_map(self):
        self.logger.info('topo map:')
        self.logger.info('{:^10s}  ->  {:^10s}      {:^10s}'.format('nod
e', 'node', 'delay'))
        # add one item: delay
```

```
        for src, dst in self.topo_map.edges:
            self.logger.info('{:^10s}      {:^10s}      {:^10s}'.format(s
tr(src), str(dst), str('%.2f' % self.topo_map.edges[src,
dst]['delay'])+'ms'))
        # print info with delay, adding unit "ms"
        self.logger.info('\n')
```

代码分析：

在_get_topology()函数中，我们对每个交换机运行之前编写的 send_echo_request(switch)函数，发送 Echo_Request，随后还需要 hub.sleep(0.5)来暂停一下。如果不暂停的话，ehco_delay 会非常大，不符合常理。这里猜测原因是：如果快速给每个交换机发送 Echo_Request 而不暂停，则每个交换机会同时快速发送 Echo_Reply 到控制器，而控制器没办法一下子同时处理大量的 Echo_Reply，需要排队进行处理，对每个 Echo_Reply 执行 echo_reply_handler()函数的延迟不同，导致测量误差非常大。随后定义了测量原理中需要的 lldp_delay1 等 4 个变量，通过对应的源 dpid 与目的 dpid 查找对应的 lldp_delay 字典与 echo_delay 字典，得到对应类型的 delay 值进行计算。同时注意，时延不应为负，测量出负数应取 0，最后把结果存入 delay 字典中。再通过 topo_map.add_edge()函数在对应的拓扑图中加边，存入 delay 信息。通过 show_topo_map()函数打印信息。

在 show_topo_map()函数中，我们修改对应的打印格式，增加了 delay 这一项，同时加上单位 ms，进行打印。

修改后的代码命名为 link_delay.py，在此文件中调用 network_awareness.py 进行控制。

在命令行终端内输入：

```
sudo python topo_1970.py
```

创建网络拓扑，再在另一个命令行终端内输入：

```
ryu-manager link_delay.py --observe-links
```

启动 Ryu 控制器

```
sdn@ubuntu:~/Desktop/mininet/custom$ sudo python topo_1970.py
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
BBN HARVARD MIT RAND SDC SRI UCLA UCSB UTAH
*** Adding switches:
s1 s2 s3 s4 s5 s6 s7 s8 s9
*** Adding links:
```

```
sdn@ubuntu:~/Desktop/mininet/custom$ ryu-manager link_delay.py --ob
serve-links
loading app link_delay.py
loading app ryu.controller.ofp_handler
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
instantiating app None of NetworkAwareness
creating context network_awareness
instantiating app link_delay.py of ShortestForward
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu.topology.switches of Switches
topo map:
   node      ->     node           delay


topo map:
   node      ->     node           delay
```

可以看到，一开始没有节点之间链路时延的信息记录。

在 mininet 命令行内输入 pingall，测试各节点之间的连通性。

```
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
BBN -> X X X X X X X X
HARVARD -> BBN MIT RAND SDC SRI UCLA UCSB UTAH
MIT -> BBN HARVARD RAND SDC SRI UCLA UCSB UTAH
RAND -> BBN HARVARD MIT SDC SRI UCLA UCSB UTAH
SDC -> BBN HARVARD MIT RAND SRI UCLA UCSB UTAH
SRI -> BBN HARVARD MIT RAND SDC UCLA UCSB UTAH
UCLA -> BBN HARVARD MIT RAND SDC SRI UCSB UTAH
UCSB -> BBN HARVARD MIT RAND SDC SRI UCLA UTAH
UTAH -> BBN HARVARD MIT RAND SDC SRI UCLA UCSB
*** Results: 11% dropped (64/72 received)
mininet>
```

可以看到，一开始由于沉默主机现象，BBN 主机到各个主机之间均无法 ping 通，随后的各个主机之间都能相互 ping 通。

```
topo map:
   node      ->     node           delay
    7                6             10.87ms
    7                8             62.93ms
    6                5             18.64ms
    2                3             12.15ms
    2                4             13.66ms
    3                4             15.31ms
    1                9             11.34ms
    9                8             18.25ms
    9                5             30.54ms
    4                5             16.20ms


host not find/no path
topo map:
   node      ->     node           delay
```

在 Ryu 控制器中我们可以看到各主机之间的链路时延，与拓扑中预设的时延相比较。

```
# add edges between switches
self.addLink( s1 , s9, bw=10, delay='10ms')
self.addLink( s2 , s3, bw=10, delay='11ms')
self.addLink( s2 , s4, bw=10, delay='13ms')
self.addLink( s3 , s4, bw=10, delay='14ms')
self.addLink( s4 , s5, bw=10, delay='15ms')
self.addLink( s5 , s9, bw=10, delay='29ms')
self.addLink( s5 , s6, bw=10, delay='17ms')
self.addLink( s6 , s7, bw=10, delay='10ms')
self.addLink( s7 , s8, bw=10, delay='62ms')
self.addLink( s8 , s9, bw=10, delay='17ms')
```

可以发现，得到的测量时延与理论时延有偏差，但仍保持相近。而且也由于沉默主机现象，显示 host not find/no path。

在 mininet 命令行内再次输入 pingall，测试各节点之间的连通性。

```
*** Results: 11% dropped (64/72 received)
mininet> pingall
*** Ping: testing ping reachability
BBN -> HARVARD MIT RAND SDC SRI UCLA UCSB UTAH
HARVARD -> BBN MIT RAND SDC SRI UCLA UCSB UTAH
MIT -> BBN HARVARD RAND SDC SRI UCLA UCSB UTAH
RAND -> BBN HARVARD MIT SDC SRI UCLA UCSB UTAH
SDC -> BBN HARVARD MIT RAND SRI UCLA UCSB UTAH
SRI -> BBN HARVARD MIT RAND SDC UCLA UCSB UTAH
UCLA -> BBN HARVARD MIT RAND SDC SRI UCSB UTAH
UCSB -> BBN HARVARD MIT RAND SDC SRI UCLA UTAH
UTAH -> BBN HARVARD MIT RAND SDC SRI UCLA UCSB
*** Results: 0% dropped (72/72 received)
mininet> 
```

可以看到，这次各个主机之间全部可以互相 ping 通。

4. 选做题：容忍链路故障

1970 年的网络硬件发展尚不成熟，通信链路和交换机端口发生故障的概率较高。请设计 Ryu app，在任务一的基础上实现容忍链路故障的路由选择：每当链路出现故障时，重新选择当前可用路径中时延最低的路径；当链路故障恢复后，也重新选择新的时延最低的路径。

模拟链路故障：

在 mininet 中可以用 link down 和 link up 来模拟链路故障和故障恢复。

控制器捕捉链路故障：

链路状态改变时，链路关联的端口状态也会变化，从而产生端口状态改变的事件，即 EventOFPPortStatus，通过将此事件与你设计的处理函数绑定在一起，就可以获取状态改变的信息，执行相应的处理。

OFPFC_DELETE 消息：

与向交换机中增加流表的 OFPFC_ADD 命令不同，OFPFC_DELETE 消息用于删除交换机中符合匹配项的所有流表。

由于添加和删除都属于 OFPFlowMod 消息，因此只需稍微修改 add_flow()函数，即可生成 delete_flow()函数。

Packet_In 消息的合理利用：

基本思路是在链路发生改变时，删除受影响的链路上所有交换机上的相关流表的信息，下一次交换机将匹配默认流表项，向控制器发送 packet_in 消息，控制器重新计算并下发最小时延路径。

我们根据实验指导书的提示，在 network_awareness.py 里定义 delete_flow()函数和 port_status_handler()函数。

```
# Task 2: delete flow
    def delete_flow(self, datapath, match):
        ofp = datapath.ofproto
        parser = datapath.ofproto_parser
        inst = []
        del_mod = parser.OFPFlowMod(datapath, 0, 0, 0, ofp.OFPFC_DELETE, 0,
0, 0, ofp.OFP_NO_BUFFER, ofp.OFPP_ANY, ofp.OFPG_ANY,
ofp.OFPFF_SEND_FLOW_REM, match, inst)
        datapath.send_msg(del_mod)

# Task 2: change port status  when links down or up
    @set_ev_cls(ofp_event.EventOFPPortStatus, MAIN_DISPATCHER)
    def port_status_handler(self, ev):
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        if msg.reason in [ofproto.OFPPR_ADD, ofproto.OFPPR_MODIFY]:
            datapath.ports[msg.desc.port_no] = msg.desc
            self.topo_map.clear()
            for dpid in self.port_info.keys():
                for port in self.port_info[dpid]:
                    match = parser.OFPMatch(in_port=port)
                    self.delete_flow(self.switch_info[dpid], match)
        elif msg.reason == ofproto.OFPPR_DELETE:
            datapath.ports.pop(msg.desc.port_no, None)
        else:
            return
        self.send_event_to_observers(ofp_event.EventOFPPortStateChange(
            datapath, msg.reason, msg.desc.port_no), datapath.state)
```

代码分析：

在 delete_flow()函数中，我们根据 add_flow()函数的格式定义了 delete_flow()函数，数据包里的指令 inst 定义为空。

在 port_status_handler()函数中，我们根据实验指导书的提示，对端口的 topo_map 进行清除。然后对使用端口进行连接的主机所连接的交换机，对这些交换机发送删除流表的指令。

保存修改后的 network_awareness.py，在命令行终端内输入：

```
sudo python topo_1970.py
```

创建网络拓扑，再在另一个命令行终端内输入：

```
ryu-manager link_delay.py --observe-links
```

启动 Ryu 控制器

首先先在 mininet 命令行内输入 pingall，解除沉默主机现象。

第二次 pingall 的时候，各节点之间都能相互 ping 通。

```
*** Results: 9% dropped (65/72 received)
mininet> pingall
*** Ping: testing ping reachability
BBN -> HARVARD MIT RAND SDC SRI UCLA UCSB UTAH
HARVARD -> BBN MIT RAND SDC SRI UCLA UCSB UTAH
MIT -> BBN HARVARD RAND SDC SRI UCLA UCSB UTAH
RAND -> BBN HARVARD MIT SDC SRI UCLA UCSB UTAH
SDC -> BBN HARVARD MIT RAND SRI UCLA UCSB UTAH
SRI -> BBN HARVARD MIT RAND SDC UCLA UCSB UTAH
UCLA -> BBN HARVARD MIT RAND SDC SRI UCSB UTAH
UCSB -> BBN HARVARD MIT RAND SDC SRI UCLA UTAH
UTAH -> BBN HARVARD MIT RAND SDC SRI UCLA UCSB
*** Results: 0% dropped (72/72 received)
```

在 mininet 命令行内输入 dump，查看各节点的详细网络信息。

```
mininet> dump
<CPULimitedHost BBN: BBN-eth0:10.0.0.1 pid=202054>
<CPULimitedHost HARVARD: HARVARD-eth0:10.0.0.2 pid=202061>
<CPULimitedHost MIT: MIT-eth0:10.0.0.3 pid=202065>
<CPULimitedHost RAND: RAND-eth0:10.0.0.4 pid=202069>
<CPULimitedHost SDC: SDC-eth0:10.0.0.5 pid=202073>
<CPULimitedHost SRI: SRI-eth0:10.0.0.6 pid=202077>
<CPULimitedHost UCLA: UCLA-eth0:10.0.0.7 pid=202081>
<CPULimitedHost UCSB: UCSB-eth0:10.0.0.8 pid=202085>
<CPULimitedHost UTAH: UTAH-eth0:10.0.0.9 pid=202089>
```

可以看到，SDC 的 IP 地址为 10.0.0.5，MIT 的 IP 地址为 10.0.0.3。

在 mininet 命令行内输入 SDC ping MIT -c5，让 SDC 主机 ping MIT 主机 5 次。

```
mininet> SDC ping MIT -c5
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=131 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=129 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=129 ms
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=128 ms
64 bytes from 10.0.0.3: icmp_seq=5 ttl=64 time=128 ms

--- 10.0.0.3 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4004ms
rtt min/avg/max/mdev = 127.927/129.087/130.755/0.953 ms
```

可以看到，此时两个主机之间的链路时延 RTT 约为 128ms。

```
path: 10.0.0.3 -> 10.0.0.5
10.0.0.3 -> 1:s8:3 -> 4:s9:3 -> 3:s5:4 -> 2:s6:1 -> 10.0.0.5
path: 10.0.0.3 -> 10.0.0.6
```

在 Ryu 控制器打印的信息中，我们可以看到，SDC 与 MIT 之间的链路经过了 s8 与 s9 两个交换机，此路径为最短时延路径。

在 mininet 命令行内输入 link s8 s9 down，断开交换机 s8 与 s9 之间的连接。再输入 SDC ping MIT -c5，查看此时 SDC 与 MIT 主机之间的链路时延。

```
mininet> link s8 s9 down
mininet> SDC ping MIT -c5
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=77.9 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=148 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=146 ms
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=147 ms
64 bytes from 10.0.0.3: icmp_seq=5 ttl=64 time=147 ms
```

可以看到，断开交换机 s8 与 s9 之间的连接后，SDC 与 MIT 主机之间的链路时延 RTT 约为 147ms。

```
path: 10.0.0.5 -> 10.0.0.3
10.0.0.5 -> 1:s6:3 -> 2:s7:3 -> 2:s8:1 -> 10.0.0.3
topo map:
   node       ->      node             delay
```

在 Ryu 控制器打印的信息中，我们可以看到，交换机 s8 与 s9 之间的连接断开后，SDC 与 MIT 之间的链路不再经过 s8 与 s9 之间的连接。同时由于当前路径不再是最短时延路径，两个主机之间的链路时延 RTT 也增加了。

在 mininet 命令行内输入 link s8 s9 up，恢复交换机 s8 与 s9 之间的连接。再输入 SDC ping MIT -c5，查看此时 SDC 与 MIT 主机之间的链路时延。

```
mininet> link s8 s9 up
mininet> SDC ping MIT -c5
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=212 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=130 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=129 ms
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=129 ms
64 bytes from 10.0.0.3: icmp_seq=5 ttl=64 time=128 ms

--- 10.0.0.3 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4007ms
rtt min/avg/max/mdev = 128.037/145.724/212.174/33.229 ms
mininet>
```

可以看到，恢复交换机 s8 与 s9 之间的连接后，SDC 与 MIT 主机之间的链路时延 RTT 约为 129ms，稍微大于断开连接操作之前的链路时延，但相差不大。

```
path: 10.0.0.5 -> 10.0.0.3
10.0.0.5 -> 1:s6:2 -> 4:s5:3 -> 3:s9:4 -> 3:s8:1 -> 10.0.0.3
topo map:
   node      ->      node            delay
```

可以看到，由于交换机 s8 与 s9 之间的连接恢复，当前路径又变换回使用 s8 与 s9 之间连接的路径，即恢复使用最短时延路径。

实验结果：

1. 学习了最小跳数路径的代码，熟悉了拓扑感知的原理与沉默主机现象。
2. 学习了计算最小跳数路径，了解了如何在 Ryu 控制器界面内打印提示信息，学习了阅读主机之间的路径。
3. 学习了计算最短时延路径的原理，熟悉了 Ryu 控制器内 switches.py 源码的数据结构与内容，根据原理自己编写函数，修改代码计算链路时延。
4. 学习了链路故障容忍的原理，自己编写函数修改代码实现主机链路的重新选择。

实验中遇到的问题：

在命令行终端内输入：启动

```
ryu-manager link_delay.py --observe-links
```

启动 Ryu 控制器后，偶尔会遇到以下情况

```
sdn@ubuntu:~/Desktop/mininet/custom$ ryu-manager link_delay.py --ob
serve-links
loading app link_delay.py
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
instantiating app None of NetworkAwareness
creating context network_awareness
instantiating app link_delay.py of ShortestForward
instantiating app ryu.topology.switches of Switches
instantiating app ryu.controller.ofp_handler of OFPHandler
topo map:
   node     ->     node           delay


hub: uncaught exception: Traceback (most recent call last):
  File "/usr/local/lib/python3.8/dist-packages/ryu/lib/hub.py", lin
e 60, in _launch
    return func(*args, **kwargs)
  File "/home/sdn/Desktop/mininet/custom/network_awareness.py", lin
e 158, in _get_topology
    self.port_info[link.dst.dpid].discard(link.dst.port_no)
KeyError: 7
```

Ryu 控制器显示 KeyError: 7。

```
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
BBN -> X X X X X X X X
HARVARD -> X ^C
Interrupt
stopping HARVARD
mininet>
```

随后在 mininet 命令行内输入 pingall，所有节点之间都无法 ping 通。

```
KeyError: 7

host not find/no path
host not find/no path
host not find/no path
host not find/no path
host not find/no path
host not find/no path
host not find/no path
host not find/no path
host not find/no path
host not find/no path
```

同时 Ryu 控制器内不会打印主机之间链路连接的信息，只会显示 host not find/no path。

此错误发生没有明显的规律，多次尝试构建拓扑与启动 Ryu 控制器，此错误有时会发生，有时不会，

查询资料得知，Python 中的"KeyError"是一个常见的错误类型，它通常表示在字典或者集合中查找一个不存在的键。这个错误可以发生在很多场合，例如：

在使用字典时，通过一个不存在的键来查找值；在使用字典时，试图添加一个不存在的键值对；在使用集合时，试图移除一个不存在的元素。

相关资料如下：https://pythonjishu.com/python-error-49/

源代码：

network_awareness.py

```
from ryu.base import app_manager
from ryu.base.app_manager import lookup_service_brick
from ryu.ofproto import ofproto_v1_3
from ryu.controller.handler import set_ev_cls
from ryu.controller.handler import MAIN_DISPATCHER, CONFIG_DISPATCHER,
DEAD_DISPATCHER, HANDSHAKE_DISPATCHER
from ryu.controller import ofp_event
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet, arp
from ryu.lib import hub
from ryu.topology import event
from ryu.topology.api import get_host, get_link, get_switch
from ryu.topology.switches import LLDPPacket
import networkx as nx
import copy
import time

GET_TOPOLOGY_INTERVAL = 2
SEND_ECHO_REQUEST_INTERVAL = .05
GET_DELAY_INTERVAL = 2


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

    def __init__(self, *args, **kwargs):
        super(NetworkAwareness, self).__init__(*args, **kwargs)
        self.switch_info = {}  # dpid: datapath
        self.link_info = {}  # (s1, s2): s1.port
        self.port_link = {}  # s1,port:s1,s2
        self.port_info = {}  # dpid: (ports linked hosts)
        self.topo_map = nx.Graph()
        self.topo_thread = hub.spawn(self._get_topology)
        self.weight = 'delay' # change the weight from hop to delay
        self.lldp_delay = {}  # save the lldp_delay
        self.echo_delay = {}  # save the echo_delay
```

```
        self.delay = {}       # save the total delay
        self.switches = None  # the instance of running switches

    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)

    # Task 2: delete flow
    def delete_flow(self, datapath, match):
        ofp = datapath.ofproto
        parser = datapath.ofproto_parser
        inst = []
        del_mod = parser.OFPFlowMod(datapath, 0, 0, 0, ofp.OFPFC_DELETE, 0,
0, 0, ofp.OFP_NO_BUFFER,
                                    ofp.OFPP_ANY, ofp.OFPG_ANY,
ofp.OFPFF_SEND_FLOW_REM, match, inst)
        datapath.send_msg(del_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)

    # Task 2: change port status  when links down or up
    @set_ev_cls(ofp_event.EventOFPPortStatus, MAIN_DISPATCHER)
    def port_status_handler(self, ev):
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
```

```
        if msg.reason in [ofproto.OFPPR_ADD, ofproto.OFPPR_MODIFY]:
            datapath.ports[msg.desc.port_no] = msg.desc
            self.topo_map.clear()
            for dpid in self.port_info.keys():
                for port in self.port_info[dpid]:
                    match = parser.OFPMatch(in_port=port)
                    self.delete_flow(self.switch_info[dpid], match)
        elif msg.reason == ofproto.OFPPR_DELETE:
            datapath.ports.pop(msg.desc.port_no, None)
        else:
            return
        self.send_event_to_observers(ofp_event.EventOFPPortStateChange(
            datapath, msg.reason, msg.desc.port_no), datapath.state)

    @set_ev_cls(ofp_event.EventOFPStateChange, [MAIN_DISPATCHER,
DEAD_DISPATCHER])
    def state_change_handler(self, ev):
        dp = ev.datapath
        dpid = dp.id
        if ev.state == MAIN_DISPATCHER:
            self.switch_info[dpid] = dp
        if ev.state == DEAD_DISPATCHER:
            del self.switch_info[dpid]

    @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
        dpid = dp.id
        # try to get lldp_delay through switches
        try:
            src_dpid, src_port_no = LLDPPacket.lldp_parse(msg.data)
            if self.switches is None:
                self.switches = lookup_service_brick('switches') # look up
running switch instance
            for port in self.switches.ports.keys():
                if src_dpid == port.dpid and src_port_no == port.port_no:
                    self.lldp_delay[(src_dpid, dpid)] =
self.switches.ports[port].delay * 1000
                    # save the lldp_delay to the dictionary
```

```
                    # the delay in Python is calc with the unit "second",
in order to change to "ms", need to multiply 1000
        except:
            return

    # send echo_request to switches
    def send_echo_request(self, switch):
        datapath = switch.dp
        parser = datapath.ofproto_parser
        echo_req = parser.OFPEchoRequest(datapath, data=bytes(("%.12f" %
time.time()).encode())) # need to encode
        datapath.send_msg(echo_req)

    # handle the echo_reply send by switches
    @set_ev_cls(ofp_event.EventOFPEchoReply, [MAIN_DISPATCHER,
CONFIG_DISPATCHER, HANDSHAKE_DISPATCHER])
    def echo_reply_handler(self, ev):
        now_timestamp = time.time() # record the time
        try:
            echo_delay = now_timestamp - eval(ev.msg.data) # calc the
echo_delay
            self.echo_delay[ev.msg.datapath.id] = echo_delay * 1000 # save
the echo_delay to the dictionary, also * 1000
        except:
            return

    def _get_topology(self):
        _hosts, _switches, _links = None, None, None
        while True:
            hosts = get_host(self)
            switches = get_switch(self)
            links = get_link(self)
            # update topo_map when topology change
            if [str(x) for x in hosts] == _hosts and [str(x) for x in switches]
== _switches and [str(x) for x in links] == _links:
                continue
            _hosts, _switches, _links = [str(x) for x in hosts], [str(x) for
x in switches], [str(x) for x in links]
            for switch in switches:
                self.port_info.setdefault(switch.dp.id, set())
                # record all ports
                for port in switch.ports:
```

```
                    self.port_info[switch.dp.id].add(port.port_no)
                self.send_echo_request(switch)
                hub.sleep(0.5)
            for host in hosts:
                # take one ipv4 address as host id
                if host.ipv4:
                    self.link_info[(host.port.dpid, host.ipv4[0])] =
host.port.port_no
                    self.topo_map.add_edge(host.ipv4[0], host.port.dpid,
hop=1, delay=0, is_host=True)
            for link in links:
                # delete ports linked switches
                self.port_info[link.src.dpid].discard(link.src.port_no)
                self.port_info[link.dst.dpid].discard(link.dst.port_no)
                # s1 -> s2: s1.port, s2 -> s1: s2.port
                self.port_link[(link.src.dpid, link.src.port_no)] =
(link.src.dpid, link.dst.dpid)
                self.port_link[(link.dst.dpid, link.dst.port_no)] =
(link.dst.dpid, link.src.dpid)
                self.link_info[(link.src.dpid, link.dst.dpid)] =
link.src.port_no
                self.link_info[(link.dst.dpid, link.src.dpid)] =
link.dst.port_no
                # define values to calc the entire delay
                lldp_delay1 = 0
                lldp_delay2 = 0
                echo_delay1 = 0
                echo_delay2 = 0
                if (link.src.dpid, link.dst.dpid) in self.lldp_delay:
                    lldp_delay1 = self.lldp_delay[(link.src.dpid,
link.dst.dpid)]
                if (link.dst.dpid, link.src.dpid) in self.lldp_delay:
                    lldp_delay2 = self.lldp_delay[(link.dst.dpid,
link.src.dpid)]
                if link.src.dpid in self.echo_delay:
                    echo_delay1 = self.echo_delay[(link.src.dpid)]
                if link.dst.dpid in self.echo_delay:
                    echo_delay2 = self.echo_delay[(link.dst.dpid)]
                # calc to whole delay
                delay = (lldp_delay1 + lldp_delay2 -echo_delay1 -
echo_delay2) / 2
```

```
                # delay is supposed to be bigger than 0, if less than 0, set
it to 0
                if delay < 0:
                    delay = 0
                # save the whole delay to the dictionary
                self.delay[(link.src.dpid, link.dst.dpid)] = delay
                # add edge to topo map
                self.topo_map.add_edge(link.src.dpid, link.dst.dpid,
hop=1, delay=delay, is_host=False)
            if self.weight == 'delay':
                self.show_topo_map()
            hub.sleep(GET_TOPOLOGY_INTERVAL)

    def shortest_path(self, src, dst, weight='hop'):
        try:
            paths = list(nx.shortest_simple_paths(
                self.topo_map, src, dst, weight=weight))
            return paths[0]
        except:
            self.logger.info('host not find/no path')

    def show_topo_map(self):
        self.logger.info('topo map:')
        self.logger.info('{:^10s}  ->  {:^10s}      {:^10s}'.format('nod
e', 'node', 'delay'))
        # add one item: delay
        for src, dst in self.topo_map.edges:
            self.logger.info('{:^10s}      {:^10s}      {:^10s}'.format(s
tr(src), str(dst), str('%.2f' % self.topo_map.edges[src,
dst]['delay'])+'ms'))
        # print info with delay, adding unit "ms"
        self.logger.info('\n')
```

link_delay.py

```
# ryu-manager link_delay.py --observe-links
from ryu.base import app_manager
from ryu.base.app_manager import lookup_service_brick
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER,
DEAD_DISPATCHER, HANDSHAKE_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.controller.handler import set_ev_cls
```

```python
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet, arp, ipv4, ether_types
from ryu.controller import ofp_event
from ryu.topology import event
import sys
import time
from network_awareness import NetworkAwareness
import networkx as nx
from ryu.topology.switches import LLDPPacket

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


class ShortestForward(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {
        'network_awareness': NetworkAwareness
    }

    def __init__(self, *args, **kwargs):
        super(ShortestForward, self).__init__(*args, **kwargs)
        self.network_awareness = kwargs['network_awareness']
        self.weight = 'delay' # change the weight from hop to delay
        self.mac_to_port = {} # learning switch
        self.sw = {}          # avoid broadcast loop
        self.path = None

    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)
```

```python
        dp.send_msg(mod)

    @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
        dpid = dp.id
        in_port = msg.match['in_port']
        pkt = packet.Packet(msg.data)
        eth_pkt = pkt.get_protocol(ethernet.ethernet)
        arp_pkt = pkt.get_protocol(arp.arp)
        ipv4_pkt = pkt.get_protocol(ipv4.ipv4)
        pkt_type = eth_pkt.ethertype
        # layer 2 self-learning
        dst_mac = eth_pkt.dst
        src_mac = eth_pkt.src
        if isinstance(arp_pkt, arp.arp):
            self.handle_arp(msg, in_port, dst_mac, src_mac, pkt, pkt_type)
        if isinstance(ipv4_pkt, ipv4.ipv4):
            self.handle_ipv4(msg, ipv4_pkt.src, ipv4_pkt.dst, pkt_type)

    # Task 0: deal with broadcast loop, using code in Lab 2: Broadcast_Loop.py
    def handle_arp(self, msg, in_port, dst, src, pkt, pkt_type):
        msg = msg
        dp = msg.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser
        dpid = dp.id
        self.mac_to_port.setdefault(dpid, {})
        in_port = in_port
        pkt = pkt
        eth_pkt = pkt.get_protocol(ethernet.ethernet)
        if pkt_type == ether_types.ETH_TYPE_LLDP:
            return
        if pkt_type == ether_types.ETH_TYPE_IPV6:
            return
        dst = dst
        src = src
        # just handle loop here
        # just like your code in exp1 mission2
```

```
        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.sw:
                # The same info comes from another port, Just Drop it
                if self.sw[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.sw[(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.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)

    def handle_ipv4(self, msg, src_ip, dst_ip, pkt_type):
        parser = msg.datapath.ofproto_parser
        dpid_path = self.network_awareness.shortest_path(src_ip, dst_ip,
weight=self.weight)
        if not dpid_path:
            return
        self.path = dpid_path
        # get port path:  h1 -> in_port, s1, out_port -> h2
        port_path = []
        for i in range(1, len(dpid_path) - 1):
            in_port = self.network_awareness.link_info[(dpid_path[i],
dpid_path[i - 1])]
            out_port = self.network_awareness.link_info[(dpid_path[i],
dpid_path[i + 1])]
            port_path.append((in_port, dpid_path[i], out_port))
        self.show_path(src_ip, dst_ip, port_path)
        # calc path delay
        # send flow mod
        for node in port_path:
            in_port, dpid, out_port = node
            self.send_flow_mod(parser, dpid, pkt_type, src_ip, dst_ip,
in_port, out_port)
            self.send_flow_mod(parser, dpid, pkt_type, dst_ip, src_ip,
out_port, in_port)
        # send packet_out
        in_port, dpid, out_port = port_path[-1]
        dp = self.network_awareness.switch_info[dpid]
        actions = [parser.OFPActionOutput(out_port)]
        out = parser.OFPPacketOut(datapath=dp, buffer_id=msg.buffer_id,
in_port=in_port, actions=actions, data=msg.data)
        dp.send_msg(out)

    def send_flow_mod(self, parser, dpid, pkt_type, src_ip, dst_ip, in_port,
out_port):
        dp = self.network_awareness.switch_info[dpid]
        match = parser.OFPMatch(in_port=in_port, eth_type=pkt_type,
ipv4_src=src_ip, ipv4_dst=dst_ip)
        actions = [parser.OFPActionOutput(out_port)]
```

```
        self.add_flow(dp, 1, match, actions, 10, 30)

    def show_path(self, src, dst, port_path):
        self.logger.info('path: {} -> {}'.format(src, dst))
        path = src + ' -> '
        for node in port_path:
            path += '{}:s{}:{}'.format(*node) + ' -> '
        path += dst
        self.logger.info(path)
```