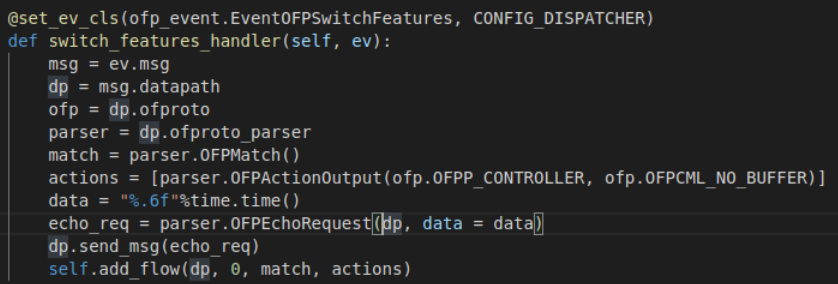
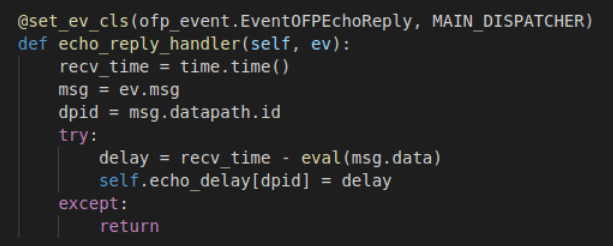
**SDN实验二**

1. 最小时延路径：

首先需要测量链路的时延，lldp\_delay可以直接通过交换机的端口获取，echo\_delay则需要自行测量，在本次实验中，通过在交换机启动时控制器向交换机发送echo\_request数据包，控制器收到echo\_reply数据包后，计算出echo\_delay。

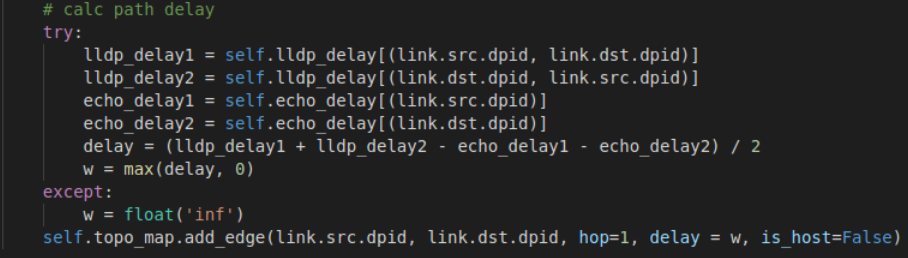


控制器向交换机发送echo\_request数据包

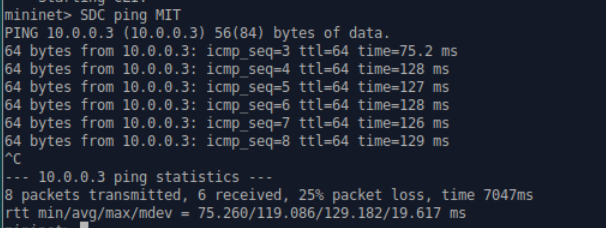


控制器收到echo\_reply后计算时延

之后跟据测量出的lldp\_delay和echo\_delay计算出链路的delay，并将链路的时延作为weight，计算出最短路径



实验结果：

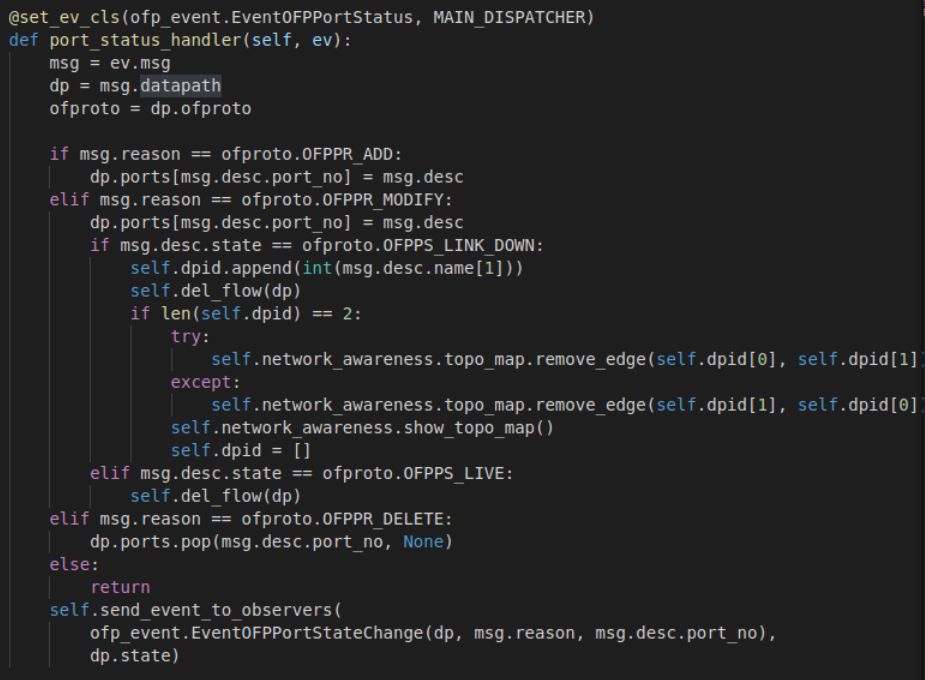




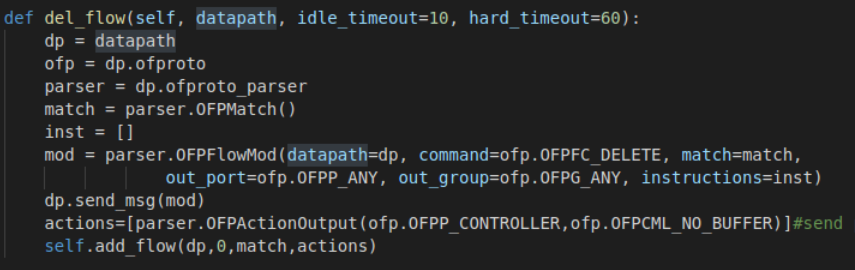
与预先设定的链路时延比较，控制器计算出的路径是时延最短路径。

1. 容忍链路故障：

首先需要捕捉到链路故障，当感知到链路断开或连接后，需要删除链路上相关交换机的流表以及保存的拓扑图中的边，同时下发默认流表，之后控制器计算出新的路径并下发新的流表。

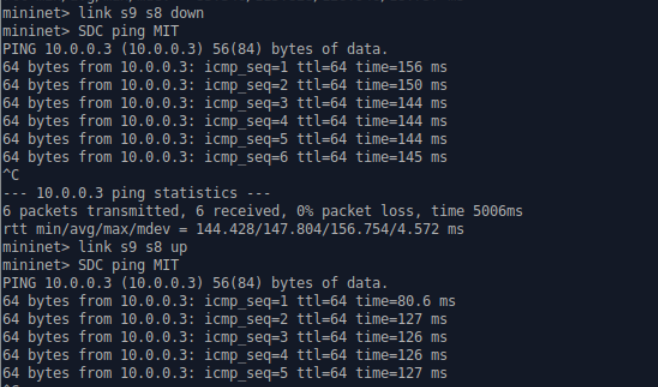


控制器捕捉链路故障并作出对应处理

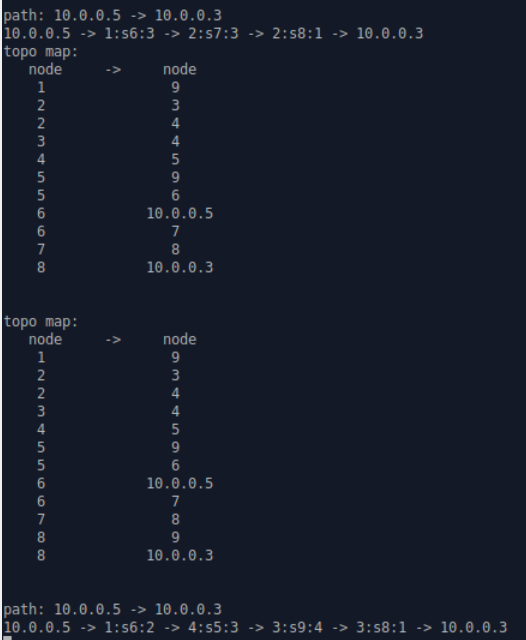


交换机删除当前流表并下发默认流表

实验结果：



断开s9和s8之间的链路后，控制器重新计算了SDC到MIT的路径，重新连接后，又恢复到之前的路径。



1. 实验中遇到的问题：
2. 当链路断开时，需要删除拓扑图中对应的边（或者直接删除整个拓扑？）
3. 路径正确，时延错误。观察各个交换机的流表发现，可能是arp请求时下发的流表影响了路由，因此提升了下分的路由流表的优先级，同时给arp流表设置了有效时间，超过设置的时间后自动删除。之后问题解决。
4. 实验代码：

network\_awareness.py

1. **from** ryu.base **import** app\_manager
2. **from** ryu.base.app\_manager **import** lookup\_service\_brick
3. **from** ryu.ofproto **import** ofproto\_v1\_3
4. **from** ryu.controller.handler **import** set\_ev\_cls
5. **from** ryu.controller.handler **import** MAIN\_DISPATCHER, CONFIG\_DISPATCHER, DEAD\_DISPATCHER
6. **from** ryu.controller **import** ofp\_event
7. **from** ryu.lib.packet **import** packet
8. **from** ryu.lib.packet **import** ethernet, arp
9. **from** ryu.lib **import** hub
10. **from** ryu.topology **import** event
11. **from** ryu.topology.api **import** get\_host, get\_link, get\_switch
12. **from** ryu.topology.switches **import** LLDPPacket
13. **from** ryu.base.app\_manager **import** lookup\_service\_brick
14. **import** networkx as nx
15. **import** copy
16. **import** time

19. GET\_TOPOLOGY\_INTERVAL = 2
20. SEND\_ECHO\_REQUEST\_INTERVAL = .05
21. GET\_DELAY\_INTERVAL = 2

24. **class** NetworkAwareness(app\_manager.RyuApp):
25. OFP\_VERSIONS = [ofproto\_v1\_3.OFP\_VERSION]
27. **def** \_\_init\_\_(self, \*args, \*\*kwargs):
28. super(NetworkAwareness, self).\_\_init\_\_(\*args, \*\*kwargs)
29. self.switch\_info = {}  # dpid: datapath
30. self.link\_info = {}  # (s1, s2): s1.port
31. self.port\_link={} # s1,port:s1,s2
32. self.port\_info = {}  # dpid: (ports linked hosts)
33. self.topo\_map = nx.Graph()
34. self.topo\_thread = hub.spawn(self.\_get\_topology)
35. self.weight = 'hop'
36. self.switches = None
37. self.lldp\_delay={}
38. self.echo\_delay={}

41. **def** add\_flow(self, datapath, priority, match, actions):
42. dp = datapath
43. ofp = dp.ofproto
44. parser = dp.ofproto\_parser
46. inst = [parser.OFPInstructionActions(ofp.OFPIT\_APPLY\_ACTIONS, actions)]
47. mod = parser.OFPFlowMod(datapath=dp, priority=priority, match=match, instructions=inst)
48. dp.send\_msg(mod)

51. @set\_ev\_cls(ofp\_event.EventOFPSwitchFeatures, CONFIG\_DISPATCHER)
52. **def** switch\_features\_handler(self, ev):
53. msg = ev.msg
54. dp = msg.datapath
55. ofp = dp.ofproto
56. parser = dp.ofproto\_parser
57. match = parser.OFPMatch()
58. actions = [parser.OFPActionOutput(ofp.OFPP\_CONTROLLER, ofp.OFPCML\_NO\_BUFFER)]
59. data = "%.6f"%time.time()
60. echo\_req = parser.OFPEchoRequest(dp, data = data)
61. dp.send\_msg(echo\_req)
62. self.add\_flow(dp, 0, match, actions)
64. @set\_ev\_cls(ofp\_event.EventOFPStateChange, [MAIN\_DISPATCHER, DEAD\_DISPATCHER])
65. **def** state\_change\_handler(self, ev):
66. dp = ev.datapath
67. dpid = dp.id
69. **if** ev.state == MAIN\_DISPATCHER:
70. self.switch\_info[dpid] = dp
72. **if** ev.state == DEAD\_DISPATCHER:
73. **del** self.switch\_info[dpid]
74. **def** \_get\_topology(self):
75. \_hosts, \_switches, \_links = None, None, None
76. **while** True:
77. hosts = get\_host(self)
78. switches = get\_switch(self)
79. links = get\_link(self)
81. # update topo\_map when topology change
82. **if** [str(x) **for** x **in** hosts] == \_hosts **and** [str(x) **for** x **in** switches] == \_switches **and** [str(x) **for** x **in** links] == \_links:
83. **continue**
84. \_hosts, \_switches, \_links = [str(x) **for** x **in** hosts], [str(x) **for** x **in** switches], [str(x) **for** x **in** links]
85. **for** switch **in** switches:
86. self.port\_info.setdefault(switch.dp.id, set())
87. # record all ports
88. **for** port **in** switch.ports:
89. self.port\_info[switch.dp.id].add(port.port\_no)
91. **for** host **in** hosts:
92. # take one ipv4 address as host id
93. **if** host.ipv4:
94. self.link\_info[(host.port.dpid, host.ipv4[0])] = host.port.port\_no
95. self.topo\_map.add\_edge(host.ipv4[0], host.port.dpid, hop=1, delay=0, is\_host=True)
96. **for** link **in** links:
97. # delete ports linked switches
98. self.port\_info[link.src.dpid].discard(link.src.port\_no)
99. self.port\_info[link.dst.dpid].discard(link.dst.port\_no)
101. # s1 -> s2: s1.port, s2 -> s1: s2.port
102. self.port\_link[(link.src.dpid,link.src.port\_no)] = (link.src.dpid, link.dst.dpid)
103. self.port\_link[(link.dst.dpid,link.dst.port\_no)] = (link.dst.dpid, link.src.dpid)
105. self.link\_info[(link.src.dpid, link.dst.dpid)] = link.src.port\_no
106. self.link\_info[(link.dst.dpid, link.src.dpid)] = link.dst.port\_no
107. # calc path delay
108. **try**:
109. lldp\_delay1 = self.lldp\_delay[(link.src.dpid, link.dst.dpid)]
110. lldp\_delay2 = self.lldp\_delay[(link.dst.dpid, link.src.dpid)]
111. echo\_delay1 = self.echo\_delay[(link.src.dpid)]
112. echo\_delay2 = self.echo\_delay[(link.dst.dpid)]
113. delay = (lldp\_delay1 + lldp\_delay2 - echo\_delay1 - echo\_delay2) / 2
114. w = max(delay, 0)
115. **except**:
116. w = float('inf')
117. self.topo\_map.add\_edge(link.src.dpid, link.dst.dpid, hop=1, delay = w, is\_host=False)
119. **if** self.weight == 'hop':
120. self.show\_topo\_map()
121. hub.sleep(GET\_TOPOLOGY\_INTERVAL)
123. **def** shortest\_path(self, src, dst, weight='hop'):
124. **try**:
125. paths = list(nx.shortest\_simple\_paths(self.topo\_map, src, dst, weight=weight))
126. **return** paths[0]
127. **except**:
128. self.logger.info('host not find/no path')
130. **def** show\_topo\_map(self):
131. self.logger.info('topo map:')
132. self.logger.info('{:^10s}  ->  {:^10s}'.format('node', 'node'))
133. **for** src, dst **in** self.topo\_map.edges:
134. self.logger.info('{:^10s}      {:^10s}'.format(str(src), str(dst)))
135. self.logger.info('\n')
137. @set\_ev\_cls(ofp\_event.EventOFPPacketIn, MAIN\_DISPATCHER)
138. **def** packet\_in\_handler(self, ev):
139. msg = ev.msg
140. dpid = msg.datapath.id
141. self.lldp\_delay.setdefault(dpid, {})
142. **try**:
143. src\_dpid, src\_port\_no = LLDPPacket.lldp\_parse(msg.data)
144. **if** self.switches **is** None:
145. self.switches = lookup\_service\_brick('switches')
146. **for** port **in** self.switches.ports.keys():
147. **if** src\_dpid == port.dpid **and** src\_port\_no == port.port\_no:
148. self.lldp\_delay[(src\_dpid, dpid)] = self.switches.ports[port].delay
149. **except**:
150. **return**
152. @set\_ev\_cls(ofp\_event.EventOFPEchoReply, MAIN\_DISPATCHER)
153. **def** echo\_reply\_handler(self, ev):
154. recv\_time = time.time()
155. msg = ev.msg
156. dpid = msg.datapath.id
157. **try**:
158. delay = recv\_time - eval(msg.data)
159. self.echo\_delay[dpid] = delay
160. **except**:
161. **return**

shortest\_forward.py

1. **from** ryu.base **import** app\_manager
2. **from** ryu.controller **import** ofp\_event
3. **from** ryu.controller.handler **import** CONFIG\_DISPATCHER, MAIN\_DISPATCHER, DEAD\_DISPATCHER, HANDSHAKE\_DISPATCHER
4. **from** ryu.controller.handler **import** set\_ev\_cls
5. **from** ryu.controller.handler **import** set\_ev\_cls
6. **from** ryu.ofproto **import** ofproto\_v1\_3
7. **from** ryu.lib.packet **import** packet
8. **from** ryu.lib.packet **import** ethernet, arp, ipv4, lldp
9. **from** ryu.controller **import** ofp\_event
10. **from** ryu.topology **import** event
11. **import** sys
12. **from** network\_awareness **import** NetworkAwareness
13. **import** networkx as nx
15. ETHERNET = ethernet.ethernet.\_\_name\_\_
16. ETHERNET\_MULTICAST = "ff:ff:ff:ff:ff:ff"
17. ARP = arp.arp.\_\_name\_\_
18. **class** ShortestForward(app\_manager.RyuApp):
19. OFP\_VERSIONS = [ofproto\_v1\_3.OFP\_VERSION]
20. \_CONTEXTS = {
21. 'network\_awareness': NetworkAwareness
22. }
24. **def** \_\_init\_\_(self, \*args, \*\*kwargs):
25. super(ShortestForward, self).\_\_init\_\_(\*args, \*\*kwargs)
26. self.network\_awareness = kwargs['network\_awareness']
27. self.weight = 'hop'
28. self.mac\_to\_port = {}
29. self.sw = {}
30. self.path=None
31. self.dpid=[]
33. **def** add\_flow(self, datapath, priority, match, actions, idle\_timeout=0, hard\_timeout=0):
34. dp = datapath
35. ofp = dp.ofproto
36. parser = dp.ofproto\_parser
38. inst = [parser.OFPInstructionActions(ofp.OFPIT\_APPLY\_ACTIONS, actions)]
39. mod = parser.OFPFlowMod(
40. datapath=dp, priority=priority,
41. idle\_timeout=idle\_timeout,
42. hard\_timeout=hard\_timeout,
43. match=match, instructions=inst)
44. dp.send\_msg(mod)
46. **def** del\_flow(self, datapath, idle\_timeout=10, hard\_timeout=60):
47. dp = datapath
48. ofp = dp.ofproto
49. parser = dp.ofproto\_parser
50. match = parser.OFPMatch()
51. inst = []
52. mod = parser.OFPFlowMod(datapath=dp, command=ofp.OFPFC\_DELETE, match=match,
53. out\_port=ofp.OFPP\_ANY, out\_group=ofp.OFPG\_ANY, instructions=inst)
54. dp.send\_msg(mod)
55. actions=[parser.OFPActionOutput(ofp.OFPP\_CONTROLLER,ofp.OFPCML\_NO\_BUFFER)]#send packets to controller
56. self.add\_flow(dp,0,match,actions)

59. @set\_ev\_cls(ofp\_event.EventOFPPacketIn, MAIN\_DISPATCHER)
60. **def** packet\_in\_handler(self, ev):
61. msg = ev.msg
62. dp = msg.datapath
63. ofp = dp.ofproto
64. parser = dp.ofproto\_parser
66. dpid = dp.id
67. in\_port = msg.match['in\_port']
69. self.mac\_to\_port.setdefault(dpid, {})
70. self.sw.setdefault(dpid, {})
72. pkt = packet.Packet(msg.data)
73. eth\_pkt = pkt.get\_protocol(ethernet.ethernet)
74. arp\_pkt = pkt.get\_protocol(arp.arp)
75. ipv4\_pkt = pkt.get\_protocol(ipv4.ipv4)
76. pkt\_type = eth\_pkt.ethertype
78. # layer 2 self-learning
79. dst\_mac = eth\_pkt.dst
80. src\_mac = eth\_pkt.src
82. **if** isinstance(arp\_pkt, arp.arp):
83. self.handle\_arp(msg, in\_port, dst\_mac,src\_mac, pkt, pkt\_type)
85. **if** isinstance(ipv4\_pkt, ipv4.ipv4):
86. self.handle\_ipv4(msg, ipv4\_pkt.src, ipv4\_pkt.dst, pkt\_type)

89. **def** handle\_arp(self, msg, in\_port, dst,src, pkt, pkt\_type):
90. #just handle loop here
91. #just like your code in exp1 mission2
92. dp = msg.datapath
93. ofp = dp.ofproto
94. parser = dp.ofproto\_parser
95. dpid = dp.id
96. header\_list = dict((p.protocol\_name, p) **for** p **in** pkt.protocols **if** type(p) != str)
97. **if** dst == ETHERNET\_MULTICAST **and** ARP **in** header\_list:
98. # you need to code here to avoid broadcast loop to finish mission 2
99. dst\_ip=header\_list[ARP].dst\_ip
100. **if** (dpid,src,dst\_ip) **not** **in** self.sw:
101. self.sw[(dpid,src,dst\_ip)]=in\_port
102. **elif** self.sw[(dpid,src,dst\_ip)]!=in\_port: #drop the packet
103. **return**
104. self.mac\_to\_port[dpid][src]=in\_port
105. **if** dst **in** self.mac\_to\_port[dpid]:
106. out\_port=self.mac\_to\_port[dpid][dst]
107. **else**:
108. out\_port=ofp.OFPP\_FLOOD
109. actions=[parser.OFPActionOutput(out\_port)]
110. #install flow
111. **if** out\_port!=ofp.OFPP\_FLOOD:
112. match=parser.OFPMatch(in\_port=in\_port,eth\_dst=dst,eth\_src=src)
113. self.add\_flow(dp,1,match,actions,10,30)
114. data=msg.data
115. out=parser.OFPPacketOut(datapath=dp,buffer\_id=msg.buffer\_id,in\_port=in\_port,actions=actions,data=data)
116. dp.send\_msg(out)
118. **def** handle\_ipv4(self, msg, src\_ip, dst\_ip, pkt\_type):
119. parser = msg.datapath.ofproto\_parser
120. dpid\_path = self.network\_awareness.shortest\_path(src\_ip, dst\_ip,weight='delay')
121. **if** **not** dpid\_path:
122. **return**
123. self.path=dpid\_path
124. # get port path:  h1 -> in\_port, s1, out\_port -> h2
125. port\_path = []
126. **for** i **in** range(1, len(dpid\_path) - 1):
127. in\_port = self.network\_awareness.link\_info[(dpid\_path[i], dpid\_path[i - 1])]
128. out\_port = self.network\_awareness.link\_info[(dpid\_path[i], dpid\_path[i + 1])]
129. port\_path.append((in\_port, dpid\_path[i], out\_port))
130. self.show\_path(src\_ip, dst\_ip, port\_path)
132. # send flow mod
133. **for** node **in** port\_path:
134. in\_port, dpid, out\_port = node
135. self.send\_flow\_mod(parser, dpid, pkt\_type, src\_ip, dst\_ip, in\_port, out\_port)
136. self.send\_flow\_mod(parser, dpid, pkt\_type, dst\_ip, src\_ip, out\_port, in\_port)
138. # send packet\_out
139. \_, dpid, out\_port = port\_path[-1]
140. dp = self.network\_awareness.switch\_info[dpid]
141. actions = [parser.OFPActionOutput(out\_port)]
142. out = parser.OFPPacketOut(
143. datapath=dp, buffer\_id=msg.buffer\_id, in\_port=in\_port, actions=actions, data=msg.data)
144. dp.send\_msg(out)
146. **def** send\_flow\_mod(self, parser, dpid, pkt\_type, src\_ip, dst\_ip, in\_port, out\_port):
147. dp = self.network\_awareness.switch\_info[dpid]
148. match = parser.OFPMatch(
149. in\_port=in\_port, eth\_type=pkt\_type, ipv4\_src=src\_ip, ipv4\_dst=dst\_ip)
150. actions = [parser.OFPActionOutput(out\_port)]
151. self.add\_flow(dp, 2, match, actions, 10, 30)
153. **def** show\_path(self, src, dst, port\_path):
154. self.logger.info('path: {} -> {}'.format(src, dst))
155. path = src + ' -> '
156. **for** node **in** port\_path:
157. path += '{}:s{}:{}'.format(\*node) + ' -> '
158. path += dst
159. self.logger.info(path)

162. @set\_ev\_cls(ofp\_event.EventOFPPortStatus, MAIN\_DISPATCHER)
163. **def** port\_status\_handler(self, ev):
164. msg = ev.msg
165. dp = msg.datapath
166. ofproto = dp.ofproto
168. **if** msg.reason == ofproto.OFPPR\_ADD:
169. dp.ports[msg.desc.port\_no] = msg.desc
170. **elif** msg.reason == ofproto.OFPPR\_MODIFY:
171. dp.ports[msg.desc.port\_no] = msg.desc
172. **if** msg.desc.state == ofproto.OFPPS\_LINK\_DOWN:
173. self.dpid.append(int(msg.desc.name[1]))
174. self.del\_flow(dp)
175. **if** len(self.dpid) == 2:
176. **try**:
177. self.network\_awareness.topo\_map.remove\_edge(self.dpid[0], self.dpid[1])
178. **except**:
179. self.network\_awareness.topo\_map.remove\_edge(self.dpid[1], self.dpid[0])
180. self.network\_awareness.show\_topo\_map()
181. self.dpid = []
182. **elif** msg.desc.state == ofproto.OFPPS\_LIVE:
183. self.del\_flow(dp)
184. **elif** msg.reason == ofproto.OFPPR\_DELETE:
185. dp.ports.pop(msg.desc.port\_no, None)
186. **else**:
187. **return**
188. self.send\_event\_to\_observers(
189. ofp\_event.EventOFPPortStateChange(dp, msg.reason, msg.desc.port\_no),
190. dp.state)