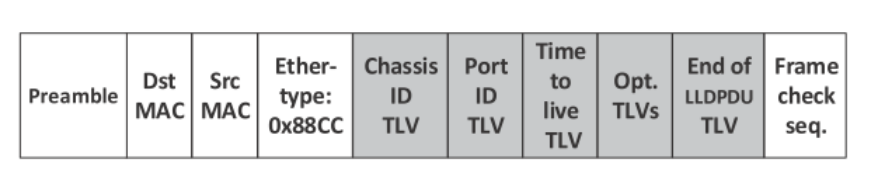
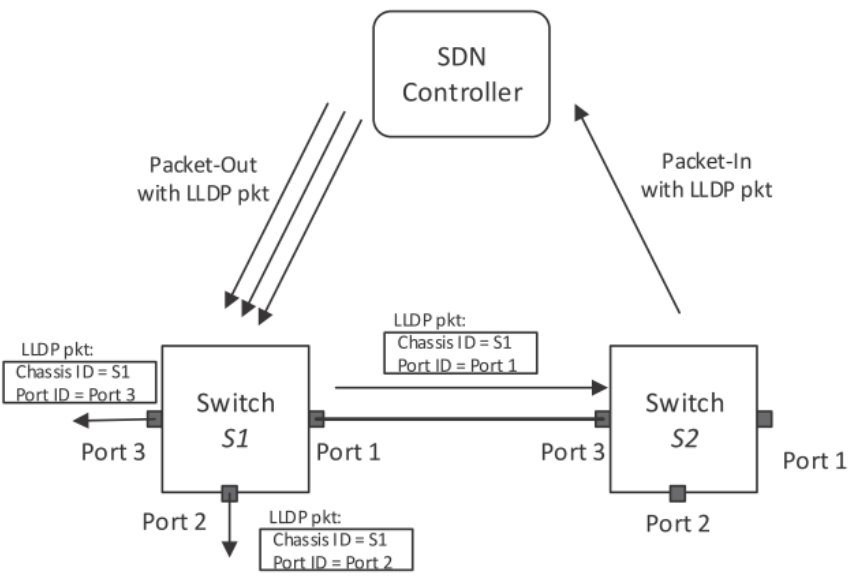
Lab3 测量链路时延与链路容忍

1. **最小时延路径**
2. **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；

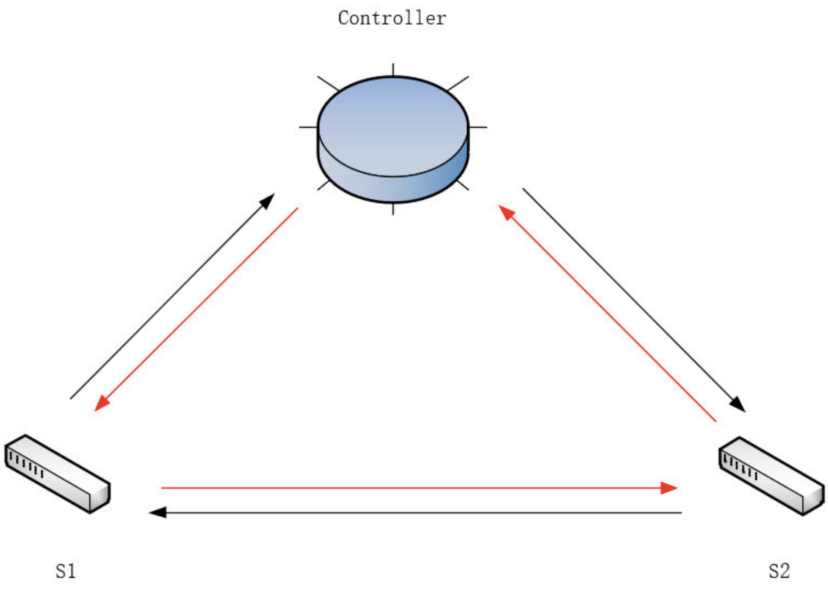
2. 控制器向交换机 S1 中下发流表，流表规则为：将从 Controller 端口收到的 LLDP 数据包从他的对应端口发送出去；

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:')

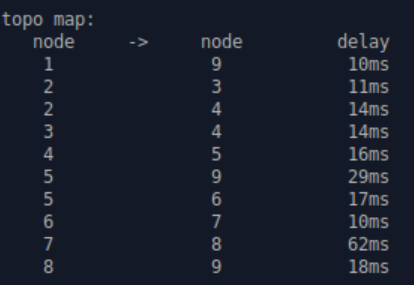
        self.logger.info('{:^10s}  ->  {:^10s}    {:^10s}'.format('node', 'node', 'delay'))

        for src, dst in self.topo\_map.edges:

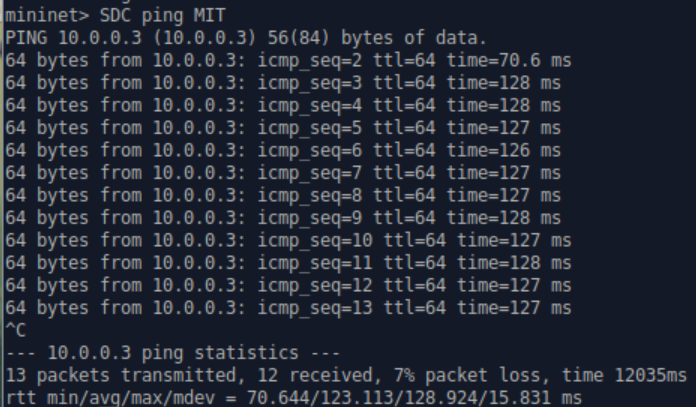
            delay = int(self.topo\_map[src][dst]['delay']\*1000)

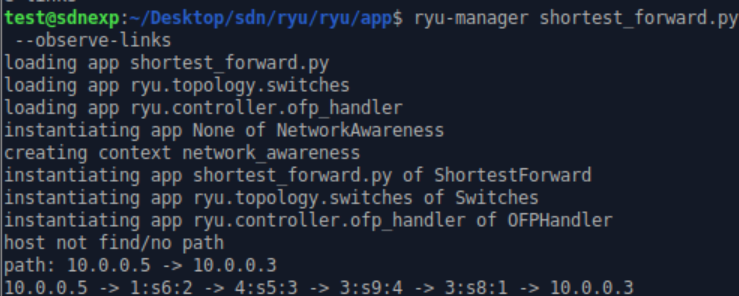
            self.logger.info('{:^10s}      {:^10s}    {:^10s}'.format(str(src), str(dst), str(delay)+'ms'))

        self.logger.info('\n')



(8)运行结果：





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

1. 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。

1. 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。

1. 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),

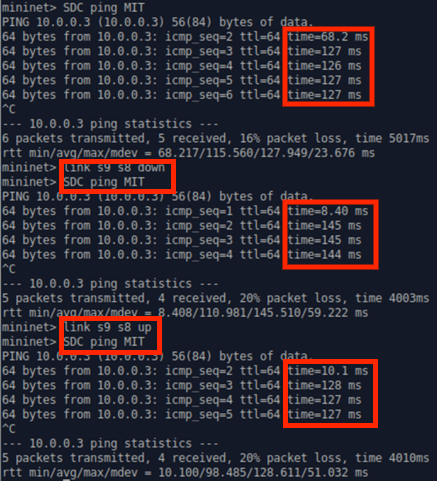
            dp.state

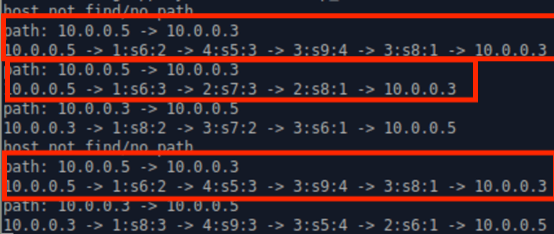
        )

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。