# 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

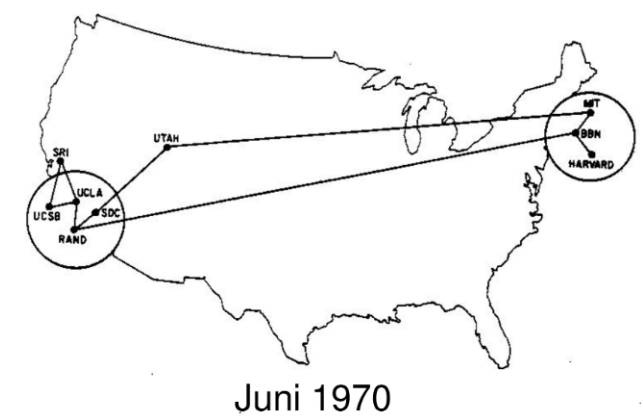
学习设计能够容忍链路故障的路由策略

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：**



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。

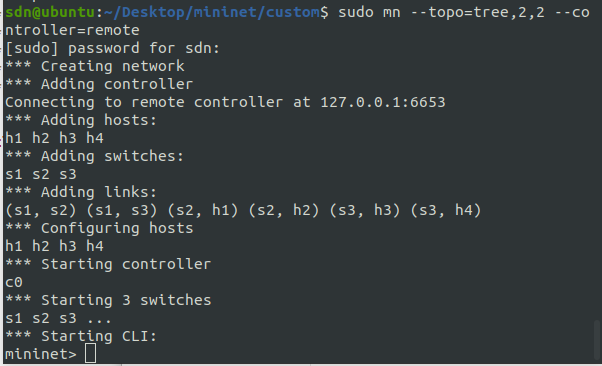
将实验指导书中的NetworkAwareness.py代码保存。在命令行终端内输入：

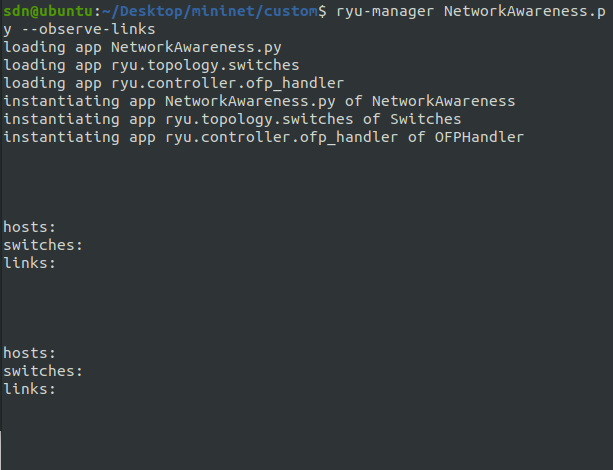
sudo mn --topo=tree,2,2 --controller remote

创建网络拓扑，再在另一个命令行终端内输入：

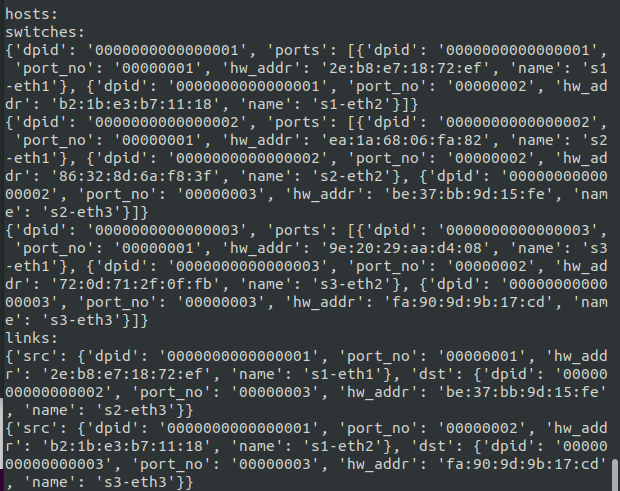
ryu-manager NetworkAwareness.py --observe-links

启动Ryu控制器





可以看到，一开始hosts, switches, links等信息均为空。

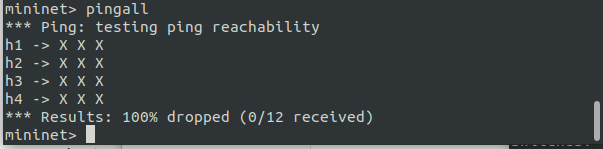


随后switches与links有信息记录，而hosts仍没有信息记录。

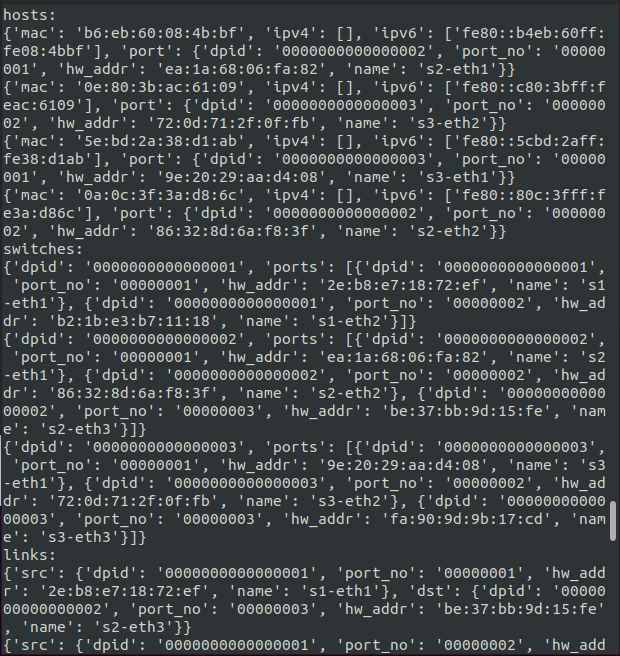
沉默主机现象

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。



可以看到，各节点之间无法ping通。



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。

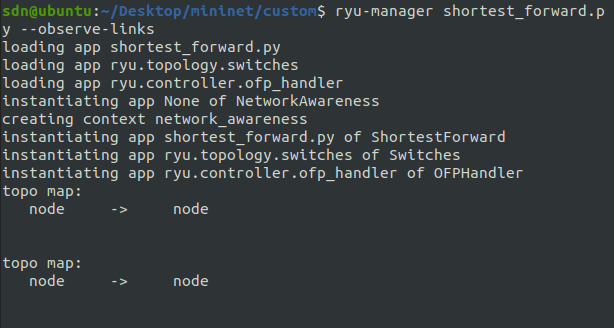
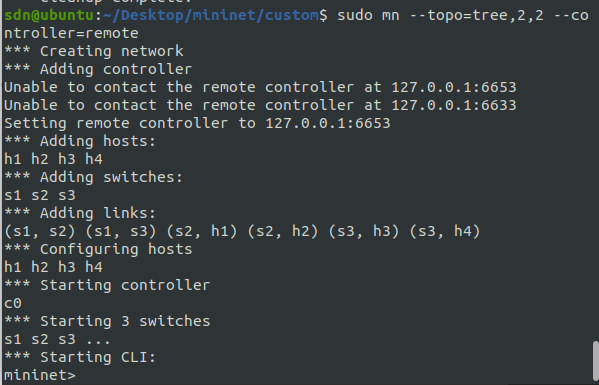
在命令行终端内输入：

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：

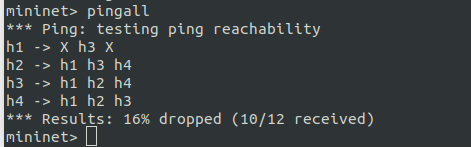
ryu-manager shortest\_forward.py --observe-links

启动Ryu控制器

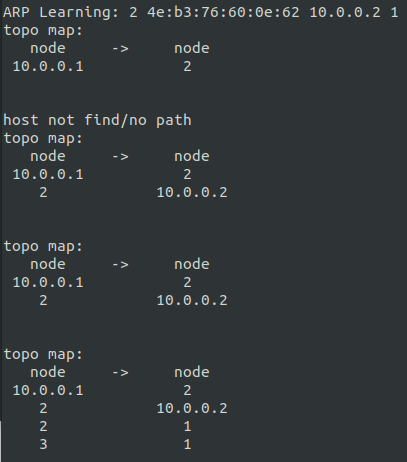


可以看到，在mininet命令行内输入指令前，Ryu控制器没有信息记录。

在mininet命令行内输入pingall，测试各节点之间的连通性。

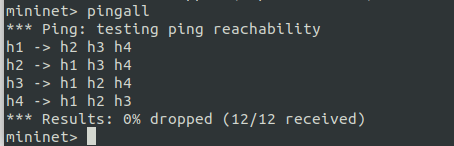


可以看到，只有部分节点能ping通。因为沉默主机现象，前几次ping可能都会输出host not find/no path，这属于正常现象。

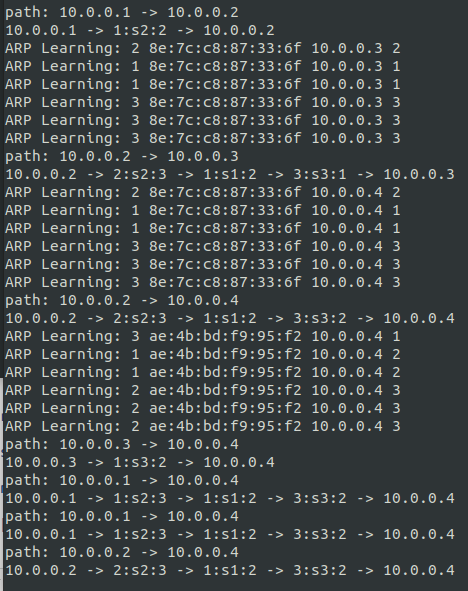


可以看到，在Ryu控制器内显示了交换机ARP记录的信息，同时也显示了host not find/no path。

再次在mininet命令行内输入pingall，测试各节点之间的连通性。



这一次所有12个节点能正常ping通。

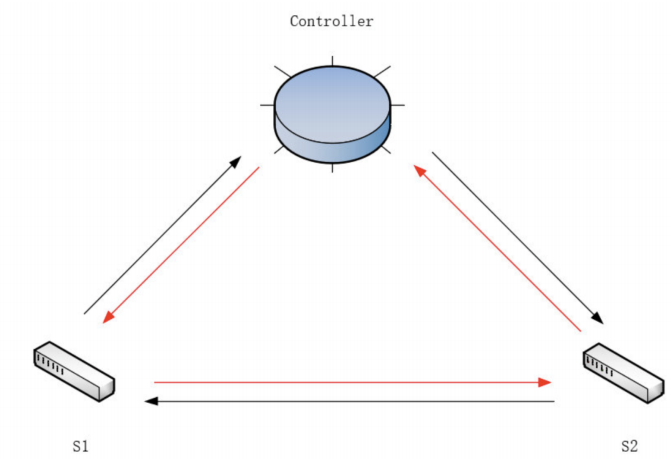


同时Ryu控制器也显示了节点之间最短路径的记录。

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.

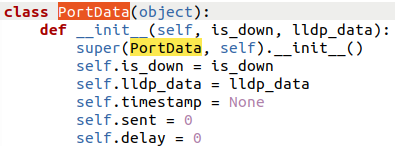
则 S1 到 S2 的时延 delay = (lldp\_delay\_s12 + lldp\_delay\_s21 - echo\_delay\_s1 - echo\_delay\_s2) / 2

为此，我们需要对 Ryu 做如下修改：

1. ryu/topology/Switches.py的PortData/\_\_init\_\_()

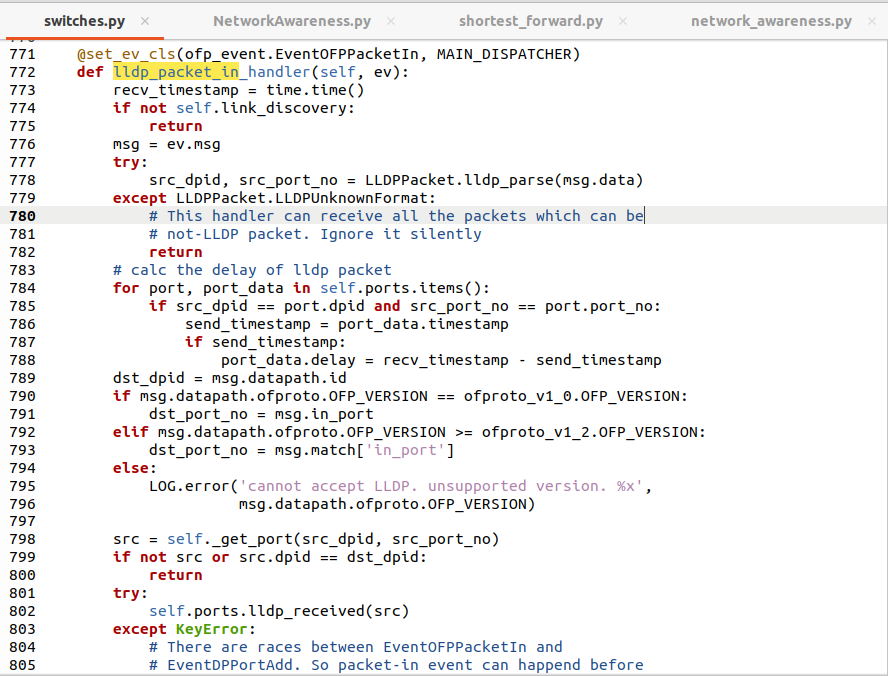
PortData记录交换机的端口信息，我们需要增加self.delay属性记录上述的lldp\_delay。

self.timestamp为LLDP包在发送时被打上的时间戳

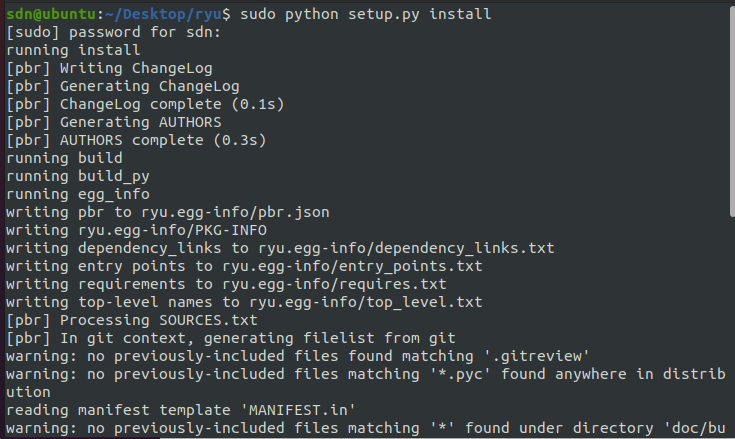


2. ryu/topology/Switches/lldp\_packet\_in\_handler()

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。



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('node', 'node', 'delay'))

        # add one item: delay

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

            self.logger.info('{:^10s}      {:^10s}      {:^10s}'.format(str(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。

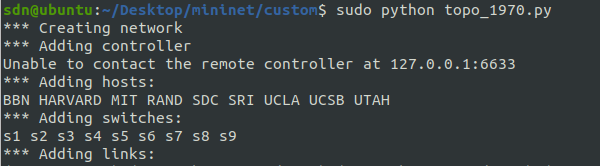
在命令行终端内输入：

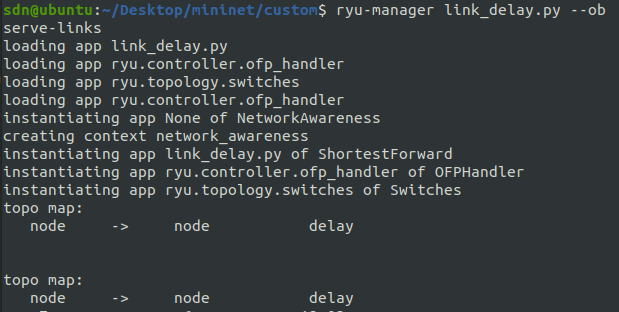
sudo python topo\_1970.py

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

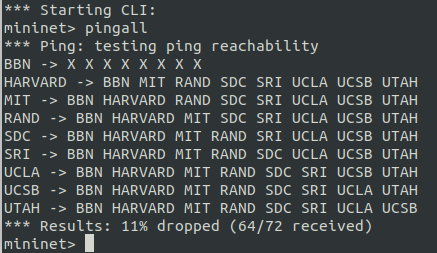
启动Ryu控制器



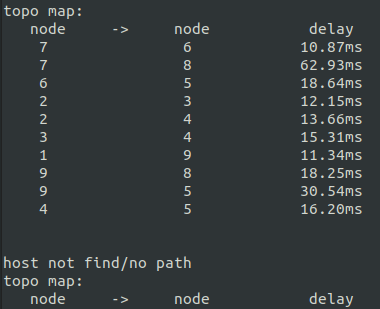


可以看到，一开始没有节点之间链路时延的信息记录。

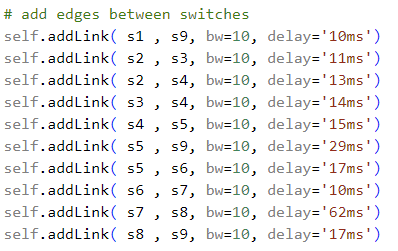
在mininet命令行内输入pingall，测试各节点之间的连通性。



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。

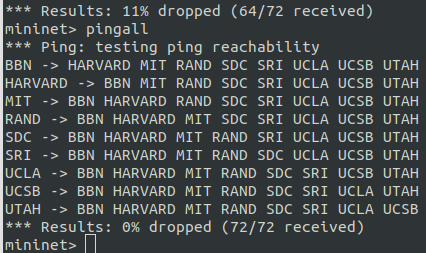


在Ryu控制器中我们可以看到各主机之间的链路时延，与拓扑中预设的时延相比较。



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。



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

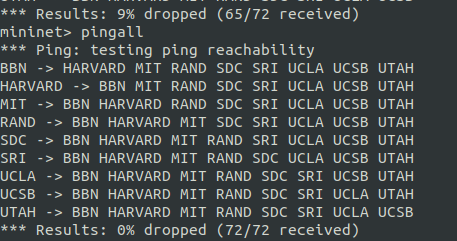
创建网络拓扑，再在另一个命令行终端内输入：

ryu-manager link\_delay.py --observe-links

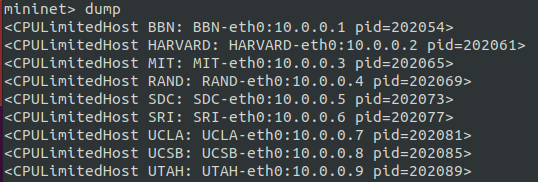
启动Ryu控制器

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。

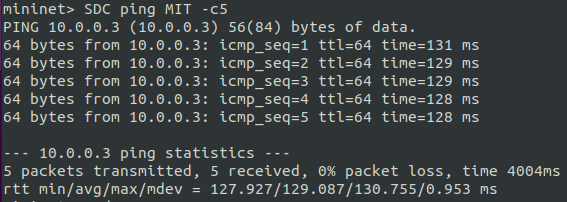


在mininet命令行内输入dump，查看各节点的详细网络信息。



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。

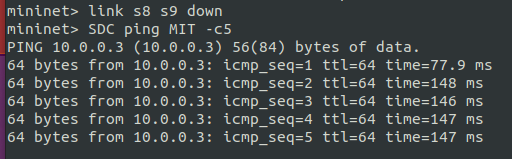


可以看到，此时两个主机之间的链路时延RTT约为128ms。

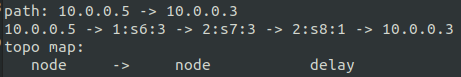


在Ryu控制器打印的信息中，我们可以看到，SDC与MIT之间的链路经过了s8与s9两个交换机，此路径为最短时延路径。

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。

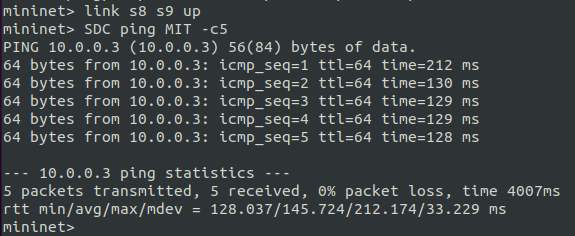


可以看到，断开交换机s8与s9之间的连接后，SDC与MIT主机之间的链路时延RTT约为147ms。

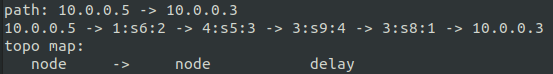


在Ryu控制器打印的信息中，我们可以看到，交换机s8与s9之间的连接断开后，SDC与MIT之间的链路不再经过s8与s9之间的连接。同时由于当前路径不再是最短时延路径，两个主机之间的链路时延RTT也增加了。

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。



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。



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。

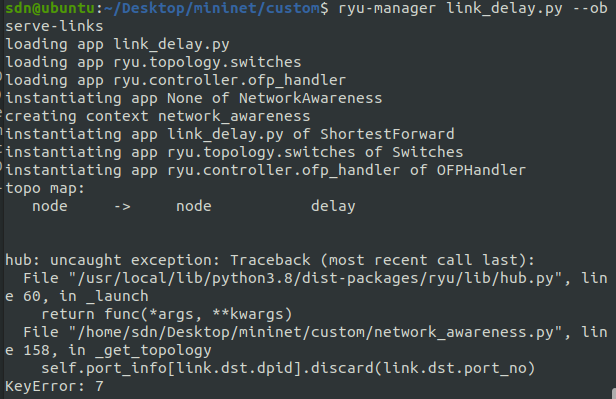
4. 学习了链路故障容忍的原理，自己编写函数修改代码实现主机链路的重新选择。

**实验中遇到的问题：**

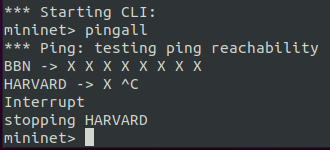
在命令行终端内输入：启动

ryu-manager link\_delay.py --observe-links

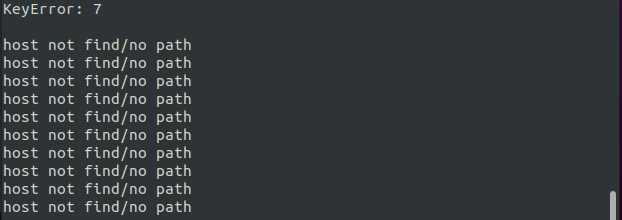
启动Ryu控制器后，偶尔会遇到以下情况



Ryu控制器显示KeyError: 7。



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



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"Key[Error](https://pythonjishu.com/tag/error/" \o "Error" \t "https://pythonjishu.com/python-error-49/_blank)"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('node', 'node', 'delay'))

        # add one item: delay

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

            self.logger.info('{:^10s}      {:^10s}      {:^10s}'.format(str(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

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)

        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)