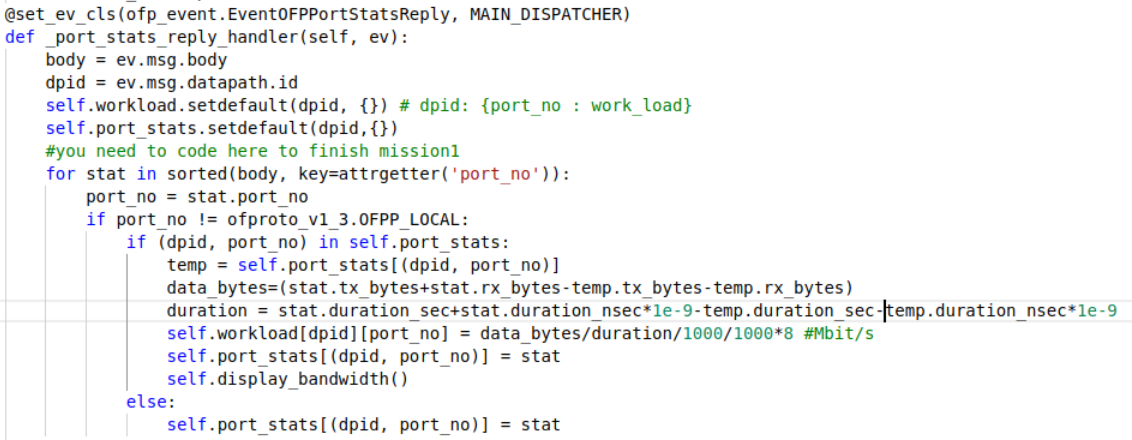
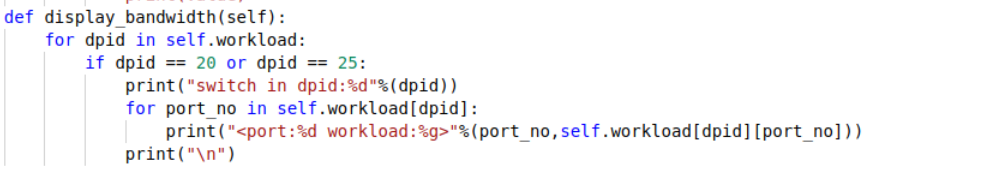
**SDN实验三**

1. 测量端口负载：

实验思路：记录下来自同一交换机的Reply报文，利用两次Reply报文中的收发总字节数之和除以时间差即可得到交换机负载，再转换成MBps时要注意除1000而不是1024，同时计算时间间隔时要注意将ns换算成s。

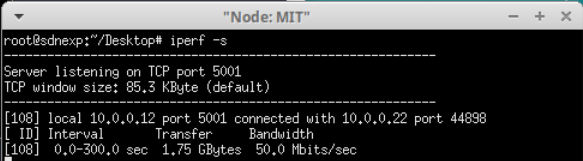


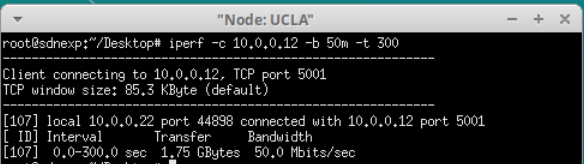
计算交换机端口负载



输出端口负载，这里为了便于观察只输出了MIT和UCLA的端口负载

实验结果：





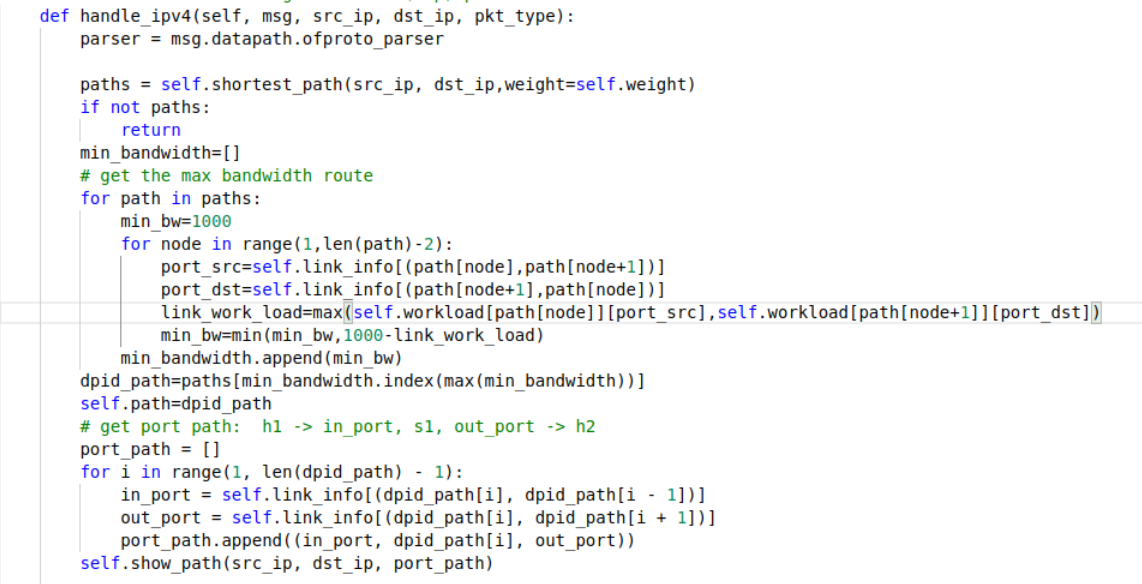


MIT和UCLA的交换机端口负载波动后稳定在50MBps左右

1. 最佳带宽路径：

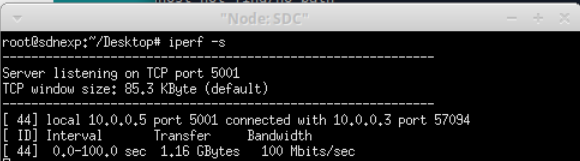
实验思路：调用shortest\_simple\_paths计算出所有路径，之后利用端口负载计算出链路可用带宽（1000 - 相连的两个交换机端口负载最大者），将paths中可用带宽最大的路径下发给交换机。

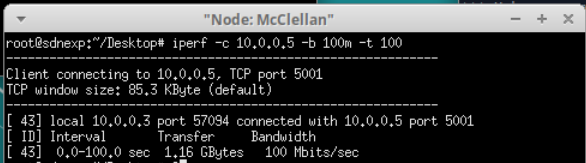
为了使同一个数据流在传输时保持同一条路径，将硬超时设置为0，即没有硬超时，软超时设置为10，当一段时间没有数据包匹配流表时就会自动删除流表。

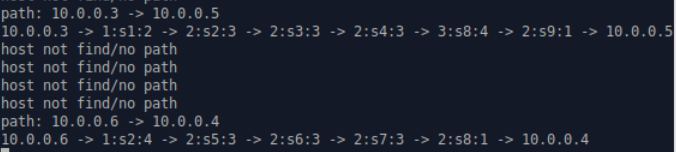


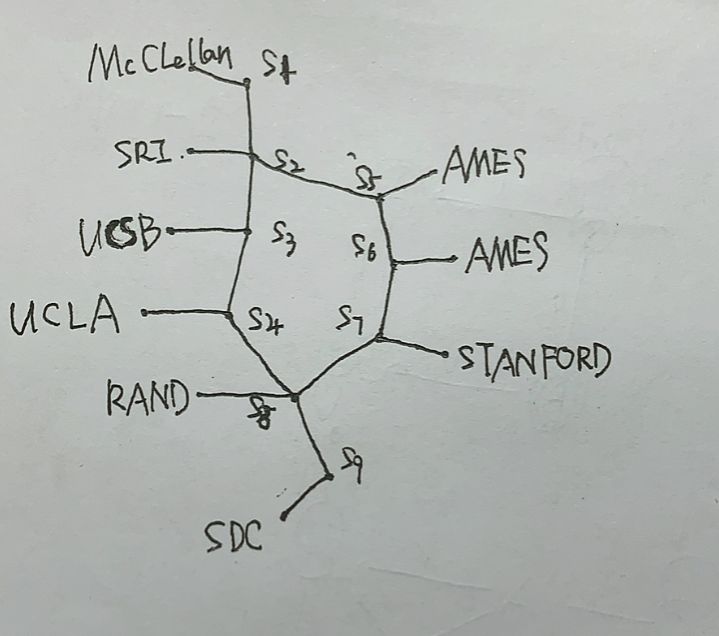
计算可用带宽最大路径

实验结果：









可以看到当SDC和McCLellan占用了[1,2,3,4,8,9]这条路径时，SRI ping RAND时控制器跟据最佳带宽原则选择了另一条路径[2,5,6,7,8]。

1. 实验代码
2. **import** networkx as nx
3. **from** operator **import** attrgetter
4. **from** ryu.base **import** app\_manager
5. **from** ryu.controller **import** ofp\_event
6. **from** ryu.controller.handler **import** MAIN\_DISPATCHER, DEAD\_DISPATCHER
7. **from** ryu.controller.handler **import** CONFIG\_DISPATCHER
8. **from** ryu.controller.handler **import** set\_ev\_cls
9. **from** ryu.ofproto **import** ofproto\_v1\_3
10. **from** ryu.lib.packet **import** packet
11. **from** ryu.lib.packet **import** ethernet
12. **from** ryu.lib.packet **import** ipv4
13. **from** ryu.lib.packet **import** arp
14. **from** ryu.lib.packet **import** tcp
15. **from** ryu.lib **import** hub
16. **from** ryu.topology **import** event, switches
17. **from** ryu.topology.api **import** get\_switch, get\_link
18. **from** ryu.lib.packet **import** ether\_types
19. **from** collections **import** defaultdict
20. **from** ryu.topology.api **import** get\_host, get\_link, get\_switch
21. ETHERNET = ethernet.ethernet.\_\_name\_\_
22. ETHERNET\_MULTICAST = "ff:ff:ff:ff:ff:ff"
23. ARP = arp.arp.\_\_name\_\_
24. **class** Workload(app\_manager.RyuApp):
26. OFP\_VERSIONS = [ofproto\_v1\_3.OFP\_VERSION]
28. **def** \_\_init\_\_(self, \*args, \*\*kwargs):
29. super(Workload, self).\_\_init\_\_(\*args, \*\*kwargs)
30. self.topology\_api\_app = self
31. self.datapaths={} #dpid: datapath
32. self.port\_stats={}# (dpid,port\_no):a list of port\_stats
33. self.link\_info = {}  # (s1, s2): s1.port
34. self.port\_link={} # s1,port:s1,s2
35. self.port\_info = {}  # dpid: (ports linked hosts)
36. self.topo\_map = nx.Graph()
37. self.workload\_thread = hub.spawn(self.\_count\_workload)
38. self.mac\_to\_port={}
39. self.sw = {} # use it to avoid arp loop
40. self.weight='hop'
41. #you need to store workload of every port here
42. self.workload={} # dpid: {port\_no : work\_load}

45. **def** \_count\_workload(self):
46. **while** True:
47. **for** dp **in** self.datapaths.values():
48. self.\_send\_request(dp)
49. self.get\_topology(None)
50. hub.sleep(4)
52. **def** \_send\_request(self,datapath):
53. ofproto = datapath.ofproto
54. parser = datapath.ofproto\_parser
55. req = parser.OFPPortStatsRequest(datapath, 0, ofproto.OFPP\_ANY)
56. datapath.send\_msg(req)
58. @set\_ev\_cls(ofp\_event.EventOFPStateChange,[MAIN\_DISPATCHER, DEAD\_DISPATCHER])
59. **def** \_state\_change\_handler(self, ev):
60. datapath = ev.datapath
61. **if** ev.state == MAIN\_DISPATCHER:
62. **if** **not** datapath.id **in** self.datapaths:
63. self.datapaths[datapath.id] = datapath
64. **elif** ev.state == DEAD\_DISPATCHER:
65. **if** datapath.id **in** self.datapaths:
66. **del** self.datapaths[datapath.id]
68. **def** add\_flow(self, dp, p, match, actions, idle\_timeout=0, hard\_timeout=0):
69. ofproto = dp.ofproto
70. parser = dp.ofproto\_parser
71. inst = [parser.OFPInstructionActions(ofproto.OFPIT\_APPLY\_ACTIONS,
72. actions)]
73. mod = parser.OFPFlowMod(datapath=dp, priority=p,
74. idle\_timeout=idle\_timeout,
75. hard\_timeout=hard\_timeout,
76. match=match, instructions=inst)
77. dp.send\_msg(mod)
79. @set\_ev\_cls(ofp\_event.EventOFPSwitchFeatures, CONFIG\_DISPATCHER)
80. **def** switch\_features\_handler(self, ev):
81. msg = ev.msg
82. dp = msg.datapath
83. ofp = dp.ofproto
84. parser = dp.ofproto\_parser
85. match = parser.OFPMatch()
86. actions = [parser.OFPActionOutput(ofp.OFPP\_CONTROLLER,ofp.OFPCML\_NO\_BUFFER)]
87. self.add\_flow(dp, 0, match, actions)
88. @set\_ev\_cls(ofp\_event.EventOFPPortStatsReply, MAIN\_DISPATCHER)
89. **def** \_port\_stats\_reply\_handler(self, ev):
90. body = ev.msg.body
91. dpid = ev.msg.datapath.id
92. self.workload.setdefault(dpid, {}) # dpid: {port\_no : work\_load}
93. self.port\_stats.setdefault(dpid,{})
94. #you need to code here to finish mission1
95. **for** stat **in** sorted(body, key=attrgetter('port\_no')):
96. port\_no = stat.port\_no
97. **if** port\_no != ofproto\_v1\_3.OFPP\_LOCAL:
98. **if** (dpid, port\_no) **in** self.port\_stats:
99. temp = self.port\_stats[(dpid, port\_no)]
100. data\_bytes=(stat.tx\_bytes+stat.rx\_bytes-temp.tx\_bytes-temp.rx\_bytes)
101. duration = stat.duration\_sec+stat.duration\_nsec\*1e-9-temp.duration\_sec-temp.duration\_nsec\*1e-9
102. self.workload[dpid][port\_no] = data\_bytes/duration/1000/1000\*8 #Mbit/s
103. self.port\_stats[(dpid, port\_no)] = stat
104. #self.display\_bandwidth()
105. **else**:
106. self.port\_stats[(dpid, port\_no)] = stat
108. #of course, you can define new function as you wish
109. '''''for stat in sorted(body, key=attrgetter('port\_no')):
110. port\_no = stat.port\_no
111. if port\_no != ofproto\_v1\_3.OFPP\_LOCAL:
112. key = (dpid, port\_no)
113. value = (stat.tx\_bytes, stat.rx\_bytes,
114. stat.duration\_sec, stat.duration\_nsec)
115. print(key)
116. print(":")
117. print(value)'''
118. **def** display\_bandwidth(self):
119. **for** dpid **in** self.workload:
120. **if** dpid == 20 **or** dpid == 25:
121. **print**("switch in dpid:%d"%(dpid))
122. **for** port\_no **in** self.workload[dpid]:
123. **print**("<port:%d workload:%g>"%(port\_no,self.workload[dpid][port\_no]))
124. **print**("\n")


128. ############################detect topology############################
129. **def** get\_topology(self, ev):
130. """
131. Gett topology info to calculate shortest paths.
132. """
133. \_hosts, \_switches, \_links = None, None, None
134. hosts = get\_host(self)
135. switches = get\_switch(self)
136. links = get\_link(self)
138. # update topo\_map when topology change
139. **if** [str(x) **for** x **in** hosts] == \_hosts **and** [str(x) **for** x **in** switches] == \_switches **and** [str(x) **for** x **in**
140. links] == \_links:
141. **return**
142. \_hosts, \_switches, \_links = [str(x) **for** x **in** hosts], [str(x) **for** x **in** switches], [str(x) **for** x **in** links]
144. **for** switch **in** switches:
145. self.port\_info.setdefault(switch.dp.id, set())
146. # record all ports
147. **for** port **in** switch.ports:
148. self.port\_info[switch.dp.id].add(port.port\_no)
150. **for** host **in** hosts:
151. # take one ipv4 address as host id
152. **if** host.ipv4:
153. self.link\_info[(host.port.dpid, host.ipv4[0])] = host.port.port\_no
154. self.topo\_map.add\_edge(host.ipv4[0], host.port.dpid, hop=1, delay=0, is\_host=True)
155. **for** link **in** links:
156. # delete ports linked switches
157. self.port\_info[link.src.dpid].discard(link.src.port\_no)
158. self.port\_info[link.dst.dpid].discard(link.dst.port\_no)
160. # s1 -> s2: s1.port, s2 -> s1: s2.port
161. self.port\_link[(link.src.dpid, link.src.port\_no)] = (link.src.dpid, link.dst.dpid)
162. self.port\_link[(link.dst.dpid, link.dst.port\_no)] = (link.dst.dpid, link.src.dpid)
164. self.link\_info[(link.src.dpid, link.dst.dpid)] = link.src.port\_no
165. self.link\_info[(link.dst.dpid, link.src.dpid)] = link.dst.port\_no
166. self.topo\_map.add\_edge(link.src.dpid, link.dst.dpid, hop=1, is\_host=False)
168. @set\_ev\_cls(ofp\_event.EventOFPPacketIn, MAIN\_DISPATCHER)
169. **def** packet\_in\_handler(self, ev):
170. msg = ev.msg
171. dp = msg.datapath
172. ofp = dp.ofproto
173. parser = dp.ofproto\_parser
175. dpid = dp.id
176. in\_port = msg.match['in\_port']
178. pkt = packet.Packet(msg.data)
179. eth\_pkt = pkt.get\_protocol(ethernet.ethernet)
180. arp\_pkt = pkt.get\_protocol(arp.arp)
181. ipv4\_pkt = pkt.get\_protocol(ipv4.ipv4)
183. pkt\_type = eth\_pkt.ethertype
184. # layer 2 self-learning
185. dst\_mac = eth\_pkt.dst
186. src\_mac = eth\_pkt.src
187. self.mac\_to\_port.setdefault(dpid, {})
188. self.sw.setdefault(dpid, {})
189. **if** isinstance(arp\_pkt, arp.arp):
190. self.handle\_arp(msg,dst\_mac,src\_mac,pkt)
192. **if** isinstance(ipv4\_pkt, ipv4.ipv4):
193. self.handle\_ipv4(msg, ipv4\_pkt.src, ipv4\_pkt.dst, pkt\_type)
195. ############################deal with loop############################
196. **def** handle\_arp(self,msg,dst,src,pkt):
197. dp = msg.datapath
198. ofp = dp.ofproto
199. parser = dp.ofproto\_parser
200. dpid = dp.id
201. in\_port = msg.match['in\_port']
202. header\_list = dict((p.protocol\_name, p) **for** p **in** pkt.protocols **if** type(p) != str)
203. **if** dst == ETHERNET\_MULTICAST **and** ARP **in** header\_list:
204. # you need to code here to avoid broadcast loop to finish mission 2
205. dst\_ip=header\_list[ARP].dst\_ip
206. **if** (dpid,src,dst\_ip) **not** **in** self.sw:
207. self.sw[(dpid,src,dst\_ip)]=in\_port
208. **elif** self.sw[(dpid,src,dst\_ip)]!=in\_port: #drop the packet
209. **return**
210. self.mac\_to\_port[dpid][src]=in\_port
211. **if** dst **in** self.mac\_to\_port[dpid]:
212. out\_port=self.mac\_to\_port[dpid][dst]
213. **else**:
214. out\_port=ofp.OFPP\_FLOOD
215. actions=[parser.OFPActionOutput(out\_port)]
216. #install flow
217. **if** out\_port!=ofp.OFPP\_FLOOD:
218. match=parser.OFPMatch(in\_port=in\_port,eth\_dst=dst,eth\_src=src)
219. self.add\_flow(dp,1,match,actions,10,0)
220. data=msg.data
221. out=parser.OFPPacketOut(datapath=dp,buffer\_id=msg.buffer\_id,in\_port=in\_port,actions=actions,data=data)
222. dp.send\_msg(out)
223. #just your code in exp1 mission2
225. ############################get shortest(hop) path############################
226. **def** handle\_ipv4(self, msg, src\_ip, dst\_ip, pkt\_type):
227. parser = msg.datapath.ofproto\_parser
229. paths = self.shortest\_path(src\_ip, dst\_ip,weight=self.weight)
230. **if** **not** paths:
231. **return**
232. min\_bandwidth=[]
233. # get the max bandwidth route
234. **for** path **in** paths:
235. min\_bw=1000
236. **for** node **in** range(1,len(path)-2):
237. port\_src=self.link\_info[(path[node],path[node+1])]
238. port\_dst=self.link\_info[(path[node+1],path[node])]
239. link\_work\_load=max(self.workload[path[node]][port\_src],self.workload[path[node+1]][port\_dst])
240. min\_bw=min(min\_bw,1000-link\_work\_load)
241. min\_bandwidth.append(min\_bw)
242. dpid\_path=paths[min\_bandwidth.index(max(min\_bandwidth))]
243. self.path=dpid\_path
244. # get port path:  h1 -> in\_port, s1, out\_port -> h2
245. port\_path = []
246. **for** i **in** range(1, len(dpid\_path) - 1):
247. in\_port = self.link\_info[(dpid\_path[i], dpid\_path[i - 1])]
248. out\_port = self.link\_info[(dpid\_path[i], dpid\_path[i + 1])]
249. port\_path.append((in\_port, dpid\_path[i], out\_port))
250. self.show\_path(src\_ip, dst\_ip, port\_path)

253. # send flow mod
254. **for** node **in** port\_path:
255. in\_port, dpid, out\_port = node
256. self.send\_flow\_mod(parser, dpid, pkt\_type, src\_ip, dst\_ip, in\_port, out\_port)
257. self.send\_flow\_mod(parser, dpid, pkt\_type, dst\_ip, src\_ip, out\_port, in\_port)
259. # send packet\_out
260. \_, dpid, out\_port = port\_path[-1]
261. dp = self.datapaths[dpid]
262. actions = [parser.OFPActionOutput(out\_port)]
263. out = parser.OFPPacketOut(
264. datapath=dp, buffer\_id=msg.buffer\_id, in\_port=in\_port, actions=actions, data=msg.data)
265. dp.send\_msg(out)
267. **def** shortest\_path(self, src, dst, weight='hop'):
268. **try**:
269. paths = list(nx.shortest\_simple\_paths(self.topo\_map, src, dst, weight=weight))
270. **return** paths
271. **except**:
272. self.logger.info('host not find/no path')
273. **def** send\_flow\_mod(self, parser, dpid, pkt\_type, src\_ip, dst\_ip, in\_port, out\_port):
274. dp = self.datapaths[dpid]
275. match = parser.OFPMatch(
276. in\_port=in\_port, eth\_type=pkt\_type, ipv4\_src=src\_ip, ipv4\_dst=dst\_ip)
277. actions = [parser.OFPActionOutput(out\_port)]
278. self.add\_flow(dp, 5, match, actions, 10, 0)
280. **def** show\_path(self, src, dst, port\_path):
281. self.logger.info('path: {} -> {}'.format(src, dst))
282. path = src + ' -> '
283. **for** node **in** port\_path:
284. path += '{}:s{}:{}'.format(\*node) + ' -> '
285. path += dst
286. self.logger.info(path)