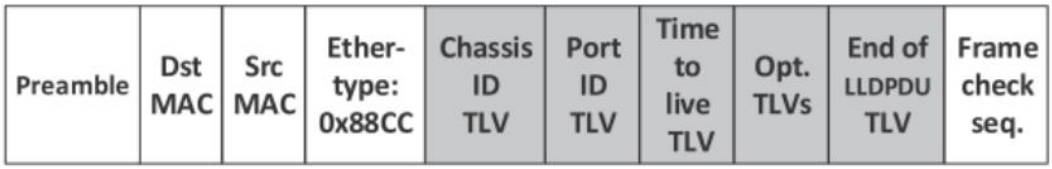
Lab3 测量链路时延与链路容忍

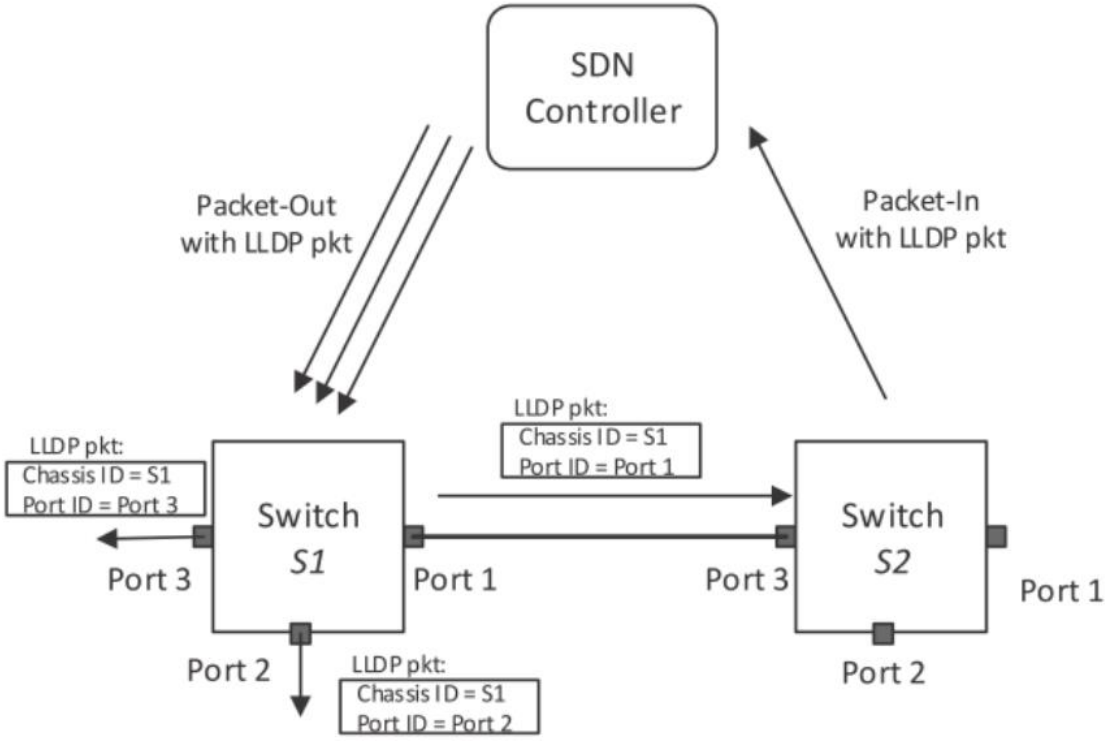
（一） 最小时延路径

（1） 原理

链路发现原理：LLDP(Link Layer Discover Protocol) 即链路层发现协议， Ryu 主要利用 LLDP 发现网络拓扑。LLDP 被封装在以太网帧中，结构如下图。其中深灰色的即为 LLDP 负载， Chassis ID TLV , Port ID,TLV 和 Time to live TLV 是三个强制字段，分别代表交换机标识符（在局域网中是独一无二的），端口号和 TTL 。



接下来介绍 Ryu 如何利用 LLDP 发现链路，假设有两个 OpenFlow 交换机连接在控制器上，如下图：



1. SDN 控制器构造 PacketOut 消息向 S1 的三个端口分别发送 LLDP 数据包，其中将 Chassis ID,TLV 和 Port ID TLV 分别置为 S1 的 dpid 和端口号；

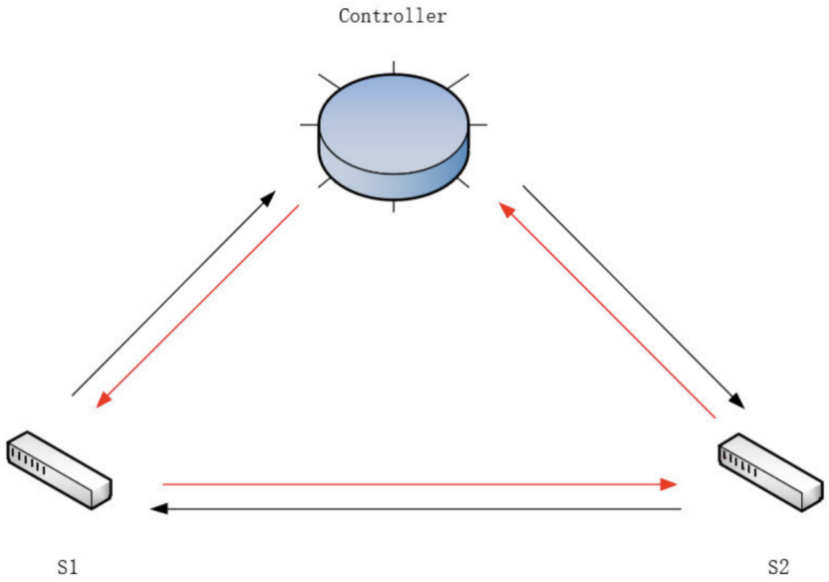
2. 控制器向交换机 S1 中下发流表，流表规则为：将从 Controller 端口收到的 LLDP 数据包从他的对应端口发送出去；

3. 控制器向交换机 S2 中下发流表，流表规则为：将从非 Controller 接收到的 LLDP 数据包发送给控制器；

4. 控制器通过解析 LLDP 数据包，得到链路的源交换机，源接口，通过收到的 PacketIn 消息知道目的交换机和目的接口。

测量原理：链路时延

测量链路时延的思路可参考下图



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

（2）对 Ryu 做如下修改：

```
    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
```

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 lldp_delay self.timestamp 为 LLDP 包在发送时被打上的时间戳。

```
    @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
```

lldp_packet_in_handler()函数负责处理收到的 LLDP 包，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 报文经过一跳后转给控制器，因此可将 lldp_delay 存入发送 LLDP 包对应的交换机端口。完成上述修改后需重新编译安装 Ryu，在安装目录下运行命令 sudo python setup.py install。

（3）获取 lldp_delay：

```
    # in network_awareness.py
    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)
        # add
        self.delay_thread = hub.spawn(self._get_delay)
        self.echo_delay = {}
        self.lldp_delay = {}
        self.switches = None
            self.weight = 'delay'
```

在 NetworkAwareness 类初始化时，使用 hub.spawn()函数在控制器上创建新的协程（coroutine）并周期性发送 Echo Request 与计算链路时延。

```
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handle(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:
                    self.lldp_delay[(src_dpid, dpid)] =
self.switches.ports[port].delay
        except:
                return
```

上面的代码尝试解析 LLDP 包以获取源交换机 ID 及端口号，调用 lookup_service_brick()函数获取正在运行的 switches 的实例，查找其中是否存在源交换机，若有则以(src_dpid, dst_dpid)作为 key，在 self.lldp_delay 中存入时延。

（4）获取 echo_delay：

```
    # in network_awareness.py
    @set_ev_cls(ofp_event.EventOFPEchoReply, MAIN_DISPATCHER)
    def echo_reply_handler(self, ev):
        try:
            echo_delay = time.time() - eval(ev.msg.data)
            self.echo_delay[ev.msg.datapath.id] = echo_delay
        except:
                return
```

控制器在收到 Echo Reply 后，用接收时间戳减去发送时间戳（存储在 echo 包中）获取控制器到交换机的往返时延，并以交换机 ID 作为 key，在 self.echo_delay 中存入时延。

（5）发送 Echo Request 及链路时延的计算：

```
    # in network_awareness.py
    def _get_delay(self):
        while True:
            for dp in self.switch_info.values():
                parser = dp.ofproto_parser
                echo_request = parser.OFPEchoRequest(dp,
data='{:.10f}'.format(time.time()).encode('utf-8'))
                dp.send_msg(echo_request)
```

```
                hub.sleep(SEND_ECHO_REQUEST_INTERVAL)

            for edge in self.topo_map.edges:
                src, dst = edge
                if not self.topo_map[src][dst]['is_host']:
                    try:
                        lldp_delay_s12 = self.lldp_delay[(src, dst)]
                        lldp_delay_s21 = self.lldp_delay[(dst, src)]
                        echo_delay_s1 = self.echo_delay[src]
                        echo_delay_s2 = self.echo_delay[dst]
                        delay = (lldp_delay_s12 + lldp_delay_s21 -
echo_delay_s1 - echo_delay_s2) / 2.0
                        self.topo_map[src][dst]['delay'] = delay if
delay > 0 else 0
                    except:
                        continue
                hub.sleep(GET_DELAY_INTERVAL)
```

该代码通过交换 Echo Request/Reply 报文和 LLDP 报文来计算网络中交换机之间的时延，从 self.switch_info.values() 获取所有交换机的 datapath 对象 dp，使用 OpenFlow 协议解析器 parser 构造 Echo Request 消息。消息的数据部分包含当前时间戳（精确到小数点后 10 位），再调用 dp.send_msg(echo_request) 发送 Echo Request 消息。调用 hub.sleep(SEND_ECHO_REQUEST_INTERVAL) 等待指定的时间间隔 SEND_ECHO_REQUEST_INTERVAL 后再继续发送下一个 Echo Request，这样就发送 Echo Request 消息，并记录发送时间。

从 self.topo_map.edges 获取网络拓扑中的所有链路 edge。检查链路是否是主机连接（is_host），如果是则跳过。

lldp_delay_s12：从 src 到 dst 的 LLDP 报文往返时延，

lldp_delay_s21：从 dst 到 src 的 LLDP 报文往返时延。

echo_delay_s1：从控制器到 src 交换机的 Echo 报文往返时延。

echo_delay_s2：从控制器到 dst 交换机的 Echo 报文往返时延。

利用公式 (lldp_delay_s12 + lldp_delay_s21 - echo_delay_s1 - echo_delay_s2) / 2.0 计算 src 到 dst 的链路时延。如果计算结果为负，则设为 0。将计算出的时延存储在 self.topo_map[src][dst]['delay'] 中。如果获取时延数据时发生异常，则跳过当前链路。

调用 hub.sleep(GET_DELAY_INTERVAL) 等待指定的时间间隔 GET_DELAY_INTERVAL 后再继续下一次时延计算。

（6）获取拓扑（修改）

```
    # in network_awareness.py _get_topology
    # 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:
                hub.sleep(GET_TOPOLOGY_INTERVAL)
                continue
                _hosts, _switches, _links = [str(x) for x in hosts],
[str(x) for x in switches], [str(x) for x in links]
```

当拓扑保持不变时，函数应休眠一段时间，否则直接使用 continue 进入下一轮循环时，又将发出大量的 LLDP 包，这可能导致网络拥塞而造成交换机转发时延变长

（7）处理 ARP 环路广播：

```
    def handle_arp(self, msg, in_port, dst, src, pkt, pkt_type):
        #just handle loop here
        #just like your code in exp1 mission2
        dp = msg.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser
        dpid = dp.id
        self.mac_to_port.setdefault(dpid, {})
        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:
            arp_dst_ip = header_list[ARP].dst_ip
            if (dpid, src, arp_dst_ip) in self.sw:
                if self.sw[(dpid, src, arp_dst_ip)] != in_port:
                    out = parser.OFPPacketOut(datapath=dp,
buffer_id=msg.buffer_id,
                              in_port=in_port, actions=[], data=None)
                    dp.send_msg(out)
                    return
            else:
                self.sw[(dpid, src, arp_dst_ip)] = in_port

        # self-learning
        self.mac_to_port[dpid][src] = in_port

        if dst in self.mac_to_port[dpid]:
            out_port = self.mac_to_port[dpid][dst]
        else:
            out_port = ofp.OFPP_FLOOD
```

```
        actions = [parser.OFPActionOutput(out_port)]

        if out_port != ofp.OFPP_FLOOD:
            match = parser.OFPMatch(in_port=in_port, eth_dst=dst,
eth_type=pkt_type)
            self.add_flow(dp, 1, match, actions, hard_timeout=5)

        data = None
        if msg.buffer_id == ofp.OFP_NO_BUFFER:
            data = msg.data

        out = parser.OFPPacketOut(datapath=dp, buffer_id=msg.buffer_id,
                                  in_port=in_port, actions=actions, data=data)
            dp.send_msg(out)
```

（8）测试两个交换机之间的时延：

```
    def show_topo_map2(self):
        self.logger.info('topo map:')
        self.logger.info('{:^10s}  ->  {:^10s}    {:^10s}'.format('node'
, 'node', 'delay'))
        for src, dst in self.topo_map.edges:
            delay = int(self.topo_map[src][dst]['delay']*1000)
            self.logger.info('{:^10s}      {:^10s}    {:^10s}'.format(st
r(src), str(dst), str(delay)+'ms'))
            self.logger.info('\n')
```

```
topo map:
   node      ->      node         delay
    1                 9           10ms
    2                 3           11ms
    2                 4           14ms
    3                 4           14ms
    4                 5           16ms
    5                 9           29ms
    5                 6           17ms
    6                 7           10ms
    7                 8           62ms
    8                 9           18ms
```

(8)运行结果：

```
mininet> SDC ping MIT
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=70.6 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=128 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=127 ms
64 bytes from 10.0.0.3: icmp_seq=6 ttl=64 time=126 ms
64 bytes from 10.0.0.3: icmp_seq=7 ttl=64 time=127 ms
64 bytes from 10.0.0.3: icmp_seq=8 ttl=64 time=127 ms
64 bytes from 10.0.0.3: icmp_seq=9 ttl=64 time=128 ms
64 bytes from 10.0.0.3: icmp_seq=10 ttl=64 time=127 ms
64 bytes from 10.0.0.3: icmp_seq=11 ttl=64 time=128 ms
64 bytes from 10.0.0.3: icmp_seq=12 ttl=64 time=127 ms
64 bytes from 10.0.0.3: icmp_seq=13 ttl=64 time=127 ms
^C
--- 10.0.0.3 ping statistics ---
13 packets transmitted, 12 received, 7% packet loss, time 12035ms
rtt min/avg/max/mdev = 70.644/123.113/128.924/15.831 ms
```

```
test@sdnexp:~/Desktop/sdn/ryu/ryu/app$ ryu-manager shortest_forward.py
 --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
host not find/no path
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
```

从上图中可以发现，SDC ping MIT 时，选择了最小时延路径进行转发，与返时延与最小时延理论值（126ms）基本相同

此外，第一次 ping 的时延明显小于后面，这是因为 SDC 将 ICMP 包发给连接的交换机时，匹配默认流表后转发给控制器，控制器根据 ICMP 包的源 IP 地址与目的 IP 地址，计算出最小时延路径，并将流表项下发给路径上的各个交换机。之后，控制器直接将 ICMP 包发送给连接 MIT 的交换机。MIT 在发送 ICMP 响应包时则通过匹配刚才下发的流表项，通过最小时延路径上转发给 SDC。因此，第一次 ping 的时延略大于往返时延的一半

（二）　容忍链路故障

（1）处理链路故障原理：

当链路状态改变时，链路关联的端口状态也会改变，从而产生端口状态改变的事件，即EventOFPPortStatus，将该事件与处理函数绑定在一起，就可以获取状态改变的信息，并执行相应的处理。

当链路状态改变时，控制器删除网络拓扑中所有交换机上除默认流表以外的流表项，下一次交换机收到数据包后将会匹配默认流表项，向控制器发送 packet_in 消息，控制器重新计算最小时延路径并下发流表。

（2）　删除流表：

```
    def del_flow(self, datapath, match):
        ofp = datapath.ofproto
        parser = datapath.ofproto_parser
        mod = parser.OFPFlowMod(
                datapath, command=ofp.OFPFC_DELETE,
                out_port=ofp.OFPP_ANY, out_group=ofp.OFPG_ANY,
                priority=1, match=match)
            datapath.send_msg(mod)
```

OFPFC_DELETE 用于删除指定流表中符合匹配规则（部分匹配即可）的流表项，del_flow()函数将指定交换机中满足匹配域的流表项删除。

（3）　链路状态改变处理函数：

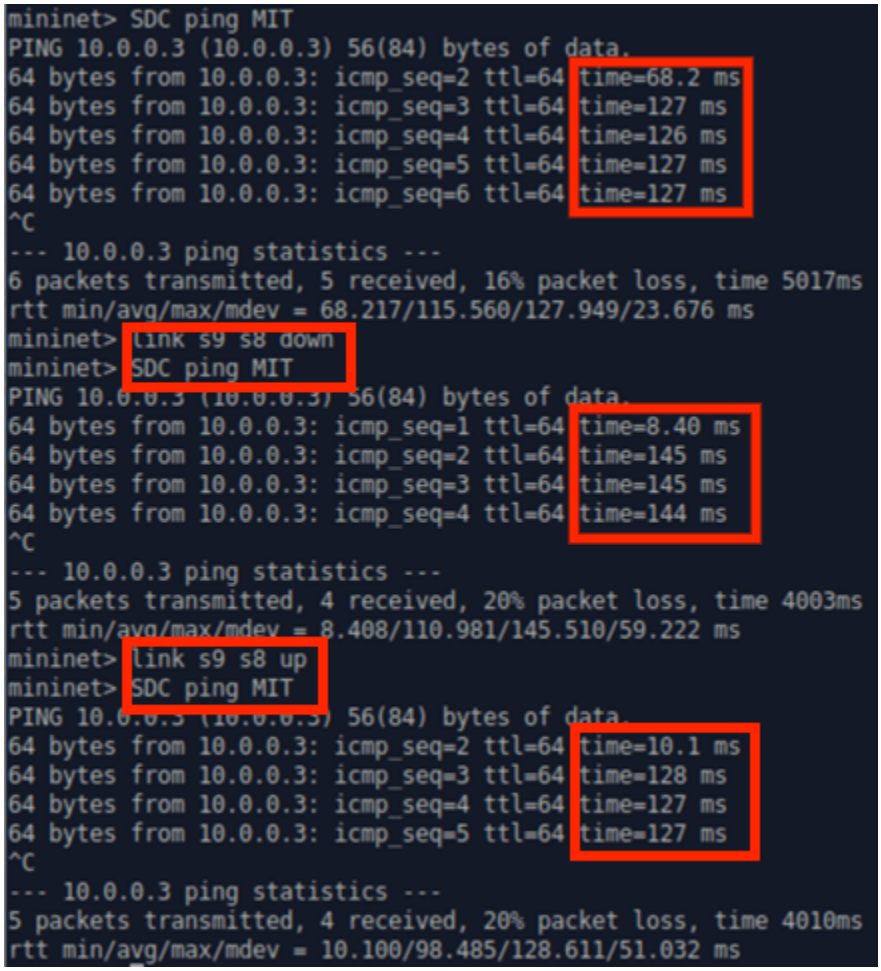
```
    @set_ev_cls(ofp_event.EventOFPPortStatus, MAIN_DISPATCHER)
    def port_status_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        if msg.reason in [ofp.OFPPR_ADD, ofp.OFPPR_MODIFY]:
            dp.ports[msg.desc.port_no] = msg.desc
        elif msg.reason == ofp.OFPPR_DELETE:
            dp.ports.pop(msg.desc.port_no, None)
        else:
            return
        switches = get_switch(self)
        for switch in switches:
            datapath = switch.dp
            match = parser.OFPMatch(eth_type=0x0800)
            self.del_flow(datapath, match)
```

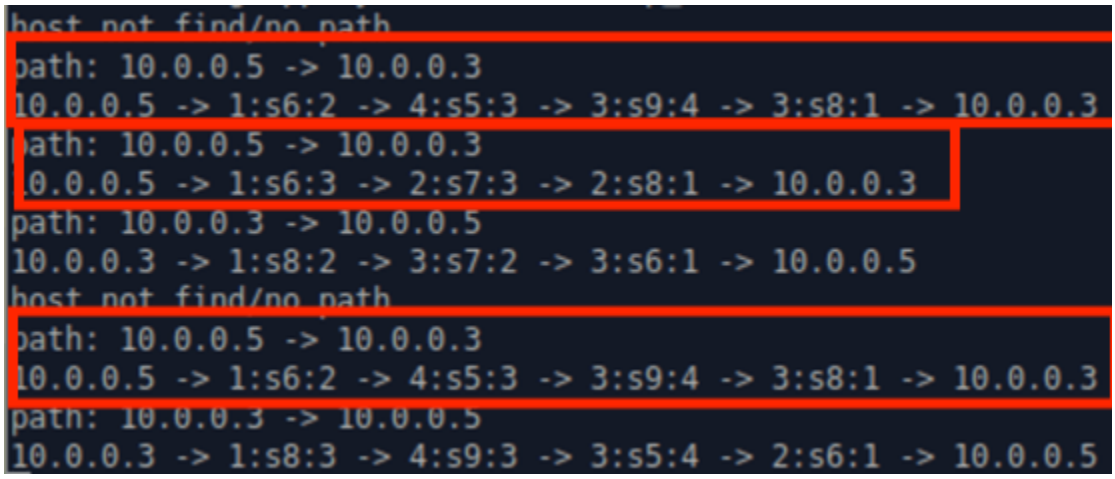
```
            match = parser.OFPMatch(eth_type=0x0806)
            self.del_flow(datapath, match)

        self.mac_to_port = {}
        self.sw = {}
        self.network_awareness.topo_map.clear()

        self.send_event_to_observers(
            ofp_event.EventOFPPortStateChange(dp, msg.reason,
msg.desc.port_no),
            dp.state
        )
```

当链路状态改变时，将执行 port_status_handler()处理函数，该函数遍历网络拓扑中的每个交换机，删除控制器先前下发的协议类型为 ARP 与 IPV4 的流表项，并清除拓扑图

（4）运行结果：

在 mininet 中使用 link down 与 link up 来模拟链路故障与故障恢复。刚开始，SDC 与 MIT 间的最小时延链路为 s6-s5-s9-s8，理论时延为 126ms。当 s9 与 s8 之间链路故障时，最小时延链路为 s6-s7-s8，理论时延为 144ms。





在 mininet 中使用 link down 与 link up 来模拟链路故障与故障恢复。刚开始，SDC 与 MIT 间的最小时延链路为 s6-s5-s9-s8，时延大致为 127ms。当 s9 与 s8 之间链路故障时，最小时延链路为 s6-s7-s8，时延大致为 144ms。