

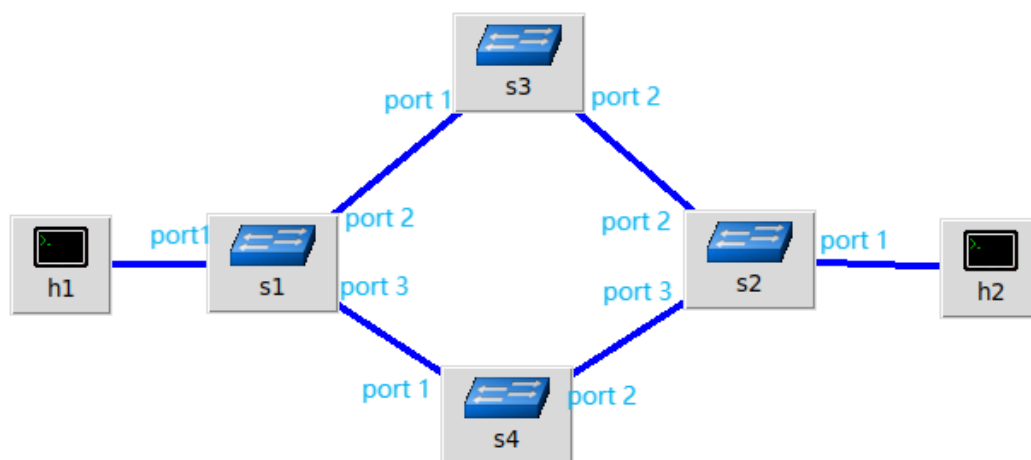
RYU Open Flow Controller

Tang Peng: 517020910038

In this lab, we learn how to use the RYU to create new network management and control applications. Specially, we are going to write a flow controller to manage the data flow into the network. By doing the following exercises, we learn more about the RYU and SDN.

Exercise 1

In this exercise, we are going to set up the following network topology with the **mininet**.



By running the code into the *code/problem1.py* file, we can construct the network. The following figure shows this network we set up.

```
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s2-eth1
s1 lo: s1-eth1:h1-eth0 s1-eth2:s3-eth1 s1-eth3:s4-eth1
s2 lo: s2-eth1:h2-eth0 s2-eth2:s3-eth2 s2-eth3:s4-eth2
s3 lo: s3-eth1:s1-eth2 s3-eth2:s2-eth2
s4 lo: s4-eth1:s1-eth3 s4-eth2:s2-eth3
```

Exercise 2

In this exercise, we are going to write a RYU controller that switches paths (h1-s1-s3-s2-h2 or h1-s1-s4-s2-h2) between h1 and h2 every 5 seconds.

Analysis

- Firstly, from the network topology, we can change the flow entries in the **s1** and **s2** to switch the two paths while keeping the flow entries in **s3** and **s4** unchanged. Specially, at the beginning, we can add a flow entry between port 1 and port 2 in **s1** such that **s1** can transmit the packets coming from port 1 to port 2. And also, we add a corresponding flow entry

between port 2 and port 1 such that **s1** can transmit the packets coming port 2 to port 1. Similarly, we can do this for **s2**, **s3**, and **s4**. Thus, we set up the path **h1-s1-s3-s2-h2** at first. Then, after 5 seconds, we can delete these flow entries in the **s1** and **s2** and add two flow entries between port 1 and port 3 for them. After doing this, we switch the path **h1-s1-s3-s2-h2** to **h1-s1-s4-s2-h2**. Repeatedly, we can do this switching every 5 seconds.

- In order to implement above function, we can do as follows. Firstly, we can use the **hard_timeout** parameter defined in the **OFPFLOWMOD** function to control the live time of the flow entry. We can set **hard_timeout** to be 5 seconds. Then, every time the flow entries we add for **s1** and **s2** will be deleted automatically after 5 seconds. Secondly, we can use table-miss flow entry to indicate the controller that the flow entries in **s1** and **s2** have been deleted and we need to add the new flow entries to them. Specially, when the flow entries in **s1** and **s2** are deleted, the packets arriving at them will be transmitted to the controller through the table-miss flow entry. Then, the controller can add the new flow entries to them.
- Here we think the 5 seconds means the time transmitting the packets, which means when there are no packets transmitted, we don't consider the time. This is to say that when there is no packet being transmitted, we don't set up the path. Only when transmitting the packets, we will switch the two paths every 5 seconds.

Code

By above analysis, we can use the following code(in *code/problem2.py*) to implement the needed function.

[illegible]

```

33         self.add_flow(datapath, 0, match, actions,0)
34
35         #add flow for s3 and s4
36         #s3
37         if datapath.id == 3:
38             kwargs = dict(in_port = 1,
eth_type=ether_types.ETH_TYPE_IP,
39                             ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
40             match = parser.OFPMatch(**kwargs)
41             actions = [parser.OFPActionOutput(2)]
42             self.add_flow(datapath,1,match,actions,0)
43
44             kwargs = dict(in_port = 2,
eth_type=ether_types.ETH_TYPE_IP,
45                             ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
46             match = parser.OFPMatch(**kwargs)
47             actions = [parser.OFPActionOutput(1)]
48             self.add_flow(datapath,1,match,actions,0)
49
50         #s4
51         if datapath.id == 4:
52             kwargs = dict(in_port = 1,
eth_type=ether_types.ETH_TYPE_IP,
53                             ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
54             match = parser.OFPMatch(**kwargs)
55             actions = [parser.OFPActionOutput(2)]
56             self.add_flow(datapath,1,match,actions,0)
57
58             kwargs = dict(in_port = 2,
eth_type=ether_types.ETH_TYPE_IP,
59                             ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
60             match = parser.OFPMatch(**kwargs)
61             actions = [parser.OFPActionOutput(1)]
62             self.add_flow(datapath,1,match,actions,0)
63
64
65     def add_flow(self, datapath, priority, match, actions,timeout):
66         ofproto = datapath.ofproto
67         parser = datapath.ofproto_parser
68
69         # construct flow_mod message and send it.
70         inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
71                                             actions)]
72         mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
73                                 match=match,
instructions=inst,hard_timeout=timeout)
74         datapath.send_msg(mod)
75
76     @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
77     def _packet_in_handler(self, ev):
78         msg = ev.msg
79         datapath = msg.datapath
80         ofproto = datapath.ofproto
81         parser = datapath.ofproto_parser
82
83         # get Datapath ID to identify OpenFlow switches.
84         dpid = datapath.id
85

```

```

86         # analyse the received packets using the packet library.
87         pkt = packet.Packet(msg.data)
88         eth_pkt = pkt.get_protocol(ethernet.ethernet)
89         ipv4_pkt = pkt.get_protocol(ipv4.ipv4)
90         #arp_pkt = pkt.get_protocol(arp.arp)
91         dst = eth_pkt.dst
92         src = eth_pkt.src
93         ethtype = eth_pkt.ethertype
94
95         in_port = msg.match['in_port']
96
97         # get the received port number from packet_in message.
98
99
100         self.logger.info("packet in %s %s %s %s %s", dpid, src, dst,
in_port, ethtype)
101         if ipv4_pkt:
102             ipv4dst = ipv4_pkt.dst
103             ipv4src = ipv4_pkt.src
104             self.logger.info("packet ipv4 in %s %s %s %s %s", dpid,
ipv4src, ipv4dst, in_port, ethtype)
105
106
107         if ethtype == ether_types.ETH_TYPE_LLDP: #ignore the LLDP packet
108             return
109
110         if ethtype == ether_types.ETH_TYPE_ARP: #deal with the arp packet
111             if in_port == 1:
112                 actions = [parser.OFPActionOutput(2)]
113             if in_port == 2:
114                 actions = [parser.OFPActionOutput(1)]
115
116             # construct packet_out message and send it.
117             out = parser.OFPPacketOut(datapath=datapath,
118                                     buffer_id=ofproto.OFP_NO_BUFFER,
119                                     in_port=in_port, actions=actions,
120                                     data=msg.data)
121             datapath.send_msg(out)
122
123
124         if ethtype == ether_types.ETH_TYPE_IP: #add the flow
125             self.currenttime = int(time.time())
126             if self.lasttime!=0:
127                 interval = self.currenttime - self.lasttime
128                 self.logger.info("the time interval t= %s s", interval)
129
130
131             if self.currentpath == 1: #h1-s1-s3-s2-h2
132                 #modify to the h1-s1-s4-s2-h2
133                 #s1
134                 #s1-s4
135                 kwargs = dict(in_port = 1,
eth_type=ether_types.ETH_TYPE_IP,
136                             ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
137                 match = parser.OFPMatch(**kwargs)
138                 actions = [parser.OFPActionOutput(3)]
139                 self.add_flow(self.datapaths[1],1,match,actions,5)
140                 #s4-s1

```

```

141         kwargs = dict(in_port = 3,
eth_type=ether_types.ETH_TYPE_IP,
142             ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
143         match = parser.OFPMatch(**kwargs)
144         actions = [parser.OFPActionOutput(1)]
145         self.add_flow(self.datapaths[1],1,match,actions,5)
146
147         #s2
148         #s2-s4
149         kwargs = dict(in_port = 1,
eth_type=ether_types.ETH_TYPE_IP,
150             ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
151         match = parser.OFPMatch(**kwargs)
152         actions = [parser.OFPActionOutput(3)]
153         self.add_flow(self.datapaths[2],1,match,actions,5)
154         #s4-s1
155         kwargs = dict(in_port = 3,
eth_type=ether_types.ETH_TYPE_IP,
156             ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
157         match = parser.OFPMatch(**kwargs)
158         actions = [parser.OFPActionOutput(1)]
159         self.add_flow(self.datapaths[2],1,match,actions,5)
160
161         self.logger.info("current path:h1-s1-s4-s2-h2; last path:
h1-s1-s3-s2-h2")
162
163
164         self.lasttime = int(time.time()) # the time set up the path
165
166         #deal with current packet
167         if in_port == 2:
168             actions = [parser.OFPActionOutput(1)]
169         if in_port == 1:
170             actions = [parser.OFPActionOutput(3)]
171         out = parser.OFPPacketOut(datapath=datapath,
172                                 buffer_id=ofproto.OFP_NO_BUFFER,
173                                 in_port=in_port, actions=actions,
174                                 data=msg.data)
175         datapath.send_msg(out)
176
177         self.currentpath = 2
178
179         elif self.currentpath == 2: #h1-s1-s4-s2-h2
180             #modify to the h1-s1-s3-s2-h2
181             #s1
182             #s1-s3
183             kwargs = dict(in_port = 1,
eth_type=ether_types.ETH_TYPE_IP,
184                 ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
185             match = parser.OFPMatch(**kwargs)
186             actions = [parser.OFPActionOutput(2)]
187             self.add_flow(self.datapaths[1],1,match,actions,5)
188             #s3-s1
189             kwargs = dict(in_port = 2,
eth_type=ether_types.ETH_TYPE_IP,
190                 ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
191             match = parser.OFPMatch(**kwargs)
192             actions = [parser.OFPActionOutput(1)]

```

```

193         self.add_flow(self.datapaths[1],1,match,actions,5)
194
195         #s2
196         #s2-s3
197         kwargs = dict(in_port = 1,
eth_type=ether_types.ETH_TYPE_IP,
198             ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
199         match = parser.OFPMatch(**kwargs)
200         actions = [parser.OFPActionOutput(2)]
201         self.add_flow(self.datapaths[2],1,match,actions,5)
202         #s3-s2
203         kwargs = dict(in_port = 2,
eth_type=ether_types.ETH_TYPE_IP,
204             ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
205         match = parser.OFPMatch(**kwargs)
206         actions = [parser.OFPActionOutput(1)]
207         self.add_flow(self.datapaths[2],1,match,actions,5)
208
209         self.logger.info("current path: h1-s1-s3-s2-h2; last path:
h1-s1-s4-s2-h2")
210
211         self.lasttime = int(time.time()) # the time set up the path
212
213         #deal with current packet
214         if in_port == 3:
215             actions = [parser.OFPActionOutput(1)]
216         if in_port == 1:
217             actions = [parser.OFPActionOutput(2)]
218         out = parser.OFPPacketOut(datapath=datapath,
219             buffer_id=ofproto.OFP_NO_BUFFER,
220             in_port=in_port, actions=actions,
221             data=msg.data)
222         datapath.send_msg(out)
223
224
225         self.currentpath = 1

```

- In the `__init__()` function, we define four member variables. The **datapaths** is used to store the datapath of four switches so that we can use the datapath when add the flow entries. The **currentpath** is a flag to identify current path (h1-s1-s3-s2-h2 or h1-s1-s4-s2-h2) so that we can know which flow entry we need to add. *currentpath* = 1 means the path h1-s1-s3-s2-h2 and *currentpath* = 2 means the path h1-s1-s4-s2-h2. The **currenttime** and **lasttime** are used to compute the time between two switching so that we can confirm the this function works.
- The **switch_features_handler()** is used to install the table-miss flow entry for every switch and record the datapath of switches in the **datapaths**. What's more, we install the flow entries for **s3** and **s4** because we don't modify them later.
- The **add_flow()** is used to add the flow entry and we need to set the **hard_timeout** parameter for it. And the matching parameters we use are **in_port**, **ipv4_src**, and **ipv4_dst**.
- The **_packet_in_hand()** is used to deal with the packets coming from the switches and install flow entries for **s1** and **s2**.
 - In this function, we need to deal with arp packets specially because these packets will not be matched by the flow entry. Besides, we will ignore the LLDP packets because we don't need it.

- Every time we get a ipv4 packet, we need to add flow entries for **s1** and **s2**. If **currentpath** is 2, then we need switch to the path h1-s1-s3-s2-h2. Thus, we need add the two flow entries between port 1 and port 2 for them. Otherwise, we need add the two flow entries between port 1 and port 3 for them.
- What's more, we can print out some information when switching the paths so that we can confirm this, such as interval time between two switching.

Result

We can use above controller to manage the network in exercise 1 by using them as following command.

```
1 ~$ sudo python3 problem1.py
2 ~$ sudo ryu-manager --verbose problem2.py
```

After running them, we can see the flow entries of 4 switches. Before starting the controller, there is no flow entries. After set up the controller, these entries are installed.

```
File Edit View Search Terminal Help
171 match = parser.OFPMatch(**kwargs)
172 actions = [parser.OFPACTIONOutput(3)]
173 self.add_flow(self.datapaths[1].match.actions.5)

tp@tpljq: ~/network$ sudo ryu-manager --verbose problem2.py
Terminated
tp@tpljq: ~/network$ sudo ryu-manager --verbose problem2.py
Registered VCS backend: git
Registered VCS backend: hg
Registered VCS backend: svn
Registered VCS backend: bzt
loading app promble2.py
loading app ryu.controller.ofp_handler
instantiating app promble2.py of ExampleSwitch13
instantiating app ryu.controller.ofp_handler of OFPHandler
BRICK ExampleSwitch13
CONSUMES EventOFPPacketIn
CONSUMES EventOFPPStateChange
CONSUMES EventOFPSwitchFeatures
BRICK ofp_event
PROVIDES EventOFPPacketIn TO {'ExampleSwitch13': {'main'}}
PROVIDES EventOFPPStateChange TO {'ExampleSwitch13': {'main', 'dead'}}
PROVIDES EventOFPSwitchFeatures TO {'ExampleSwitch13': {'config'}}
CONSUMES EventOFPEchoReply
CONSUMES EventOFPEchoRequest
CONSUMES EventOFPErrormsg
CONSUMES EventOFPPHello

mininet> dpctl dump-flows
*** s1 ***
cookie=0x0, duration=13.913s, table=0, n_packets=4, n_bytes=354, priority=0 actions=CONTROLLER:65535
*** s2 ***
cookie=0x0, duration=13.911s, table=0, n_packets=5, n_bytes=424, priority=0 actions=CONTROLLER:65535
*** s3 ***
cookie=0x0, duration=13.916s, table=0, n_packets=0, n_bytes=0, priority=1, ip, in_port="s3-eth1", nw_src=10.0.0.1, nw_dst=10.0.0.2 actions=output:"s3-eth2"
cookie=0x0, duration=13.916s, table=0, n_packets=0, n_bytes=0, priority=1, ip, in_port="s3-eth2", nw_src=10.0.0.2, nw_dst=10.0.0.1 actions=output:"s3-eth1"
*** s4 ***
cookie=0x0, duration=13.916s, table=0, n_packets=0, n_bytes=0, priority=1, ip, in_port="s4-eth1", nw_src=10.0.0.1, nw_dst=10.0.0.2 actions=output:"s4-eth2"
cookie=0x0, duration=13.916s, table=0, n_packets=0, n_bytes=0, priority=1, ip, in_port="s4-eth2", nw_src=10.0.0.2, nw_dst=10.0.0.1 actions=output:"s4-eth1"
mininet>
```

Then, we can use the ping command to send packets from **h1** to **h2** as follows.

```
1 mininet> h1 ping h2
```

Then we can use the following commands to see the flow table of them.

```
1 ~$ sudo ovs-ofctl dump-flows s1
2 ~$ sudo ovs-ofctl dump-flows s2
```

Then, we can see the change of the flow table between two switching as following figure shows.


```
packet ipv4 in 2 10.0.0.1 10.0.0.2 2 2048
the time interval t= 5 s
current path: h1-s1-s4-s2-h2; last path: h1-s1-s3-s2-h2
EVENT ofp_event->ExampleSwitch13 EventOFPPacketIn
packet in 2 72:70:d6:66:65:16 4e:4f:50:42:49:5f 1 2048
packet ipv4 in 2 10.0.0.2 10.0.0.1 1 2048
the time interval t= 5 s
current path: h1-s1-s4-s2-h2; last path: h1-s1-s4-s2-h2

tp@tplqj:~$ sudo ovs-ofctl dump-flows s1
cookie=0x0, duration=1.609s, table=0, n_packets=2, n_bytes=196, hard_timeout=5, priority=1,ip,in_port="s1-eth1",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=output:"s1-eth3"
cookie=0x0, duration=302.169s, table=0, n_packets=124, n_bytes=8556, priority=0 actions=CONTROLLER:65535
tp@tplqj:~$ sudo ovs-ofctl dump-flows s2
cookie=0x0, duration=3.559s, table=0, n_packets=4, n_bytes=392, hard_timeout=5, priority=1,ip,in_port="s2-eth1",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=output:"s2-eth3"
cookie=0x0, duration=301.117s, table=0, n_packets=135, n_bytes=9494, priority=0 actions=CONTROLLER:65535
tp@tplqj:~$ sudo ovs-ofctl dump-flows s1
cookie=0x0, duration=0.734s, table=0, n_packets=1, n_bytes=98, hard_timeout=5, priority=1,ip,in_port="s1-eth2",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=output:"s1-eth1"
cookie=0x0, duration=317.324s, table=0, n_packets=127, n_bytes=8738, priority=0 actions=CONTROLLER:65535
tp@tplqj:~$ sudo ovs-ofctl dump-flows s2
cookie=0x0, duration=3.743s, table=0, n_packets=4, n_bytes=392, hard_timeout=5, priority=1,ip,in_port="s2-eth1",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=output:"s2-eth2"
cookie=0x0, duration=3.743s, table=0, n_packets=3, n_bytes=294, hard_timeout=5, priority=1,ip,in_port="s2-eth2",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=output:"s2-eth1"
cookie=0x0, duration=317.329s, table=0, n_packets=137, n_bytes=9690, priority=0 actions=CONTROLLER:65535
tp@tplqj:~$
```

What's more, we can see the time interval when transmitting data packets as following figure. There appears a 6 seconds, which may caused by converting the time to the second.

```
64 bytes from 10.0.0.2: icmp_seq=58 ttl=64 time=40.3 ms
64 bytes from 10.0.0.2: icmp_seq=59 ttl=64 time=44.2 ms
64 bytes from 10.0.0.2: icmp_seq=60 ttl=64 time=42.0 ms
64 bytes from 10.0.0.2: icmp_seq=61 ttl=64 time=41.1 ms
64 bytes from 10.0.0.2: icmp_seq=62 ttl=64 time=41.6 ms
64 bytes from 10.0.0.2: icmp_seq=63 ttl=64 time=43.1 ms
64 bytes from 10.0.0.2: icmp_seq=64 ttl=64 time=43.0 ms
64 bytes from 10.0.0.2: icmp_seq=65 ttl=64 time=42.2 ms
64 bytes from 10.0.0.2: icmp_seq=66 ttl=64 time=41.3 ms
64 bytes from 10.0.0.2: icmp_seq=67 ttl=64 time=42.9 ms
64 bytes from 10.0.0.2: icmp_seq=68 ttl=64 time=43.0 ms
64 bytes from 10.0.0.2: icmp_seq=69 ttl=64 time=42.1 ms
64 bytes from 10.0.0.2: icmp_seq=70 ttl=64 time=43.4 ms
64 bytes from 10.0.0.2: icmp_seq=71 ttl=64 time=41.6 ms
64 bytes from 10.0.0.2: icmp_seq=72 ttl=64 time=40.1 ms
64 bytes from 10.0.0.2: icmp_seq=73 ttl=64 time=43.0 ms
64 bytes from 10.0.0.2: icmp_seq=74 ttl=64 time=42.2 ms
64 bytes from 10.0.0.2: icmp_seq=75 ttl=64 time=52.6 ms
64 bytes from 10.0.0.2: icmp_seq=76 ttl=64 time=43.1 ms
64 bytes from 10.0.0.2: icmp_seq=77 ttl=64 time=43.3 ms
64 bytes from 10.0.0.2: icmp_seq=78 ttl=64 time=42.7 ms
64 bytes from 10.0.0.2: icmp_seq=79 ttl=64 time=42.6 ms
64 bytes from 10.0.0.2: icmp_seq=80 ttl=64 time=42.9 ms
64 bytes from 10.0.0.2: icmp_seq=81 ttl=64 time=43.9 ms
64 bytes from 10.0.0.2: icmp_seq=82 ttl=64 time=43.0 ms
64 bytes from 10.0.0.2: icmp_seq=83 ttl=64 time=40.9 ms

EVENT ofp_event->ExampleSwitch13 EventOFPPacketIn
packet in 2 72:70:d6:66:65:16 4e:4f:50:42:49:5f 1 2048
packet ipv4 in 2 10.0.0.2 10.0.0.1 1 2048
the time interval t= 5 s
current path: h1-s1-s3-s2-h2; last path: h1-s1-s4-s2-h2
EVENT ofp_event->ExampleSwitch13 EventOFPPacketIn
packet in 1 72:70:d6:66:65:16 4e:4f:50:42:49:5f 2 2048
packet ipv4 in 1 10.0.0.2 10.0.0.1 2 2048
the time interval t= 5 s
current path: h1-s1-s4-s2-h2; last path: h1-s1-s3-s2-h2
EVENT ofp_event->ExampleSwitch13 EventOFPPacketIn
packet in 1 4e:4f:50:42:49:5f 72:70:d6:66:65:16 1 2048
packet ipv4 in 1 10.0.0.1 10.0.0.2 1 2048
the time interval t= 6 s
current path: h1-s1-s3-s2-h2; last path: h1-s1-s4-s2-h2
EVENT ofp_event->ExampleSwitch13 EventOFPPacketIn
packet in 2 72:70:d6:66:65:16 4e:4f:50:42:49:5f 1 2048
packet ipv4 in 2 10.0.0.2 10.0.0.1 1 2048
the time interval t= 5 s
current path: h1-s1-s4-s2-h2; last path: h1-s1-s3-s2-h2
EVENT ofp_event->ExampleSwitch13 EventOFPPacketIn
packet in 1 72:70:d6:66:65:16 4e:4f:50:42:49:5f 3 2048
packet ipv4 in 1 10.0.0.2 10.0.0.1 3 2048
the time interval t= 5 s
current path: h1-s1-s3-s2-h2; last path: h1-s1-s4-s2-h2
```

Exercise 3

In this exercise, we are going to write a RYU controller that uses both paths to forward packets from h1 to h2.

Analysis

In order to use both paths, we need to use the group entry such that when we match a packet, we can use multiple actions to forward it. Specially, In our network, we can add a group entry to **s1** such that it can transmit the packets coming from port 1 to port 2 or port 3. Similarly, we can add a group entry to **s2**. Thus, we need two actions in the group entry and the weights of them are equal such that we can transmit the packets by two paths with same rate. What's more, the group type we choose is **SELECT** because we just choose one action every time.

Code

By above analysis, we can implement the code(in *code/problem3.py*) as follows.

```
1
2 from ryu.base import app_manager
3 from ryu.controller import ofp_event
4 from ryu.controller.handler import CONFIG_DISPATCHER,
  MAIN_DISPATCHER, DEAD_DISPATCHER
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, ipv4, arp, ether_types
9 import time
10 from ryu.lib.packet import in_proto as inet
```



```

11
12
13 class ExampleSwitch13(app_manager.RyuApp):
14     OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
15
16     def __init__(self, *args, **kwargs):
17         super(ExampleSwitch13, self).__init__(*args, **kwargs)
18         self.datapaths = {}
19
20     @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
21     def switch_features_handler(self, ev):
22         datapath = ev.msg.datapath
23         ofproto = datapath.ofproto
24         parser = datapath.ofproto_parser
25
26
27         self.datapaths[datapath.id] = datapath
28
29         # install the table-miss flow entry.
30         match = parser.OFPMatch()
31         actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
32                                           ofproto.OFPCML_NO_BUFFER)]
33         self.add_flow(datapath, 0, match, actions, 0)
34
35     #add flows
36     #s1
37     if datapath.id == 1:
38         #s3-s1
39         kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
40                       ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
41         match = parser.OFPMatch(**kwargs)
42         actions = [parser.OFPActionOutput(1)]
43         self.add_flow(datapath, 1, match, actions, 0)
44
45         #s4-s1
46         kwargs = dict(in_port = 3, eth_type=ether_types.ETH_TYPE_IP,
47                       ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
48         match = parser.OFPMatch(**kwargs)
49         actions = [parser.OFPActionOutput(1)]
50         self.add_flow(datapath, 1, match, actions, 0)
51         #group s1-s3, s1-s4
52
53         kwargs = dict(in_port = 1, eth_type=ether_types.ETH_TYPE_IP,
54                       ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
55         #kwargs = dict(in_port = 1)
56         match = parser.OFPMatch(**kwargs)
57         output1 = 2
58         output2 = 3
59         weight1 = 50
60         weight2 = 50
61         watch_port1 = 2
62         watch_group1 = ofproto.OFPG_ANY
63         watch_port2 = 3
64         watch_group2 = ofproto.OFPG_ANY
65         group_id = 2021
66
67         self.add_group(datapath, match, output1, output2, weight1, weight2, watch_port
68                        1, watch_port2, watch_group1, watch_group2, group_id)

```

```

67
68     #s2
69     if datapath.id == 2:
70         #s3-s2
71         kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
72                       ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
73         match = parser.OFPMatch(**kwargs)
74         actions = [parser.OFPACTIONOutput(1)]
75         self.add_flow(datapath,1,match,actions,0)
76
77         #s4-s2
78         kwargs = dict(in_port = 3, eth_type=ether_types.ETH_TYPE_IP,
79                       ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
80         match = parser.OFPMatch(**kwargs)
81         actions = [parser.OFPACTIONOutput(1)]
82         self.add_flow(datapath,1,match,actions,0)
83
84         #group s2-s3,s2-s4
85         kwargs = dict(in_port = 1, eth_type=ether_types.ETH_TYPE_IP,
86                       ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
87         match = parser.OFPMatch(**kwargs)
88         output1 = 2
89         output2 = 3
90         weight1 = 50
91         weight2 = 50
92         watch_port1 = 2
93         watch_group1 = ofproto.OFPG_ANY
94         watch_port2 = 3
95         watch_group2 = ofproto.OFPG_ANY
96         group_id = 2022
97
98         self.add_group(datapath,match,output1,output2,weight1,weight2,watch_port
99                        1,watch_port2,watch_group1,watch_group2,group_id)
100
101
102     #s3
103     if datapath.id == 3:
104         kwargs = dict(in_port = 1, eth_type=ether_types.ETH_TYPE_IP,
105                       ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
106         match = parser.OFPMatch(**kwargs)
107         actions = [parser.OFPACTIONOutput(2)]
108         self.add_flow(datapath,1,match,actions,0)
109
110         kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
111                       ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
112         match = parser.OFPMatch(**kwargs)
113         actions = [parser.OFPACTIONOutput(1)]
114         self.add_flow(datapath,1,match,actions,0)
115
116     #s4
117     if datapath.id == 4:
118         kwargs = dict(in_port = 1, eth_type=ether_types.ETH_TYPE_IP,
119                       ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
120         match = parser.OFPMatch(**kwargs)
121         actions = [parser.OFPACTIONOutput(2)]
122         self.add_flow(datapath,1,match,actions,0)

```

```

123         kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
124                        ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
125         match = parser.OFPMatch(**kwargs)
126         actions = [parser.OFPActionOutput(1)]
127         self.add_flow(datapath,1,match,actions,0)
128
129
130
131
132     def
133     add_group(self,datapath,match,output1,output2,weight1,weight2,watch_port1
134     ,watch_port2,watch_group1,watch_group2,group_id):
135         ofproto = datapath.ofproto
136         parser = datapath.ofproto_parser
137         actions1 = [parser.OFPActionOutput(output1)]
138         actions2 = [parser.OFPActionOutput(output2)]
139         buckets = [parser.OFPBucket(weight1, watch_port1, watch_group1,
140                                     actions1),
141                   parser.OFPBucket(weight2, watch_port2, watch_group2,
142                                     actions2)]
143
144         req = parser.OFPGroupMod(datapath, ofproto.OFPGC_ADD,
145                                ofproto.OFPGT_SELECT, group_id, buckets)
146         datapath.send_msg(req)
147         actions = [parser.OFPActionGroup(group_id=group_id)]
148         inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
149                                             actions)]
150         mod = parser.OFPFlowMod(datapath=datapath, priority=1,
151                                match=match, instructions=inst)
152         datapath.send_msg(mod)
153
154
155     def add_flow(self, datapath, priority, match, actions,timeout):
156         ofproto = datapath.ofproto
157         parser = datapath.ofproto_parser
158
159         # construct flow_mod message and send it.
160         inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
161                                             actions)]
162         mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
163                                match=match,
164                                instructions=inst,hard_timeout=timeout)
165         datapath.send_msg(mod)
166
167     @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
168     def _packet_in_handler(self, ev):
169         msg = ev.msg
170         datapath = msg.datapath
171         ofproto = datapath.ofproto
172         parser = datapath.ofproto_parser
173
174         # get Datapath ID to identify OpenFlow switches.
175         dpid = datapath.id
176
177         # analyse the received packets using the packet library.
178         pkt = packet.Packet(msg.data)

```

```

178     eth_pkt = pkt.get_protocol(ethernet.ethernet)
179     ipv4_pkt = pkt.get_protocol(ipv4.ipv4)
180     #arp_pkt = pkt.get_protocol(arp.arp)
181     dst = eth_pkt.dst
182     src = eth_pkt.src
183     ethtype = eth_pkt.ethertype
184
185     in_port = msg.match['in_port']
186
187     # get the received port number from packet_in message.
188
189
190     self.logger.info("packet in %s %s %s %s %s", dpid, src, dst,
in_port, ethtype)
191     if ipv4_pkt:
192         ipv4dst = ipv4_pkt.dst
193         ipv4src = ipv4_pkt.src
194         self.logger.info("packet ipv4 in %s %s %s %s %s", dpid,
ipv4src, ipv4dst, in_port, ethtype)
195         if ethtype == ether_types.ETH_TYPE_LLDP: #ignore the LLDP packet
196             return
197
198         if ethtype == ether_types.ETH_TYPE_ARP or ethtype ==
ether_types.ETH_TYPE_IP: #deal with the arp packet
199             if in_port == 1:
200                 actions = [parser.OFPActionOutput(ofproto.OFPP_FLOOD)]
201             if in_port == 2 or in_port==3:
202                 actions = [parser.OFPActionOutput(1)]
203
204
205             # construct packet_out message and send it.
206             out = parser.OFPPacketOut(datapath=datapath,
207                                     buffer_id=ofproto.OFP_NO_BUFFER,
208                                     in_port=in_port, actions=actions,
209                                     data=msg.data)
210             datapath.send_msg(out)

```

- In the `__init__()` function, we define the **datapaths** as exercise 2.
- In the **switch_features_handler()** function, we add all flow entries for all switches because we won't modify them.
 - For **s1**, we add two flow entries and one group entry. One flow entry is used to forward the packets coming from port 2 to port 1. Similarly, the other flow entry is used to forward the packets coming from port 3 to port 1. Then group entry is used to forward packets coming from port 1 to port 2 or port 3 randomly. Here the weights of both port 2 and port 3 are 50 so that the two ports will be selected at the same rate.
 - For **s2**, we add two flow entries and one group entry. They are same with **s1**'s.
 - For **s3** and **s4**, we add two flow entries between their two ports same as the exercise 2.
- The **add_group()** function is used to add the group entry to the flow table. Here the group entry type we use is **SELECT**.
- The **_packet_in_handler()** is only used to deal with the arp packets.

Result

We can run above controller application and the network as following commands.

```
1 ~$ sudo python3 problem1.py
2 ~$ sudo ryu-manager --verbose problem3.py
```

After running above commands, we can see the flow table of switches. The following figure shows that.

```
mininet> dpctl dump-flows
*** s1 ***
cookie=0x0, duration=25.077s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s1-eth2",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=output:"s1-eth1"
cookie=0x0, duration=25.076s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s1-eth3",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=output:"s1-eth1"
cookie=0x0, duration=25.076s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s1-eth1",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=group:2021
cookie=0x0, duration=25.077s, table=0, n_packets=15, n_bytes=1495, priority=0 actions=CONTROLLER:65535
*** s2 ***
cookie=0x0, duration=25.084s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s2-eth2",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=output:"s2-eth1"
cookie=0x0, duration=25.084s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s2-eth3",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=output:"s2-eth1"
cookie=0x0, duration=25.084s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s2-eth1",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=group:2022
cookie=0x0, duration=25.085s, table=0, n_packets=15, n_bytes=1532, priority=0 actions=CONTROLLER:65535
*** s3 ***
cookie=0x0, duration=25.080s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s3-eth1",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=output:"s3-eth2"
cookie=0x0, duration=25.080s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s3-eth2",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=output:"s3-eth1"
cookie=0x0, duration=25.080s, table=0, n_packets=12, n_bytes=1285, priority=0 actions=CONTROLLER:65535
*** s4 ***
cookie=0x0, duration=25.083s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s4-eth1",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=output:"s4-eth2"
cookie=0x0, duration=25.083s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s4-eth2",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=output:"s4-eth1"
cookie=0x0, duration=25.087s, table=0, n_packets=13, n_bytes=1392, priority=0 actions=CONTROLLER:65535
mininet>
```

From above figure, we can see the group entries in the flow tables of **s1** and **s2**. What's more, we can see the group detailedly by running the following commands.

```
1 ~$ sudo ovs-ofctl dump-groups s1 -O OpenFlow13
2 ~$ sudo ovs-ofctl dump-groups s2 -O OpenFlow13
```

```
tp@tpljqj:~$ sudo ovs-ofctl dump-groups s1 -O OpenFlow13
[sudo] password for tp:
OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
  group_id=2021,type=select,bucket=weight:50,watch_port:"s1-eth2",actions=output:"s1-eth2",
  bucket=weight:50,watch_port:"s1-eth3",actions=output:"s1-eth3"
tp@tpljqj:~$ sudo ovs-ofctl dump-groups s2 -O OpenFlow13
OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
  group_id=2022,type=select,bucket=weight:50,watch_port:"s2-eth2",actions=output:"s2-eth2",
  bucket=weight:50,watch_port:"s2-eth3",actions=output:"s2-eth3"
tp@tpljqj:~$
```

From above figure, we can see there are two buckets in each group and they are what we want.

Then we can test this network by using **h1** ping **h2**. We get following figure.

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data:
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=90.1 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=41.2 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=41.1 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=41.1 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=40.9 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=40.9 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=40.5 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=41.1 ms
^C
--- 10.0.0.2 ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 700ms
rtt min/avg/max/mdev = 40.586/47.164/90.187/16.263 ms
mininet> dpctl dump-flows
*** s1 ***
cookie=0x0, duration=38.134s, table=0, n_packets=8, n_bytes=784, priority=1,ip,in_port="s1-eth2",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=output:"s1-eth1"
cookie=0x0, duration=38.134s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s1-eth3",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=output:"s1-eth1"
cookie=0x0, duration=38.134s, table=0, n_packets=8, n_bytes=784, priority=1,ip,in_port="s1-eth1",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=group:2021
cookie=0x0, duration=38.134s, table=0, n_packets=14, n_bytes=802, priority=0 actions=CONTROLLER:65535
*** s2 ***
cookie=0x0, duration=38.134s, table=0, n_packets=8, n_bytes=784, priority=1,ip,in_port="s2-eth2",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=output:"s2-eth1"
cookie=0x0, duration=38.134s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s2-eth3",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=output:"s2-eth1"
cookie=0x0, duration=38.134s, table=0, n_packets=8, n_bytes=784, priority=1,ip,in_port="s2-eth1",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=group:2022
cookie=0x0, duration=38.134s, table=0, n_packets=14, n_bytes=802, priority=0 actions=CONTROLLER:65535
*** s3 ***
cookie=0x0, duration=38.138s, table=0, n_packets=8, n_bytes=784, priority=1,ip,in_port="s3-eth1",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=output:"s3-eth2"
cookie=0x0, duration=38.138s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s3-eth2",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=output:"s3-eth1"
cookie=0x0, duration=38.138s, table=0, n_packets=8, n_bytes=784, priority=1,ip,in_port="s3-eth1",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=output:"s3-eth2"
n_bytes=606, priority=0 actions=CONTROLLER:65535
*** s4 ***
cookie=0x0, duration=38.140s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s4-eth1",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=output:"s4-eth2"
cookie=0x0, duration=38.140s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s4-eth2",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=output:"s4-eth1"
cookie=0x0, duration=38.142s, table=0, n_packets=11, n_bytes=676, priority=0 actions=CONTROLLER:65535
```

From above figure, we can see that there is no packets going through **s4**, which means all packets coming from port 1 in **s1** are forwarded to port 3. Thus, the group entry always chooses one bucket, not both with same rate. After referring to the **openflow-spec-v1.3.0.pdf**, this may be caused by the switch-computed selection algorithm. It may only select one bucket when all

buckets are live. To confirm this, we break down the link between **s1** and **s3**, also the link between **s2** and **s3** by following commands.

```
1 mininet> link s2 s3 down
2 mininet> link s1 s3 down
```

Then, we use **h1** ping **h2** again, we can get the following result. From the following figure, we can see the packets are transmitted by **s4**.

```

mininet> link s1 s3 down
mininet> link s2 s3 down
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=41.3 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=42.2 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=41.0 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=41.7 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=41.9 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=41.3 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=42.2 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=41.1 ms
^C
--- 10.0.0.2 ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 701ms
rtt min/avg/max/mdev = 41.070/41.641/42.218/0.434 ms
mininet> dpctl dump-flows
*** s1 ***
cookie=0x0, duration=116.845s, table=0, n_packets=8, n_bytes=784, priority=1,ip,in_port="s1-eth2",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=output:"s1-eth1"
cookie=0x0, duration=116.845s, table=0, n_packets=8, n_bytes=784, priority=1,ip,in_port="s1-eth3",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=output:"s1-eth1"
cookie=0x0, duration=116.845s, table=0, n_packets=16, n_bytes=1568, priority=1,ip,in_port="s1-eth1",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=group:2021
cookie=0x0, duration=116.845s, table=0, n_packets=26, n_bytes=1641, priority=0 actions=CONTROLLER:65535
*** s2 ***
cookie=0x0, duration=116.845s, table=0, n_packets=8, n_bytes=784, priority=1,ip,in_port="s2-eth2",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=output:"s2-eth1"
cookie=0x0, duration=116.845s, table=0, n_packets=8, n_bytes=784, priority=1,ip,in_port="s2-eth3",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=output:"s2-eth1"
cookie=0x0, duration=116.845s, table=0, n_packets=16, n_bytes=1568, priority=1,ip,in_port="s2-eth1",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=group:2022
cookie=0x0, duration=116.845s, table=0, n_packets=26, n_bytes=1641, priority=0 actions=CONTROLLER:65535
*** s3 ***
cookie=0x0, duration=116.849s, table=0, n_packets=8, n_bytes=784, priority=1,ip,in_port="s3-eth1",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=output:"s3-eth2"
cookie=0x0, duration=116.849s, table=0, n_packets=8, n_bytes=784, priority=1,ip,in_port="s3-eth2",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=output:"s3-eth1"
cookie=0x0, duration=116.849s, table=0, n_packets=14, n_bytes=960, priority=0 actions=CONTROLLER:65535
*** s4 ***
cookie=0x0, duration=116.851s, table=0, n_packets=8, n_bytes=784, priority=1,ip,in_port="s4-eth1",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=output:"s4-eth2"
cookie=0x0, duration=116.851s, table=0, n_packets=8, n_bytes=784, priority=1,ip,in_port="s4-eth2",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=output:"s4-eth1"
cookie=0x0, duration=116.853s, table=0, n_packets=22, n_bytes=1482, priority=0 actions=CONTROLLER:65535
mininet>

```

What's more, we can use the following commands to force the two paths to be used to transmit the packets.

```
1 mininet> xterm h1 h2
2 root@tpljqj:~/network/lab6# iperf -s -P 10
3 root@tpljqj:~/network/lab6# iperf -c 10.0.0.2 -p 5001 -b 100M -P 10
```

- The first command is used to open the terminal of **h1** and **h2**.
- The second command is be executed in the **h2**'s terminal to let **h2** be a server and the **-P** is used to set the number of threads.
- The third command is be executed in the **h1**'s terminal to let **h2** be a client.

By doing this, we can get the following result. From the figure, we can see that the two paths are be used at the same time and the number of packets in two paths nearly equals.

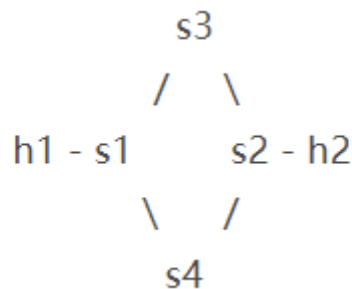
[illegible]

Exercise 4

In the exercise, we are going to write a RYU controller that uses the first path (h1-s1-s3-s2-h2) for routing packets from h1 to h2 and uses the second path for backup.

Analysis

- In order to do this, we also the group entry. However, this time the group type we use is **OFPPT_FF** rather than **OFPPT_SELECT**. The **OFPPT_FF** means choose the first live bucket and when the first bucket is not live, it will chose the second live bucket. Each action bucket is associated with a specific port and/or group that controls its liveness. Here we use the out_port of the action as the watch_port to control the bucket's liveness, which means when the out port is good, the bucket is live wile when the out port is broken, the bucket is not live. Then this bucket can not be used to forward the packets to avoid packets loss.
- What's more, we need to add this group entry for four switches because there are some special situations to deal. Consider the network again and suppose the first path is h1-s1-s3-s2-h2. Now if the link between **s1** and **s3** is down, then the **s1** will chose the second bucket in the group to forward the packets to **s4**. However, the **s2** still forwards the packets to **s3** because the first bucket in its group is still live. Then, then packets will be dropped by **s3**. In order to avoid this, we need add the group entry for **s3** such that when the first bucket is not live, **s3** will transmit the packets back to **s2** by using the second bucket rather than drop the packets directly. Thus, the **s2** need add a flow entry to forward the packets coming from **s3** to **s4**. We can do this because we will use the address as the matching, not only the in_port. We use the same method to deal with other links' breakdown.



- Thus, we can specify the flow table for four switches.
 - For **s1**, we need add 4 flow entries and one group entry. Two flow entries are used to forward the packets coming from port 2 and port 3 to port 1. The third flow entry is used to forward the packets coming from port 2 to port 3 when the link between **s2** and **s3** is broken. The final flow entry is used to forward the packets coming from port 3 to port 2 when the link between **s2** and **s4** is broken. The group entry is used to forward the packets coming port 1. The first bucket in the group is used to forward the packets to port 2 and its watch_port is also port 2. When then link between between **s1** and **s3** is broken, which means the port 2 is breakdown, then The first bucket is not live. Then then second will be used to forward the packets to port 3 and its watch_port is also port 3.
 - **s2** is same as **s1**.
 - For **s3**, we add two group entries and each of them has two buckets. One group is used to forward the packets coming port 1. And its first bucket forwards the packets to port 2 and its watch_port is also port 2. When the link between between **s2** and **s3** is broken, which means the port 2 is breakdown, then The first bucket is not live. Then then second will be used to forward the packets back to port 1 and its watch_port is also port 1. Thus, the packet will be transmitted back to the **s1**. The Other group is used to

forward the packets coming port 2 and the second bucket will be use when the link between **s1** and **s3** is broken.

- o **s4** is same as **s3**.

code

By above analysis, we can implement the code(in *code/problem4.py*) as follows.

```
1
2 from ryu.base import app_manager
3 from ryu.controller import ofp_event
4 from ryu.controller.handler import CONFIG_DISPATCHER,
  MAIN_DISPATCHER, DEAD_DISPATCHER
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, ipv4, arp, ether_types
9 import time
10 from ryu.lib.packet import in_proto as inet
11
12
13 class Exampleswitch13(app_manager.RyuApp):
14     OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
15
16     def __init__(self, *args, **kwargs):
17         super(Exampleswitch13, self).__init__(*args, **kwargs)
18         # initialize mac address table.
19         self.mac_to_port = {}
20         self.datapaths = {}
21
22     @set_ev_cls(ofp_event.EventOFPStateChange,
23               [MAIN_DISPATCHER, DEAD_DISPATCHER])
24     def _state_change_handler(self, ev): #get all switches
25         datapath = ev.datapath
26         if ev.state == MAIN_DISPATCHER:
27             if datapath.id not in self.datapaths:
28                 self.logger.debug('register datapath: %016x', datapath.id)
29                 self.datapaths[datapath.id] = datapath
30             elif ev.state == DEAD_DISPATCHER:
31                 if datapath.id in self.datapaths:
32                     self.logger.debug('unregister datapath: %016x',
datapath.id)
33                     del self.datapaths[datapath.id]
34
35
36
37     @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
38     def switch_features_handler(self, ev):
39         datapath = ev.msg.datapath
40         ofproto = datapath.ofproto
41         parser = datapath.ofproto_parser
42
43         # install the table-miss flow entry.
44         match = parser.OFPMatch()
45         actions = [parser.OFPACTIONOutput(ofproto.OFPP_CONTROLLER,
ofproto.OFPCML_NO_BUFFER)]
46
47         self.add_flow(datapath, 0, match, actions,0)
```

```

48
49     #add flows
50     #s1
51     if datapath.id == 1:
52         #s3-s1 get the packet from h2
53         kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
54                       ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
55         match = parser.OFPMatch(**kwargs)
56         actions = [parser.OFPACTIONOutput(1)]
57         self.add_flow(datapath,1,match,actions,0)
58
59         #s4-s1 get packets from h1
60         kwargs = dict(in_port = 3, eth_type=ether_types.ETH_TYPE_IP,
61                       ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
62         match = parser.OFPMatch(**kwargs)
63         actions = [parser.OFPACTIONOutput(1)]
64         self.add_flow(datapath,1,match,actions,0)
65
66
67     # add a group, the first patch is s1-s3, the second path is
s1-s4
68     kwargs = dict(in_port = 1, eth_type=ether_types.ETH_TYPE_IP,
69                   ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
70     #kwargs = dict(in_port = 1)
71     match = parser.OFPMatch(**kwargs)
72     output1 = 2
73     output2 = 3
74     weight1 = 0
75     weight2 = 0
76     watch_port1 = 2
77     watch_group1 = ofproto.OFPG_ANY
78     watch_port2 = 3
79     watch_group2 = ofproto.OFPG_ANY
80     group_id = 2021
81
82     self.add_group(datapath,match,output1,output2,weight1,weight2,watch_port
1,watch_port2,watch_group1,watch_group2,group_id)
83
84
85
86
87     #s3-s1 when link between s3 and s2 down , the packet in s3 will
forwarding the packet to the s1, so s1 need transmit the packet to the s4
88     # s1 forwarding the packet to the port 3(s4)
89
90     kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
91                   ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
92     match = parser.OFPMatch(**kwargs)
93     actions = [parser.OFPACTIONOutput(3)]
94     self.add_flow(datapath,1,match,actions,0)
95
96     # when link between s4 and s2 down , the packet in s4 will
forwarding the packet to the s1, so s1 need transmit the packet to the s3
97     # s1 forwarding the packet to the port 2(s3)
98     kwargs = dict(in_port = 3, eth_type=ether_types.ETH_TYPE_IP,
99                   ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
100    match = parser.OFPMatch(**kwargs)

```

```

101         actions = [parser.OFPActionOutput(2)]
102         self.add_flow(datapath,1,match,actions,0)
103
104     #s2
105     if datapath.id == 2:
106         #s3-s2
107         kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
108                       ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
109         match = parser.OFPMatch(**kwargs)
110         actions = [parser.OFPActionOutput(1)]
111         self.add_flow(datapath,1,match,actions,0)
112
113
114         #s4-s2
115         kwargs = dict(in_port = 3, eth_type=ether_types.ETH_TYPE_IP,
116                       ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
117         match = parser.OFPMatch(**kwargs)
118         actions = [parser.OFPActionOutput(1)]
119         self.add_flow(datapath,1,match,actions,0)
120
121         #group s2-s3,s2-s4
122         kwargs = dict(in_port = 1, eth_type=ether_types.ETH_TYPE_IP,
123                       ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
124         match = parser.OFPMatch(**kwargs)
125         output1 = 2
126         output2 = 3
127         weight1 = 0
128         weight2 = 0
129         watch_port1 = 2
130         watch_group1 = ofproto.OFPG_ANY
131         watch_port2 = 3
132         watch_group2 = ofproto.OFPG_ANY
133         group_id = 2022
134
135         self.add_group(datapath,match,output1,output2,weight1,weight2,watch_port
136         1,watch_port2,watch_group1,watch_group2,group_id)
137
138     #when the link down between s1 and s3
139     kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
140                   ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
141     match = parser.OFPMatch(**kwargs)
142     actions = [parser.OFPActionOutput(3)]
143     self.add_flow(datapath,1,match,actions,0)
144
145     #when the link down between s1 and s4
146     kwargs = dict(in_port = 3, eth_type=ether_types.ETH_TYPE_IP,
147                   ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
148     match = parser.OFPMatch(**kwargs)
149     actions = [parser.OFPActionOutput(2)]
150     self.add_flow(datapath,1,match,actions,0)
151
152     #s3
153     if datapath.id == 3:
154
155         #group when get the packet from in_port 1, it will forwarding
156         it to the port 2 firstly

```

```

155         # if the link between s2 and s3 down(port 2), forwarding the
packet back to the s1, the s1 will forwarding it to the s4
156
157         kwargs = dict(in_port = 1, eth_type=ether_types.ETH_TYPE_IP,
158                       ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
159         #kwargs = dict(in_port = 1)
160         match = parser.OFPMatch(**kwargs)
161         output1 = 2
162         output2 = ofproto.OFPP_IN_PORT # when port 1 down, forwarding
to the port2
163         weight1 = 0
164         weight2 = 0
165         watch_port1 = 2
166         watch_group1 = ofproto.OFPG_ANY
167         watch_port2 = 1
168         watch_group2 = ofproto.OFPG_ANY
169         group_id = 2023
170
171         self.add_group(datapath,match,output1,output2,weight1,weight2,watch_port
172         1,watch_port2,watch_group1,watch_group2,group_id)
173
174         #group when get the packet from in_port 2, if the link between
s1 and s3 down, forwarding the packet to the s2
175         kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
176                       ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
177         #kwargs = dict(in_port = 1)
178         match = parser.OFPMatch(**kwargs)
179         output1 = 1
180         output2 = ofproto.OFPP_IN_PORT # when port 1 down, forwarding
to the port2
181         weight1 = 0
182         weight2 = 0
183         watch_port1 = 1
184         watch_group1 = ofproto.OFPG_ANY
185         watch_port2 = 2
186         watch_group2 = ofproto.OFPG_ANY
187         group_id = 2024
188
189         self.add_group(datapath,match,output1,output2,weight1,weight2,watch_port
190         1,watch_port2,watch_group1,watch_group2,group_id)
191
192
193         #s4
194         if datapath.id == 4:
195
196
197         #group when get the packet from in_port 1, if the link between
s2 and s3 down, forwarding the packet to the s1
198
199         kwargs = dict(in_port = 1, eth_type=ether_types.ETH_TYPE_IP,
200                       ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
201         #kwargs = dict(in_port = 1)
202         match = parser.OFPMatch(**kwargs)
203         output1=2

```

```

204         output2 = ofproto.OFPP_IN_PORT# when port 2 down, forwarding
to the port1
205         weight1 = 0
206         weight2 = 0
207         watch_port1 = 2
208         watch_group1 = ofproto.OFPG_ANY
209         watch_port2 = 1
210         watch_group2 = ofproto.OFPG_ANY
211
212         group_id = 2025
213
214
215         self.add_group(datapath,match,output1,output2,weight1,weight2,watch_port
216 1,watch_port2,watch_group1,watch_group2,group_id)
217
218         #group when get the packet from in_port 2, if the link between
219 s1 and s3 down, forwarding the packet to the s2
220         kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
221             ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
222         #kwargs = dict(in_port = 1)
223         match = parser.OFPMatch(**kwargs)
224         output1 = 1
225         output2 = ofproto.OFPP_IN_PORT
226         weight1 = 0
227         weight2 = 0
228         watch_port1 = 1
229         watch_group1 = ofproto.OFPG_ANY
230         watch_port2 = 2
231         watch_group2 = ofproto.OFPG_ANY
232         group_id = 2026
233
234         self.add_group(datapath,match,output1,output2,weight1,weight2,watch_port
235 1,watch_port2,watch_group1,watch_group2,group_id)
236
237     def
238 add_group(self,datapath,match,output1,output2,weight1,weight2,watch_port1
239 ,watch_port2,watch_group1,watch_group2,group_id):
240         ofproto = datapath.ofproto
241         parser = datapath.ofproto_parser
242         actions1 = [parser.OFPActionOutput(output1)]
243         actions2 = [parser.OFPActionOutput(output2)]
244         buckets = [parser.OFPBucket(weight1, watch_port1, watch_group1,
245             actions1),
246             parser.OFPBucket(weight2, watch_port2, watch_group2,
247                 actions2)]
248
249         req = parser.OFPGGroupMod(datapath, ofproto.OFPGC_ADD,
250             ofproto.OFPGT_FF, group_id, buckets)
251         datapath.send_msg(req)
252         actions = [parser.OFPActionGroup(group_id=group_id)]
253         inst = [parser.OFPIInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
254             actions)]
255         mod = parser.OFPFlowMod(datapath=datapath, priority=1,
256             match=match, instructions=inst)

```

```

254         datapath.send_msg(mod)
255
256
257     def add_flow(self, datapath, priority, match, actions, timeout):
258         ofproto = datapath.ofproto
259         parser = datapath.ofproto_parser
260
261         # construct flow_mod message and send it.
262         inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
263                                             actions)]
264         mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
265                                 match=match,
266                                 instructions=inst, hard_timeout=timeout)
267         datapath.send_msg(mod)
268
269     @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
270     def _packet_in_handler(self, ev):
271         msg = ev.msg
272         datapath = msg.datapath
273         ofproto = datapath.ofproto
274         parser = datapath.ofproto_parser
275
276         # get Datapath ID to identify OpenFlow switches.
277         dpid = datapath.id
278         self.mac_to_port.setdefault(dpid, {})
279
280         # analyse the received packets using the packet library.
281         pkt = packet.Packet(msg.data)
282         eth_pkt = pkt.get_protocol(ethernet.ethernet)
283         ipv4_pkt = pkt.get_protocol(ipv4.ipv4)
284         #arp_pkt = pkt.get_protocol(arp.arp)
285         dst = eth_pkt.dst
286         src = eth_pkt.src
287         ethtype = eth_pkt.ethertype
288
289         in_port = msg.match['in_port']
290
291         # get the received port number from packet_in message.
292
293         self.logger.info("packet in %s %s %s %s %s", dpid, src, dst,
294                          in_port, ethtype)
295         if ipv4_pkt:
296             ipv4dst = ipv4_pkt.dst
297             ipv4src = ipv4_pkt.src
298             self.logger.info("packet ipv4 in %s %s %s %s %s", dpid,
299                              ipv4src, ipv4dst, in_port, ethtype)
300             if ethtype == ether_types.ETH_TYPE_LLDP: #ignore the LLDP packet
301                 return
302             if ethtype == ether_types.ETH_TYPE_ARP or ethtype ==
303                 ether_types.ETH_TYPE_IP: #deal with the arp packet
304                 if in_port == 1: #send to both output1 and 2
305                     actions = [parser.OFPActionOutput(ofproto.OFPP_FLOOD)]
306                 if in_port == 2 or in_port==3:
307                     actions = [parser.OFPActionOutput(1)]

```

```

308 # construct packet_out message and send it.
309 out = parser.OFPPacketOut(datapath=datapath,
310                             buffer_id=ofproto.OFP_NO_BUFFER,
311                             in_port=in_port, actions=actions,
312                             data=msg.data)
313 datapath.send_msg(out)

```

- In the **switch_features_handler()**, we add all needed flow entries and group entries to all switches. Here the weights of all buckets must be 0 because the group type is **OFPGT_FF** and we need choose the right watch_port. And because we needn't watch_group, we can set its value to be **OFPG_ANY**. And for **s3** and **s4**, if we want forward the packets to the in_port, we need use the **OFPF_IN_PORT** as the out_port.
- The **packet_in_handler()** is used to deal with the arp packets.

Result

We can run above controller application and the network as following commands.

```

1 ~$ sudo python3 problem1.py
2 ~$ sudo ryu-manager --verbose problem4.py

```

After running above commands, we can see the flow table of switches. The following figure shows that.

```

mininet> dpctl dump-flows
*** s1
cookie=0x0, duration=4.958s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s1-eth2",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=output:"s1-eth1"
cookie=0x0, duration=4.958s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s1-eth3",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=output:"s1-eth1"
cookie=0x0, duration=4.958s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s1-eth1",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions-group:2021
cookie=0x0, duration=4.958s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s1-eth2",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions-output:"s1-eth3"
cookie=0x0, duration=4.958s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s1-eth3",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions-output:"s1-eth2"
*** s2
cookie=0x0, duration=4.961s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s2-eth2",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions-output:"s2-eth1"
cookie=0x0, duration=4.961s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s2-eth3",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions-output:"s2-eth1"
cookie=0x0, duration=4.961s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s2-eth1",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions-group:2022
cookie=0x0, duration=4.961s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s2-eth2",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions-output:"s2-eth3"
cookie=0x0, duration=4.961s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s2-eth3",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions-output:"s2-eth2"
cookie=0x0, duration=4.961s, table=0, n_packets=2, n_bytes=214, priority=0 actions=CONTROLLER:65535
*** s3
cookie=0x0, duration=4.963s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s3-eth1",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions-group:2023
cookie=0x0, duration=4.963s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s3-eth2",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions-group:2024
cookie=0x0, duration=4.963s, table=0, n_packets=2, n_bytes=214, priority=0 actions=CONTROLLER:65535
*** s4
cookie=0x0, duration=4.965s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s4-eth1",nw_src=10.0.0.1,nw_dst=10.0.0.2 actions-group:2025
cookie=0x0, duration=4.965s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s4-eth2",nw_src=10.0.0.2,nw_dst=10.0.0.1 actions-group:2026
cookie=0x0, duration=4.968s, table=0, n_packets=2, n_bytes=214, priority=0 actions=CONTROLLER:65535
mininet>

tp@tpljqj:~$ sudo ovs-ofctl dump-groups s2 -O OpenFlow13
[sudo] password for tp:
OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
  group_id=2022,type=ff,bucket=watch_port:"s2-eth2",actions=output:"s2-eth2",bucket=watch_port:"s2-eth3",actions=output:"s2-eth3"
tp@tpljqj:~$ sudo ovs-ofctl dump-groups s1 -O OpenFlow13
OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
  group_id=2021,type=ff,bucket=watch_port:"s1-eth2",actions=output:"s1-eth2",bucket=watch_port:"s1-eth3",actions=output:"s1-eth3"
tp@tpljqj:~$ sudo ovs-ofctl dump-groups s3 -O OpenFlow13
OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
  group_id=2024,type=ff,bucket=watch_port:"s3-eth2",actions=output:"s3-eth2",bucket=watch_port:"s3-eth1",actions=IN_PORT
  group_id=2023,type=ff,bucket=watch_port:"s3-eth1",actions=output:"s3-eth1",bucket=watch_port:"s3-eth2",actions=IN_PORT
tp@tpljqj:~$ sudo ovs-ofctl dump-groups s4 -O OpenFlow13
OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
  group_id=2025,type=ff,bucket=watch_port:"s4-eth2",actions=output:"s4-eth2",bucket=watch_port:"s4-eth1",actions=IN_PORT
  group_id=2026,type=ff,bucket=watch_port:"s4-eth1",actions=output:"s4-eth1",bucket=watch_port:"s4-eth2",actions=IN_PORT
tp@tpljqj:~$

```

From above figure, we can see that the flow entries and the group entries are what we want.

Then we can use **h1** ping **h2** and break down some links to see the results.


```

mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=89.5 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=40.7 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=41.1 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=40.8 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=40.3 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=40.1 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=40.1 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=42.2 ms
^C
--- 10.0.0.2 ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 7009ms
rtt min/avg/max/mdev = 40.117/46.895/89.583/16.148 ms
mininet> link s1 s3 down
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=52.9 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=52.1 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=52.1 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=50.2 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=51.2 ms
^C
--- 10.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4003ms
rtt min/avg/max/mdev = 50.292/51.747/52.955/0.930 ms
mininet> link s1 s3 up
mininet> link s2 s3 down
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=51.8 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=52.0 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=51.6 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=51.2 ms
^C
--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3002ms
rtt min/avg/max/mdev = 51.232/51.714/52.078/0.386 ms
mininet>

```

From above figure, we can see that when we break some link, it still work well. Then we can break some link when the packets is been transmitting to see whether there is packet loss. We can doing this by following commands.

```

1 | mininet> h1 ping -c 50 h2 > log.txt &
2 | mininet> link s2 s3 down

```

With above commands, we can get the following result. From the following figure, we can see that when the data is been transmitting and we break the link between **s2** and **s3**, there is no packets loss. So, this application works well.

```
64 bytes from 10.0.0.2: icmp_seq=32 ttl=64 time=52.5 ms
64 bytes from 10.0.0.2: icmp_seq=33 ttl=64 time=53.7 ms
64 bytes from 10.0.0.2: icmp_seq=34 ttl=64 time=53.8 ms
64 bytes from 10.0.0.2: icmp_seq=35 ttl=64 time=51.1 ms
64 bytes from 10.0.0.2: icmp_seq=36 ttl=64 time=52.5 ms
64 bytes from 10.0.0.2: icmp_seq=37 ttl=64 time=52.7 ms
64 bytes from 10.0.0.2: icmp_seq=38 ttl=64 time=52.8 ms
64 bytes from 10.0.0.2: icmp_seq=39 ttl=64 time=51.3 ms
64 bytes from 10.0.0.2: icmp_seq=40 ttl=64 time=51.0 ms
64 bytes from 10.0.0.2: icmp_seq=41 ttl=64 time=50.8 ms
64 bytes from 10.0.0.2: icmp_seq=42 ttl=64 time=50.6 ms
64 bytes from 10.0.0.2: icmp_seq=43 ttl=64 time=51.9 ms
64 bytes from 10.0.0.2: icmp_seq=44 ttl=64 time=51.6 ms
64 bytes from 10.0.0.2: icmp_seq=45 ttl=64 time=51.6 ms
64 bytes from 10.0.0.2: icmp_seq=46 ttl=64 time=52.2 ms
64 bytes from 10.0.0.2: icmp_seq=47 ttl=64 time=52.0 ms
64 bytes from 10.0.0.2: icmp_seq=48 ttl=64 time=50.8 ms
64 bytes from 10.0.0.2: icmp_seq=49 ttl=64 time=51.0 ms
64 bytes from 10.0.0.2: icmp_seq=50 ttl=64 time=50.3 ms

--- 10.0.0.2 ping statistics ---
50 packets transmitted, 50 received, 0% packet loss, time 49072ms
rtt min/avg/max/mdev = 40.349/50.440/53.801/3.491 ms
```

```
File Edit View Search Terminal Help
--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3002ms
rtt min/avg/max/mdev = 51.232/51.714/52.078/0.386 ms
mininet> link s2 s3 up
mininet> h1 -c 50 ping h2 > log.txt &
bash: -c: command not found
mininet> h1 ping -c 50 h2 > log.txt &
mininet> link s2 s3 down
mininet>
```

With the following command, we can break the link between **s1** and **s3**, and we can get the same result.

```
1 mininet> h1 ping -c 50 h2 > log.txt &
2 mininet> link s1 s3 down
```

```
Open log1.txt [Read-Only] Save
64 bytes from 10.0.0.2: icmp_seq=25 ttl=64 time=51.5 ms
64 bytes from 10.0.0.2: icmp_seq=26 ttl=64 time=51.0 ms
64 bytes from 10.0.0.2: icmp_seq=27 ttl=64 time=52.4 ms
64 bytes from 10.0.0.2: icmp_seq=28 ttl=64 time=50.7 ms
64 bytes from 10.0.0.2: icmp_seq=29 ttl=64 time=51.3 ms
64 bytes from 10.0.0.2: icmp_seq=30 ttl=64 time=50.2 ms
64 bytes from 10.0.0.2: icmp_seq=31 ttl=64 time=50.5 ms
64 bytes from 10.0.0.2: icmp_seq=32 ttl=64 time=51.1 ms
64 bytes from 10.0.0.2: icmp_seq=33 ttl=64 time=51.3 ms
64 bytes from 10.0.0.2: icmp_seq=34 ttl=64 time=52.1 ms
64 bytes from 10.0.0.2: icmp_seq=35 ttl=64 time=51.0 ms
64 bytes from 10.0.0.2: icmp_seq=36 ttl=64 time=50.6 ms
64 bytes from 10.0.0.2: icmp_seq=37 ttl=64 time=53.1 ms
64 bytes from 10.0.0.2: icmp_seq=38 ttl=64 time=51.8 ms
64 bytes from 10.0.0.2: icmp_seq=39 ttl=64 time=51.0 ms
64 bytes from 10.0.0.2: icmp_seq=40 ttl=64 time=51.5 ms
64 bytes from 10.0.0.2: icmp_seq=41 ttl=64 time=51.6 ms
64 bytes from 10.0.0.2: icmp_seq=42 ttl=64 time=52.1 ms
64 bytes from 10.0.0.2: icmp_seq=43 ttl=64 time=52.0 ms
64 bytes from 10.0.0.2: icmp_seq=44 ttl=64 time=50.3 ms
64 bytes from 10.0.0.2: icmp_seq=45 ttl=64 time=50.6 ms
64 bytes from 10.0.0.2: icmp_seq=46 ttl=64 time=52.8 ms
64 bytes from 10.0.0.2: icmp_seq=47 ttl=64 time=50.9 ms
64 bytes from 10.0.0.2: icmp_seq=48 ttl=64 time=53.1 ms
64 bytes from 10.0.0.2: icmp_seq=49 ttl=64 time=50.7 ms
64 bytes from 10.0.0.2: icmp_seq=50 ttl=64 time=51.0 ms

--- 10.0.0.2 ping statistics ---
50 packets transmitted, 50 received, 0% packet loss, time 49064ms
rtt min/avg/max/mdev = 40.601/50.661/96.569/7.612 ms
```

```
tp@tpljqj: ~/network/
File Edit View Search Terminal Help
loss) ...(30.00Mbit 5ms delay 0.00000%
elay 0.00000% loss) (10.00Mbit 5ms dela
0Mbit 5ms delay 0.00000% loss) (10.00Mb
loss) (10.00Mbit 5ms delay 0.00000% los
y 0.00000% loss) (10.00Mbit 5ms delay 0
it 5ms delay 0.00000% loss) (10.00Mbit
s)
Dumping host connections
h1 h1-eth0:s1-eth1
h2 h2-eth0:s2-eth1
*** Starting CLI:
mininet> h1 ping -c 50 h2 >log1.txt &
mininet> link s1 s3 down
mininet>
```

Conclusion

With the lab, we learned how to define the network by the software and how to create our own controller.