**lab3实验报告**

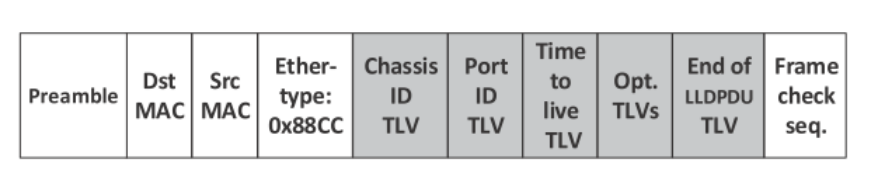
**一、实验原理**

**链路发现原理:**

LLDP(Link Layer Discover Protocol) 即链路层发现协议， OSKen 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 。



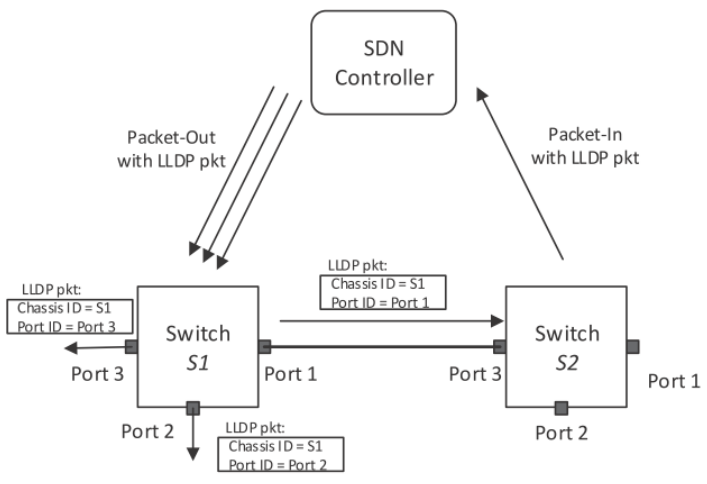
OSKen 如何利⽤ LLDP 发现链路（假设有两个 OpenFlow 交换机连接在控制器上）：

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。



**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 demo.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。

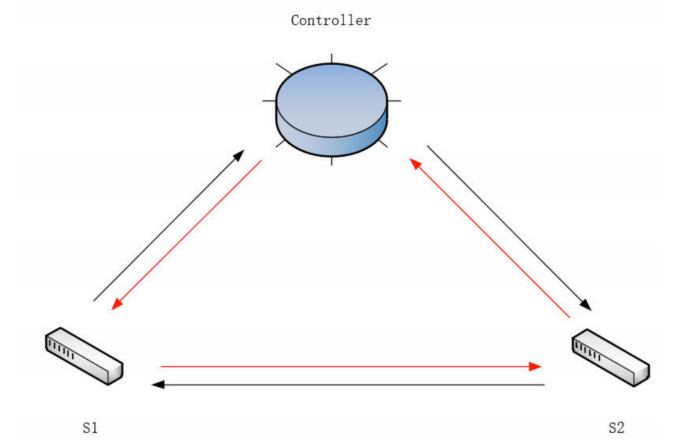
**测量原理：链路时延**

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，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



**二、实验过程**

**（一）必做题：最小时延路径**

在 least\_hops.py 的基础上进行修改：

·以时延作为权重指标self.weight = 'delay'

·补充handle\_arp()处理环路

·补充handle\_ipv4()计算并记录路径延时

在 network\_awareness.py 的基础上进行修改：

·补充packet\_in\_hander()记录lldp\_delay

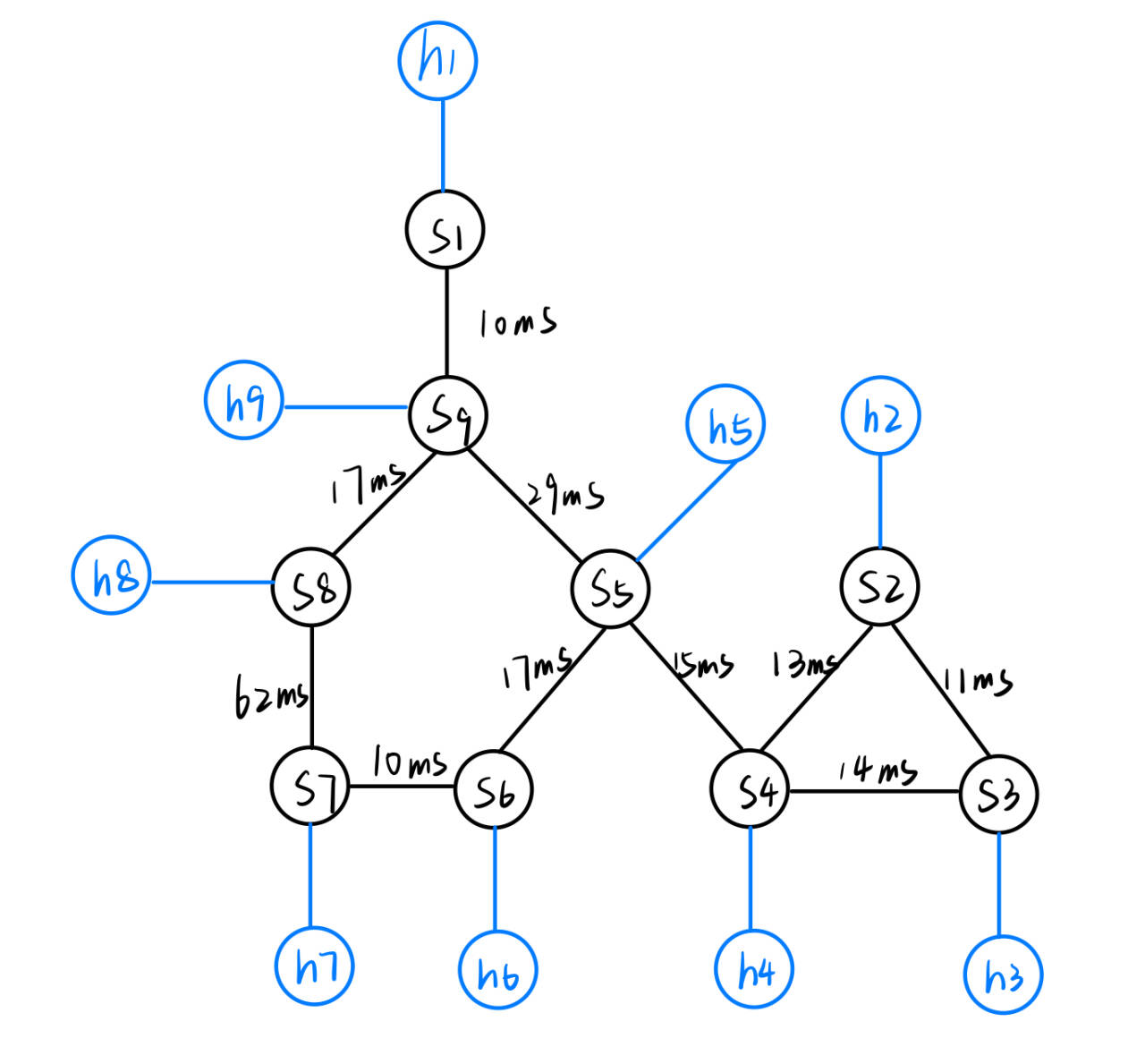
·使用lldp\_delay和echo\_delay字典存储链路延迟数据

·增加calculate\_link\_delay()计算双向链路延迟

·增加echo\_send\_requests()发送Echo请求

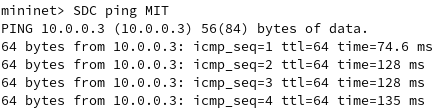
·补充\_get\_topology()计算总延迟delay

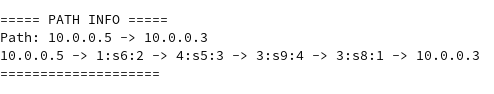
topo\_1970.py拓扑结构：



SDC ping MIT (h6 ping h8)

理论值(17+29+17)×2=126ms





**（二）选做题：容忍链路故障**

在任务一的基础上修改network\_awareness.py

·补充事件处理函数port\_status\_handler()检测端口变化，当端口状态变化时清空拓扑图

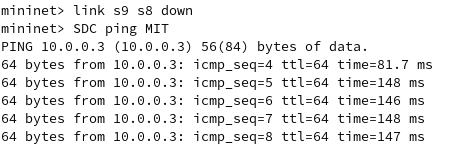
·补充delete\_flow()删除指定交换机上特定端口关联的所有流表项

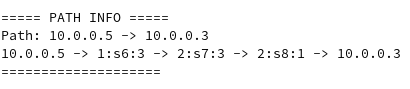
|  |
| --- |
| @set\_ev\_cls(ofp\_event.EventOFPPortStatus, MAIN\_DISPATCHER)  def port\_status\_handler(self, ev):  self.logger.info("Note: Now in port\_status\_handle")  msg = ev.msg  datapath = msg.datapath  ofproto = datapath.ofproto  if msg.reason in [ofproto.OFPPR\_ADD, ofproto.OFPPR\_MODIFY]:  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]:  self.delete\_flow(self.switch\_info[dpid],port)  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) |

|  |
| --- |
| def delete\_flow(self, datapath,port\_no):  ofproto = datapath.ofproto  parser = datapath.ofproto\_parser  try:  match = parser.OFPMatch(in\_port=port\_no)  flow\_mod = parser.OFPFlowMod(datapath=datapath,command=ofproto.OFPFC\_DELETE,out\_port=ofproto.OFPP\_ANY,  out\_group=ofproto.OFPG\_ANY,match=match)  datapath.send\_msg(flow\_mod)  self.logger.info("Deleted flow entries associated with port %s on switch %s", port\_no, datapath.id)  except Exception as e:  self.logger.error("Failed to delete flow entries associated with port %s on switch %s: %s",  port\_no, datapath.id, str(e)) |

删除link s9 s8

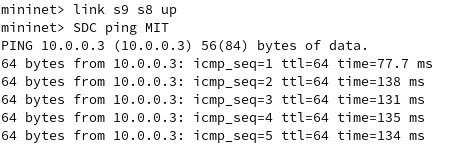
理论值(10+62)×2=144ms

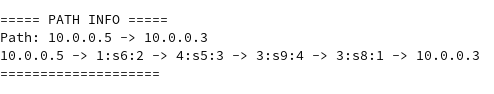




恢复链路

理论值(17+29+17)×2=126ms





与实验一结果一致

**三、代码附录**

ShortestForward.py

|  |
| --- |
| from os\_ken.base import app\_manager  from os\_ken.controller import ofp\_event  from os\_ken.controller.handler import CONFIG\_DISPATCHER, MAIN\_DISPATCHER, DEAD\_DISPATCHER, HANDSHAKE\_DISPATCHER  from os\_ken.controller.handler import set\_ev\_cls  from os\_ken.controller.handler import set\_ev\_cls  from os\_ken.ofproto import ofproto\_v1\_3  from os\_ken.lib.packet import packet  from os\_ken.lib.packet import ethernet, arp, ipv4  from os\_ken.lib.packet import ether\_types  from os\_ken.controller import ofp\_event  from os\_ken.topology import event  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.OSKenApp):  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  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  dpid = dp.id  self.mac\_to\_port.setdefault(dpid, {})  eth\_pkt = pkt.get\_protocol(ethernet.ethernet)  if eth\_pkt.ethertype == ether\_types.ETH\_TYPE\_LLDP:  return  if eth\_pkt.ethertype == ether\_types.ETH\_TYPE\_IPV6:  return  dst = eth\_pkt.dst  src = eth\_pkt.src  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\_packet = header\_list[ARP]  dst\_ip = arp\_packet.dst\_ip  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.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: # Already learned, therefore sending flow table entries  match = parser.OFPMatch(in\_port = in\_port, eth\_src = src, eth\_dst = dst)  self.add\_flow(dp, 1, match, actions,idle\_timeout = 5,hard\_timeout = 5)  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)  # calculate path delay  delay = 0  for i in range(len(dpid\_path) - 1):  src\_dpid = dpid\_path[i]  dst\_dpid = dpid\_path[i + 1]  delay += self.network\_awareness.lldp\_delay.get((src\_dpid, dst\_dpid), 0)  # 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) |

network\_awareness.py

|  |
| --- |
| # network\_awareness.py  from os\_ken.base import app\_manager  from os\_ken.base.app\_manager import lookup\_service\_brick  from os\_ken.ofproto import ofproto\_v1\_3  from os\_ken.controller.handler import set\_ev\_cls  from os\_ken.controller.handler import MAIN\_DISPATCHER, CONFIG\_DISPATCHER, DEAD\_DISPATCHER, HANDSHAKE\_DISPATCHER  from os\_ken.controller import ofp\_event  from os\_ken.lib.packet import packet, ethernet, arp, ether\_types  from os\_ken.lib import hub  from os\_ken.topology import event  from os\_ken.topology.api import get\_host, get\_link, get\_switch  from os\_ken.topology.switches import LLDPPacket  import networkx as nx  import copy  import time  import struct  GET\_TOPOLOGY\_INTERVAL = 2  SEND\_ECHO\_REQUEST\_INTERVAL = .05  GET\_DELAY\_INTERVAL = 2  class NetworkAwareness(app\_manager.OSKenApp):  OFP\_VERSIONS = [ofproto\_v1\_3.OFP\_VERSION]  def \_\_init\_\_(self, \*args, \*\*kwargs):  super(NetworkAwareness, self).\_\_init\_\_(\*args, \*\*kwargs)  self.switch\_info = {}  self.link\_info = {}  self.port\_link={}  self.port\_info = {}  self.topo\_map = nx.Graph()  self.topo\_thread = hub.spawn(self.\_get\_topology)  self.switches = {}  self.lldp\_delay = {}  self.echo\_delay = {}  self.weight = 'delay'  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:  del self.switch\_info[dpid]    @set\_ev\_cls(ofp\_event.EventOFPPacketIn, MAIN\_DISPATCHER)  def packet\_in\_hander(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    @set\_ev\_cls(ofp\_event.EventOFPEchoReply, MAIN\_DISPATCHER)  def echo\_hander(self,ev):  recv\_time = time.time()  try:  msg = ev.msg  dpid = msg.datapath.id  send\_time = struct.unpack('d', msg.data)[0]  self.echo\_delay[dpid] = recv\_time - send\_time  except:  return    @set\_ev\_cls(ofp\_event.EventOFPPortStatus, MAIN\_DISPATCHER)  def port\_status\_handler(self, ev):  self.logger.info("Note: Now in port\_status\_handle")  msg = ev.msg  datapath = msg.datapath  ofproto = datapath.ofproto  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]:  self.delete\_flow(self.switch\_info[dpid],port)  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)  def echo\_send\_requests(self, switch):  datapath = switch.dp  parser = datapath.ofproto\_parser  send\_time = time.time()  data = struct.pack('d', send\_time)  echo\_req = parser.OFPEchoRequest(datapath, data = data)  datapath.send\_msg(echo\_req)  hub.sleep(0.2)  return    def delete\_flow(self, datapath,port\_no):  ofproto = datapath.ofproto  parser = datapath.ofproto\_parser  try:  match = parser.OFPMatch(in\_port=port\_no)  flow\_mod = parser.OFPFlowMod(datapath=datapath,command=ofproto.OFPFC\_DELETE,out\_port=ofproto.OFPP\_ANY,  out\_group=ofproto.OFPG\_ANY,match=match)  datapath.send\_msg(flow\_mod)  self.logger.info("Deleted flow entries associated with port %s on switch %s", port\_no, datapath.id)  except Exception as e:  self.logger.error("Failed to delete flow entries associated with port %s on switch %s: %s",  port\_no, datapath.id, str(e))  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.echo\_send\_requests(switch)  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  delay = self.calculate\_link\_delay(link.src.dpid, link.dst.dpid)  self.logger.info("Added link: src\_dpid=%s dst\_dpid=%s delay= %.5fms",link.src.dpid, link.dst.dpid, delay\*1000)  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 calculate\_link\_delay(self, src\_dpid, dst\_dpid):  lldp\_delay\_s12 = self.lldp\_delay.get((src\_dpid, dst\_dpid), 0)  lldp\_delay\_s21 = self.lldp\_delay.get((dst\_dpid, src\_dpid), 0)  echo\_delay\_s1 = self.echo\_delay.get(src\_dpid, 0)  echo\_delay\_s2 = self.echo\_delay.get(dst\_dpid, 0)  delay = (lldp\_delay\_s12 + lldp\_delay\_s21 - echo\_delay\_s1 - echo\_delay\_s2) / 2  return max(delay, 0)  def shortest\_path(self, src, dst, weight='delay'):  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') |