软件定义网络实验三

目录

[一、 问题简述：最小时延路径 1](#_Toc133150074)

[二、 实验过程 1](#_Toc133150076)

[三、 运行结果 5](#_Toc133150077)

[四、 完整代码 7](#_Toc133150078)

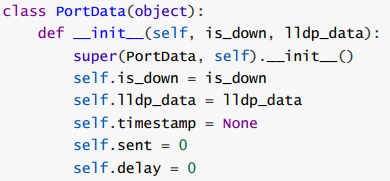
[五、 备注 16](#_Toc133150081)

1. 问题简述：最小时延路径

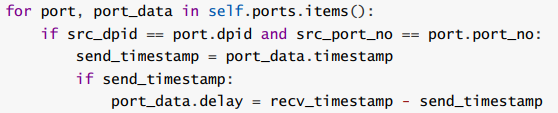
跳数最少的路由不一定是最快的路由，链路时延也会对路由的快慢产生重要影响。请实时地（周期地）利用 LLDP 和 Echo 数据包测量各链路的时延，在网络拓扑的基础上构建一个有权图，然后基于此图计算最小时延路径。具体任务是，找出一条从 SDC 到 MIT 时延最短的路径，输出经过的路线及总的时延，利用 Ping 包的 RTT 验证你的结果。

1. 实验过程

1.准备

在switches里添加port类的delay，用来记录交换机端口对应的时延：

更改lldp\_packet\_in\_handler函数的内容。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：



在安装目录下运行 sudo python setup.py install。

2.具体修改

修改network\_awareness的内容，将其中的weight属性更改为‘delay’（意即令networkx.shortest\_simple\_paths(self.topo\_map, src, dst, weight=weight)函数以delay为边的代价），添加一个link\_delay字典用于存储两个交换机间的时延，并将所添加边属性中的delay属性赋值为networkx中的delay设成相应的时延（此处时延获取方法见下文）：

                dl = (self.lldp\_delay[(link.src.dpid,link.dst.dpid)] + self.lldp\_delay[(link.dst.dpid,link.src.dpid)] - self.echo\_delay[link.dst.dpid] - self.echo\_delay[link.src.dpid]) / 2

                if dl <0:

                    dl =0

                self.topo\_map.add\_edge(link.src.dpid, link.dst.dpid, delay=dl, is\_host=False)

                self.link\_delay[(link.src.dpid,link.dst.dpid)] = dl

初始化一个self.echo\_delay变量，并添加一个死循环线程，用以在后续代码发送echo包：

    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.send\_echo\_request\_thread = hub.spawn(self.\_send\_echo\_request)

        self.topo\_thread = hub.spawn(self.\_get\_topology)

        self.echo\_delay = {}  #dpid: delay

        self.weight = 'delay'

        self.link\_delay = {} #(dpid1,dpid2):delay

        self.lldp\_delay = {} #(src\_dpid,dst\_dpid): lldp\_delay

        self.switches = {}

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，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

若要获取lldp的时延，则可在network\_awareness里更改packet\_in\_hander：

    def packet\_in\_handler(self, ev):

        msg = ev.msg

        dpid = msg.datapath.id

        try:

            src\_dpid, src\_port\_no = LLDPPacket.lldp\_parse(msg.data)

            if not self.switches:

                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

由于if self.switch is None语句会报错，故需将其改为if not self.switches。获取接口保存的delay，并将其存入self.lldp\_delay里，对应源和目的的交换机的id。

然后在network\_awareness里添加发送echo报文和处理echo报文的函数，并计算往返时延，记录到self.echo\_delay里。其中在发送echo报文时应加一个休眠时长，以防止大量echo报文占用信道：

发送：

    def \_send\_echo\_request(self):

        while True:

            for dp in self.switch\_info.values():

                ofp\_parser = dp.ofproto\_parser

                send\_time = time.time() #get the send time

                data = str(send\_time)

                out = ofp\_parser.OFPEchoRequest(dp,data=bytes(data))

                dp.send\_msg(out)

                hub.sleep(SEND\_ECHO\_REQUEST\_INTERVAL)

                hub.sleep(GET\_DELAY\_INTERVAL)

接收：

    @set\_ev\_cls(ofp\_event.EventOFPEchoReply, [MAIN\_DISPATCHER, CONFIG\_DISPATCHER, HANDSHAKE\_DISPATCHER])

    def \_echo\_request\_handler(self,ev):

        msg = ev.msg

        dp = msg.datapath

        dpid = dp.id

        data = msg.data

        recv\_time = time.time()

        send\_time = eval(data)

        echo\_delay = recv\_time - send\_time

        self.echo\_delay[dpid] = echo\_delay

其中修饰符中的第一个参数代表接收到echo包，eval为计算表达式的值的函数，可以获取时间。计算delay后存入echo\_delay字典中。

在echo\_delay和lldp\_delay都获得后，在\_get\_topology里改变delay的值，按照上文计算单条链路的时延：

                dl = (self.lldp\_delay[(link.src.dpid,link.dst.dpid)] + self.lldp\_delay[(link.dst.dpid,link.src.dpid)] - self.echo\_delay[link.dst.dpid] - self.echo\_delay[link.src.dpid]) / 2

                if dl <0:

                    dl =0

此处加判断条件以确保时延大于等于0。

最终保存每个边的权值保存，在shortest\_forward的handle\_ipv4里，需要计算出每条完整路径的时延之和，并且作为预估的RTT输出这里对每个sum\_delay乘以2000的原因是，先换算成毫秒，然后把单向的延时\*2作为双向的往返延时RTT：

        # calc path delay

        sum\_delay = 0

        for i in range(2, len(dpid\_path)-1):

            temp = self.network\_awareness.link\_delay[(dpid\_path[i-1],dpid\_path[i])]

            sum\_delay += temp

        self.logger.info("RTT=")

        self.logger.info(sum\_delay \* 2000)

在给出的删除交换机代码中，没有加条件判断以至于在程序开始时会报错。故需添加if语句以判断其是否存在。

        if ev.state == DEAD\_DISPATCHER:

            if self.switch\_info.has\_key(dpid):

                del self.switch\_info[dpid]

1. 运行结果

为确认结果是否正确，需对比ping的实际时延和自己估算的RTT与预设时延。下面为在topo\_1970中预设的交换机之间的时延：

        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')

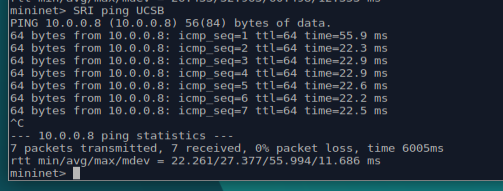
        self.addLink( s6 , s7, bw=10, delay='10ms')

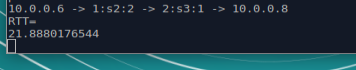
        self.addLink( s7 , s8, bw=10, delay='62ms')

        self.addLink( s8 , s9, bw=10, delay='17ms')

SRI ping UCSB：

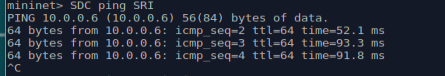
下列为实际时延、路径和自己估算的RTT：

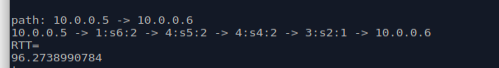




可以看到均在22ms左右，同时通过传播路径可以计算出预设时延为11\*2=22ms，证明了方法的正确性。

SDC ping SRI：

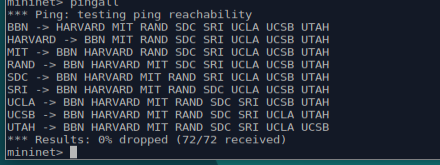


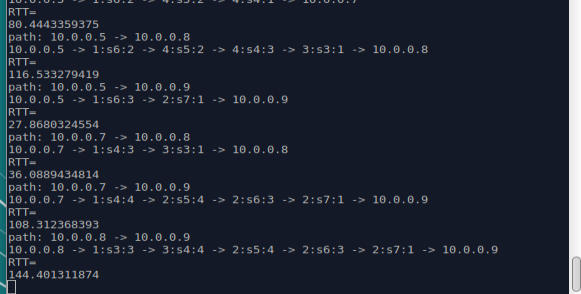


可以看到均在90多毫秒，同时通过传播路径可以计算出预设时延为（17+15+13）\*2=90ms，发现基本都在误差范围内一致，证明了方法的正确性。

Pingall：

可以全部ping通。





1. 完整代码

4.1 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.send\_echo\_request\_thread = hub.spawn(self.\_send\_echo\_request)

        self.topo\_thread = hub.spawn(self.\_get\_topology)

        self.echo\_delay = {}  #dpid: delay

        self.weight = 'delay'

        self.link\_delay = {} #(dpid1,dpid2):delay

        self.lldp\_delay = {} #(src\_dpid,dst\_dpid): lldp\_delay

        self.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)

    @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)

    @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:

            if self.switch\_info.has\_key(dpid):

                del self.switch\_info[dpid]

    @set\_ev\_cls(ofp\_event.EventOFPPacketIn, MAIN\_DISPATCHER)

    def packet\_in\_handler(self, ev):

        msg = ev.msg

        dpid = msg.datapath.id

        try:

            src\_dpid, src\_port\_no = LLDPPacket.lldp\_parse(msg.data)

            if not self.switches:

                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

    def \_send\_echo\_request(self):

        while True:

            for dp in self.switch\_info.values():

                ofp\_parser = dp.ofproto\_parser

                send\_time = time.time() #get the send time

                data = str(send\_time)

                out = ofp\_parser.OFPEchoRequest(dp,data=bytes(data))

                dp.send\_msg(out)

                hub.sleep(SEND\_ECHO\_REQUEST\_INTERVAL)

                hub.sleep(GET\_DELAY\_INTERVAL)

    @set\_ev\_cls(ofp\_event.EventOFPEchoReply, [MAIN\_DISPATCHER, CONFIG\_DISPATCHER, HANDSHAKE\_DISPATCHER])

    def \_echo\_request\_handler(self,ev):

        msg = ev.msg

        dp = msg.datapath

        dpid = dp.id

        data = msg.data

        recv\_time = time.time()

        send\_time = eval(data)

        echo\_delay = recv\_time - send\_time

        self.echo\_delay[dpid] = echo\_delay

        #print("echo\_delay",dpid)

        #print(self.echo\_delay[dpid])

    def \_get\_topology(self):

        hub.sleep(5)

        \_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)

            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

                    #weight = (self.lldp\_delay[(host.port.dpid, host.ipv4[0])] + self.lldp\_delay[(host.ipv4[0],host.port.dpid)] - self.echo\_delay[host.port.dpid] - self.echo\_delay[host.ipv4[0]]) / 2

                    #if weight<0:

                    #    weight = 0

                    self.topo\_map.add\_edge(host.ipv4[0], host.port.dpid, hop=1, delay=0, is\_host=True)

                    #self.link\_delay[(host.ipv4[0],host.port.dpid)] = weight

            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

                dl = (self.lldp\_delay[(link.src.dpid,link.dst.dpid)] + self.lldp\_delay[(link.dst.dpid,link.src.dpid)] - self.echo\_delay[link.dst.dpid] - self.echo\_delay[link.src.dpid]) / 2

                if dl <0:

                    dl =0

                self.topo\_map.add\_edge(link.src.dpid, link.dst.dpid, delay=dl, is\_host=False)

                self.link\_delay[(link.src.dpid,link.dst.dpid)] = dl

            if self.weight == 'delay':

                self.show\_topo\_map()

            hub.sleep(GET\_TOPOLOGY\_INTERVAL)

    def shortest\_path(self, src, dst, weight):

        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}'.format('node', 'node'))

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

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

        self.logger.info('\n')

4.2 shortest\_forward.py

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

from ryu.base import app\_manager

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

from ryu.controller import ofp\_event

from ryu.topology import event

from ryu.lib.packet import ether\_types

import sys

from network\_awareness import NetworkAwareness

import networkx as nx

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'

        self.mac\_to\_port = {}

        self.sw = {}

        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

        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)

    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

        # the identity of switch

        dpid = dp.id

        self.mac\_to\_port.setdefault(dpid, {})

        # the port that receive the packet

        in\_port = msg.match['in\_port']

        if pkt\_type == ether\_types.ETH\_TYPE\_LLDP:

            return

        if pkt\_type == ether\_types.ETH\_TYPE\_IPV6:

            return

        # get protocols

        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

            if header\_list[ARP].opcode == arp.ARP\_REQUEST:

                if (dpid,src,dst\_ip) in self.sw:

                    if self.sw[(dpid,src,dst\_ip)] != in\_port:

                        return

                else:

                    self.sw[(dpid,src,dst\_ip)] = in\_port

        #self.logger.info("packet in %s %s %s %s", dpid, src, dst, in\_port)

        # learn a mac address to avoid FLOOD next time.

        self.mac\_to\_port[dpid][src] = msg.match['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)]

        # install a flow to avoid packet\_in next time

        if out\_port != ofp.OFPP\_FLOOD:

            match = parser.OFPMatch(in\_port=in\_port, eth\_dst=dst, eth\_src=src)

            # verify if we have a valid buffer\_id, if yes avoid to send both

            # flow\_mod & packet\_out

            if msg.buffer\_id != ofp.OFP\_NO\_BUFFER:

                self.add\_flow(dp, 1, match, actions, msg.buffer\_id)

                return

            else:

                self.add\_flow(dp, 1, match, actions)

        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)

    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

        sum\_delay = 0

        for i in range(2, len(dpid\_path)-1):

            temp = self.network\_awareness.link\_delay[(dpid\_path[i-1],dpid\_path[i])]

            sum\_delay += temp

        self.logger.info("RTT=")

        self.logger.info(sum\_delay \* 2000)

        # 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

        \_, 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)

1. 备注

本人因虚拟机有问题，本次实验在他人电脑上完成。