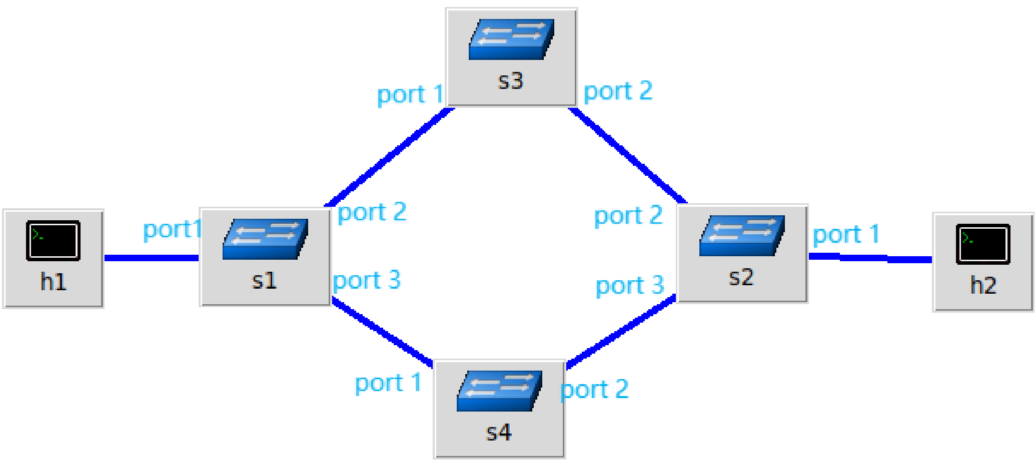
RYU Open Flow Controller

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 int the network. By doing the following exercises, we learn more about the 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 int 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 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**. Repeatly, we can do this switching every 5 seconds.

- In oder to implement above function, we can do as follows. Firstly, we can use the **hard_timeout** parameter defined in the **OFPFlowMod** function to controll 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 be 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 is no packets transmitted, we don't consider the time. This is to say that when the there is no packet been transmitting, 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.

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER,
MAIN_DISPATCHER,DEAD_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet,ipv4,arp,ether_types
import time

class ExampleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(ExampleSwitch13, self).__init__(*args, **kwargs)
        # initialize mac address table.

        self.datapaths = {}
        self.currentpath = 2
        self.currenttime = 0
        self.lasttime = 0


    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        self.datapaths[datapath.id] = datapath
        # install the table-miss flow entry.
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
```

```
        self.add_flow(datapath, 0, match, actions,0)

        #add flow for s3 and s4
        #s3
        if datapath.id == 3:
                kwargs = dict(in_port = 1, eth_type=ether_types.ETH_TYPE_IP,
                        ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
                match = parser.OFPMatch(**kwargs)
                actions = [parser.OFPActionOutput(2)]
                self.add_flow(datapath,1,match,actions,0)

                kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
                        ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
                match = parser.OFPMatch(**kwargs)
                actions = [parser.OFPActionOutput(1)]
                self.add_flow(datapath,1,match,actions,0)

        #s4
        if datapath.id == 4:
                kwargs = dict(in_port = 1, eth_type=ether_types.ETH_TYPE_IP,
                        ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
                match = parser.OFPMatch(**kwargs)
                actions = [parser.OFPActionOutput(2)]
                self.add_flow(datapath,1,match,actions,0)

                kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
                        ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
                match = parser.OFPMatch(**kwargs)
                actions = [parser.OFPActionOutput(1)]
                self.add_flow(datapath,1,match,actions,0)


    def add_flow(self, datapath, priority, match, actions,timeout):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        # construct flow_mod message and send it.
        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                             actions)]
        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                match=match, instructions=inst,hard_timeout=timeout)
        datapath.send_msg(mod)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        # get Datapath ID to identify OpenFlow switches.
        dpid = datapath.id

```

```
        # analyse the received packets using the packet library.
        pkt = packet.Packet(msg.data)
        eth_pkt = pkt.get_protocol(ethernet.ethernet)
        ipv4_pkt = pkt.get_protocol(ipv4.ipv4)
        #arp_pkt = pkt.get_protocol(arp.arp)
        dst = eth_pkt.dst
        src = eth_pkt.src
        ethtype = eth_pkt.ethertype

        in_port = msg.match['in_port']

        # get the received port number from packet_in message.


        self.logger.info("packet in %s  %s %s %s %s", dpid, src, dst, in_port,ethtype)
        if ipv4_pkt:
            ipv4dst = ipv4_pkt.dst
            ipv4src = ipv4_pkt.src
            self.logger.info("packet ipv4 in %s  %s %s %s %s", dpid, ipv4src, ipv4dst, in_port,ethtype)


        if ethtype == ether_types.ETH_TYPE_LLDP:  #ingore the LLDP packet
            return

        if ethtype == ether_types.ETH_TYPE_ARP: #deal with the arp packet
            if in_port == 1:
                actions = [parser.OFPActionOutput(2)]
            if in_port == 2:
                actions = [parser.OFPActionOutput(1)]

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


        if ethtype == ether_types.ETH_TYPE_IP: #add the flow
            self.currenttime = int(time.time())
            if self.lasttime!=0:
                interval = self.currenttime - self.lasttime
                self.logger.info("the time intervel t= %s s", interval)


            if self.currentpath == 1:  #h1-s1-s3-s2-h2
                #modify to the h1-s1-s4-s2-h2
                #s1
                #s1-s4
                kwargs = dict(in_port = 1, eth_type=ether_types.ETH_TYPE_IP,
                        ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
                match = parser.OFPMatch(**kwargs)
                actions = [parser.OFPActionOutput(3)]
                self.add_flow(self.datapaths[1],1,match,actions,5)
                #s4-s1
```

```
                kwargs = dict(in_port = 3, eth_type=ether_types.ETH_TYPE_IP,
                        ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
                match = parser.OFPMatch(**kwargs)
                actions = [parser.OFPActionOutput(1)]
                self.add_flow(self.datapaths[1],1,match,actions,5)

                #s2
                #s2-s4
                kwargs = dict(in_port = 1, eth_type=ether_types.ETH_TYPE_IP,
                        ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
                match = parser.OFPMatch(**kwargs)
                actions = [parser.OFPActionOutput(3)]
                self.add_flow(self.datapaths[2],1,match,actions,5)
                #s4-s1
                kwargs = dict(in_port = 3, eth_type=ether_types.ETH_TYPE_IP,
                        ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
                match = parser.OFPMatch(**kwargs)
                actions = [parser.OFPActionOutput(1)]
                self.add_flow(self.datapaths[2],1,match,actions,5)

                self.logger.info("current path: h1-s1-s4-s2-h2; last path: h1-s1-s3-s2-h2")


                self.lasttime = int(time.time()) # the time set up the path

                #deal with current packet
                if in_port == 2:
                    actions = [parser.OFPActionOutput(1)]
                if in_port == 1:
                    actions = [parser.OFPActionOutput(3)]
                out = parser.OFPPacketOut(datapath=datapath,
                                  buffer_id=ofproto.OFP_NO_BUFFER,
                                  in_port=in_port, actions=actions,
                                  data=msg.data)
                datapath.send_msg(out)

                self.currentpath = 2

            elif self.currentpath == 2:  #h1-s1-s4-s2-h2
                #modify to the h1-s1-s3-s2-h2
                #s1
                #s1-s3
                kwargs = dict(in_port = 1, eth_type=ether_types.ETH_TYPE_IP,
                        ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
                match = parser.OFPMatch(**kwargs)
                actions = [parser.OFPActionOutput(2)]
                self.add_flow(self.datapaths[1],1,match,actions,5)
                #s3-s1
                kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
                        ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
                match = parser.OFPMatch(**kwargs)
                actions = [parser.OFPActionOutput(1)]
```

```
                self.add_flow(self.datapaths[1],1,match,actions,5)

                #s2
                #s2-s3
                kwargs = dict(in_port = 1, eth_type=ether_types.ETH_TYPE_IP,
                        ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
                match = parser.OFPMatch(**kwargs)
                actions = [parser.OFPActionOutput(2)]
                self.add_flow(self.datapaths[2],1,match,actions,5)
                #s3-s2
                kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
                        ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
                match = parser.OFPMatch(**kwargs)
                actions = [parser.OFPActionOutput(1)]
                self.add_flow(self.datapaths[2],1,match,actions,5)

                self.logger.info("current path: h1-s1-s3-s2-h2; last path: h1-s1-s4-s2-h2")

                self.lasttime = int(time.time()) # the time set up the path

                #deal with current packet
                if in_port == 3:
                    actions = [parser.OFPActionOutput(1)]
                if in_port == 1:
                    actions = [parser.OFPActionOutput(2)]
                out = parser.OFPPacketOut(datapath=datapath,
                                  buffer_id=ofproto.OFP_NO_BUFFER,
                                  in_port=in_port, actions=actions,
                                  data=msg.data)
                datapath.send_msg(out)


                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.

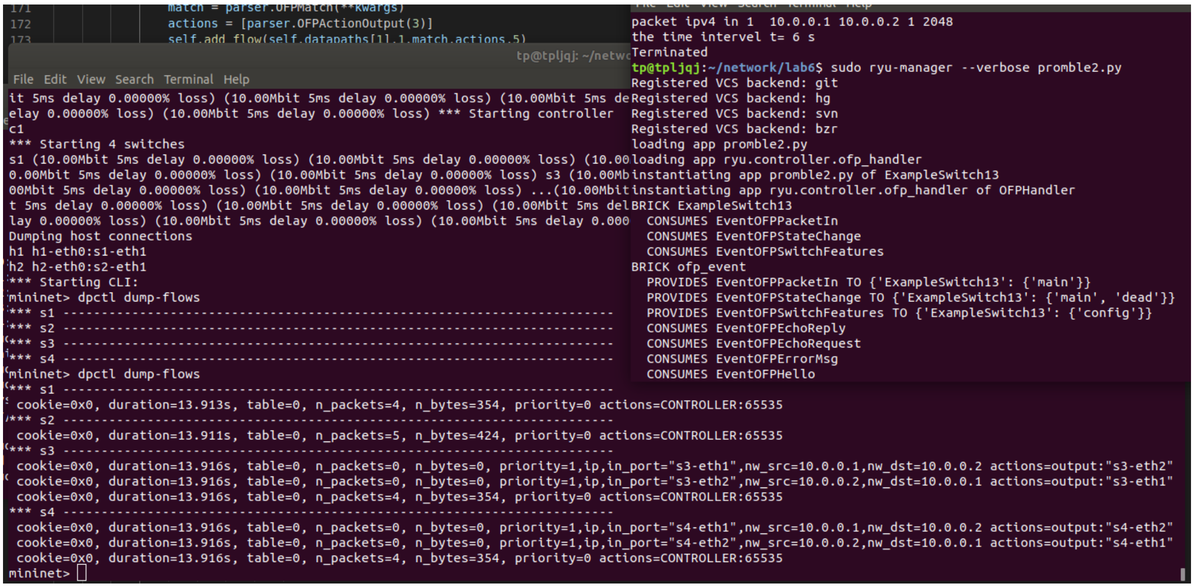
- Every time we get a ipv4 packet, we need to add flow entries for **s1** and **s2**. If **currentpath** is 2, then we need switch to the path h1-s1-s3-s2-h2. Thus, we need add the two flow entries between port 1 and port 2 for them. Otherwise, we need add the two flow entries between port 1 and port 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.

```
~$ sudo python3 problem1.py
~$ 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.



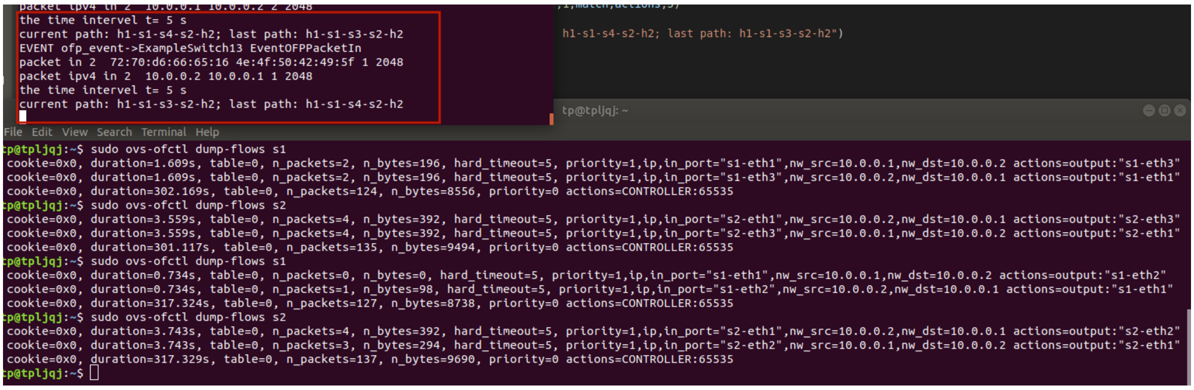
Then, we can use the ping command to send packets from **h1** to **h2** as follows.

```
mininet> h1 ping h2
```

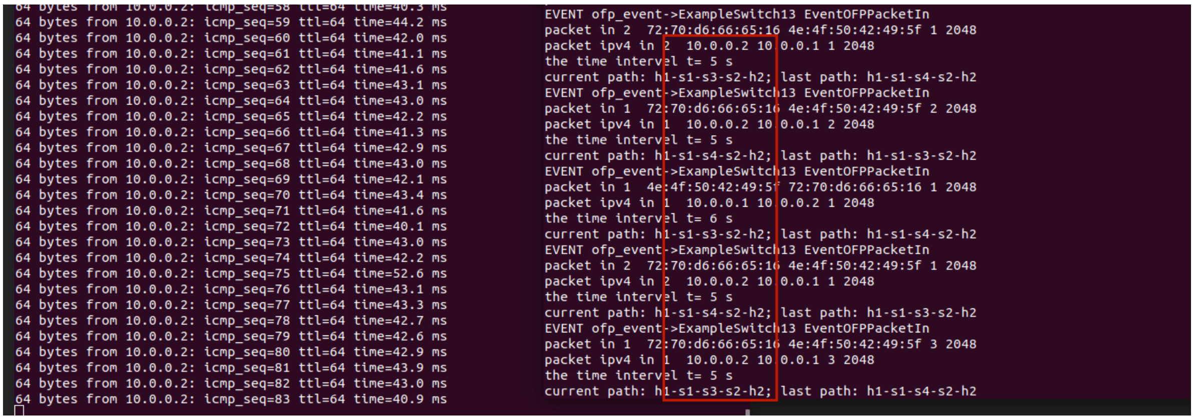
Then we can use the following commands to see the flow table of them.

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

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



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.



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.

```

from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER,
MAIN_DISPATCHER,DEAD_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet,ipv4,arp,ether_types
import time
from ryu.lib.packet import  in_proto as inet
```

```python


class ExampleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(ExampleSwitch13, self).__init__(*args, **kwargs)
        self.datapaths = {}

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser


        self.datapaths[datapath.id] = datapath

        # install the table-miss flow entry.
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions,0)

        #add flows
        #s1
        if datapath.id == 1:
            #s3-s1
            kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
                    ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            self.add_flow(datapath,1,match,actions,0)

            #s4-s1
            kwargs = dict(in_port = 3, eth_type=ether_types.ETH_TYPE_IP,
                    ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            self.add_flow(datapath,1,match,actions,0)
            #group s1-s3,s1-s4

            kwargs = dict(in_port = 1, eth_type=ether_types.ETH_TYPE_IP,
                  ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
            #kwargs = dict(in_port = 1)
            match = parser.OFPMatch(**kwargs)
            outport1 = 2
            outport2 = 3
            weight1 = 50
            weight2 = 50
            watch_port1 = 2
            watch_group1 = ofproto.OFPG_ANY
            watch_port2 = 3
            watch_group2 = ofproto.OFPG_ANY
            group_id = 2021

            self.add_group(datapath,match,outport1,outport2,weight1,weight2,watch_port1,watch_port2,watch_group1,watch_group2,group_id)
```

```
        #s2
        if datapath.id == 2:
            #s3-s2
            kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
                    ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            self.add_flow(datapath,1,match,actions,0)

            #s4-s2
            kwargs = dict(in_port = 3, eth_type=ether_types.ETH_TYPE_IP,
                    ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            self.add_flow(datapath,1,match,actions,0)

            #group s2-s3,s2-s4
            kwargs = dict(in_port = 1, eth_type=ether_types.ETH_TYPE_IP,
                    ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
            match = parser.OFPMatch(**kwargs)
            outport1 = 2
            outport2 = 3
            weight1 = 50
            weight2 = 50
            watch_port1 = 2
            watch_group1 = ofproto.OFPG_ANY
            watch_port2 = 3
            watch_group2 = ofproto.OFPG_ANY
            group_id = 2022
            self.add_group(datapath,match,outport1,outport2,weight1,weight2,watch_port1,watch_port2,watch_group1,watch_group2,group_id)




        #s3
        if datapath.id == 3:
            kwargs = dict(in_port = 1, eth_type=ether_types.ETH_TYPE_IP,
                    ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(2)]
            self.add_flow(datapath,1,match,actions,0)

            kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
                    ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            self.add_flow(datapath,1,match,actions,0)

        #s4
        if datapath.id == 4:
            kwargs = dict(in_port = 1, eth_type=ether_types.ETH_TYPE_IP,
                    ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(2)]
            self.add_flow(datapath,1,match,actions,0)
```

```python

            kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
                    ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            self.add_flow(datapath,1,match,actions,0)



    def add_group(self,datapath,match,outport1,outport2,weight1,weight2,watch_port1,watch_port2,watch_group1,watch_group2,group_id):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        actions1 = [parser.OFPActionOutput(outport1)]
        actions2 = [parser.OFPActionOutput(outport2)]
        buckets = [parser.OFPBucket(weight1, watch_port1, watch_group1,
                                        actions1),
                    parser.OFPBucket(weight2, watch_port2, watch_group2,
                                        actions2)]

        req = parser.OFPGroupMod(datapath, ofproto.OFPGC_ADD,
                                ofproto.OFPGT_SELECT, group_id, buckets)
        datapath.send_msg(req)
        actions = [parser.OFPActionGroup(group_id=group_id)]
        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                                actions)]
        mod = parser.OFPFlowMod(datapath=datapath, priority=1,
                                match=match, instructions=inst)
        datapath.send_msg(mod)




    def add_flow(self, datapath, priority, match, actions,timeout):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        # construct flow_mod message and send it.
        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                             actions)]
        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                match=match, instructions=inst,hard_timeout=timeout)
        datapath.send_msg(mod)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        # get Datapath ID to identify OpenFlow switches.
        dpid = datapath.id

        # analyse the received packets using the packet library.
        pkt = packet.Packet(msg.data)
```

```
        eth_pkt = pkt.get_protocol(ethernet.ethernet)
        ipv4_pkt = pkt.get_protocol(ipv4.ipv4)
        #arp_pkt = pkt.get_protocol(arp.arp)
        dst = eth_pkt.dst
        src = eth_pkt.src
        ethtype = eth_pkt.ethertype

        in_port = msg.match['in_port']

        # get the received port number from packet_in message.


        self.logger.info("packet in %s  %s %s %s %s", dpid, src, dst, in_port,ethtype)
        if ipv4_pkt:
            ipv4dst = ipv4_pkt.dst
            ipv4src = ipv4_pkt.src
            self.logger.info("packet ipv4 in %s  %s %s %s %s", dpid, ipv4src, ipv4dst, in_port,ethtype)
        if ethtype == ether_types.ETH_TYPE_LLDP:  #ingore the LLDP packet
            return

        if ethtype == ether_types.ETH_TYPE_ARP or ethtype == ether_types.ETH_TYPE_IP: #deal with the arp packet
            if in_port == 1:
                actions = [parser.OFPActionOutput(ofproto.OFPP_FLOOD)]
            if in_port == 2 or in_port==3:
                actions = [parser.OFPActionOutput(1)]


            # construct packet_out message and send it.
            out = parser.OFPPacketOut(datapath=datapath,
                                      buffer_id=ofproto.OFP_NO_BUFFER,
                                      in_port=in_port, actions=actions,
                                      data=msg.data)
            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.

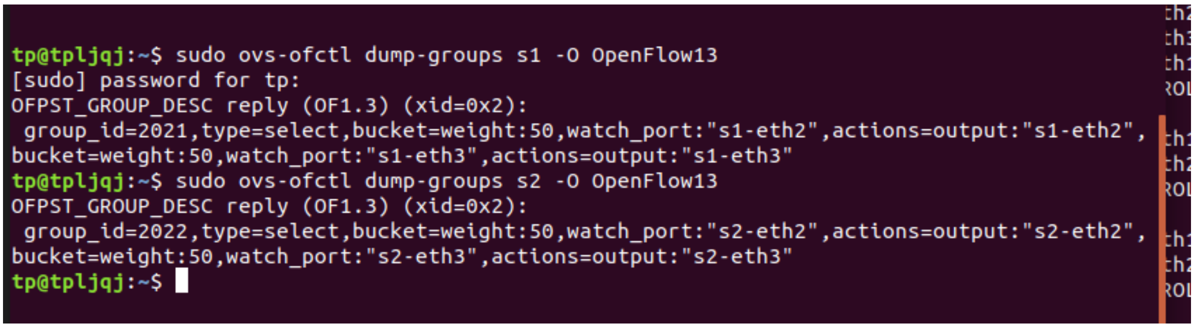
```
~$ sudo python3 problem1.py
~$ sudo ryu-manager --verbose problem3.py
```

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.

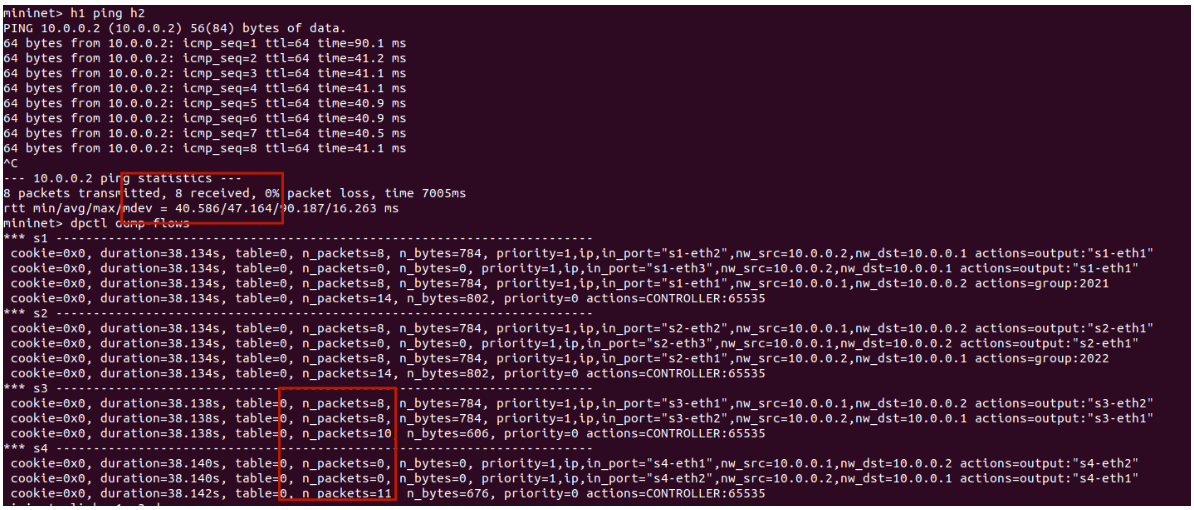
```
~$ sudo ovs-ofctl dump-groups s1 -O OpenFlow13
~$ sudo ovs-ofctl dump-groups s2 -O OpenFlow13
```



```
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.

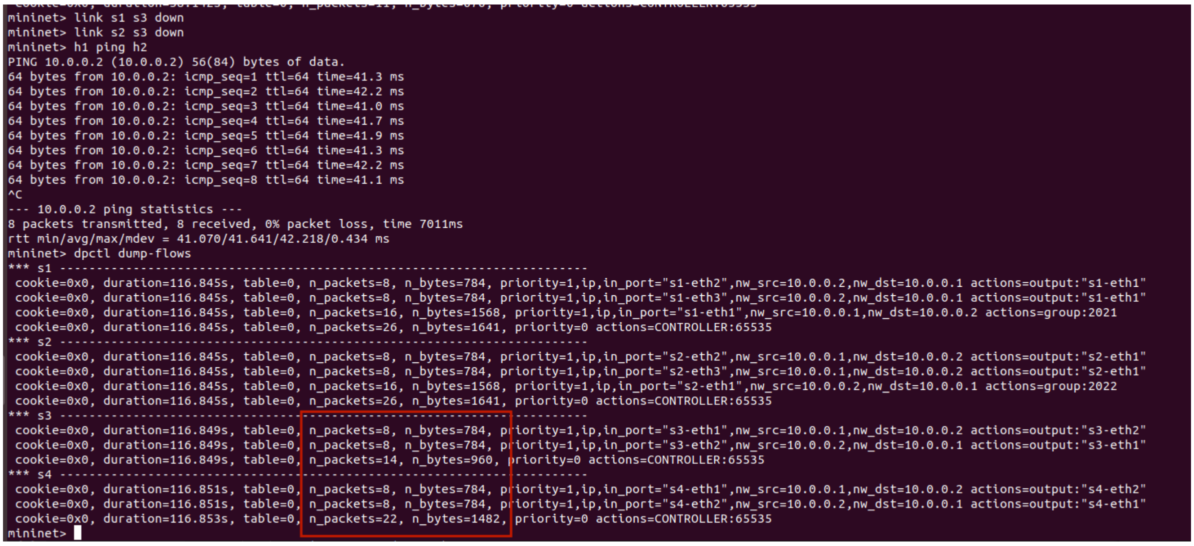


```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=90.1 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=41.2 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=41.1 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=41.1 ms
64 bytes from 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 7005ms
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=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=38.138s, table=0, n_packets=10, 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
```

Form 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 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.

```
mininet> link s2 s3 down
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**.

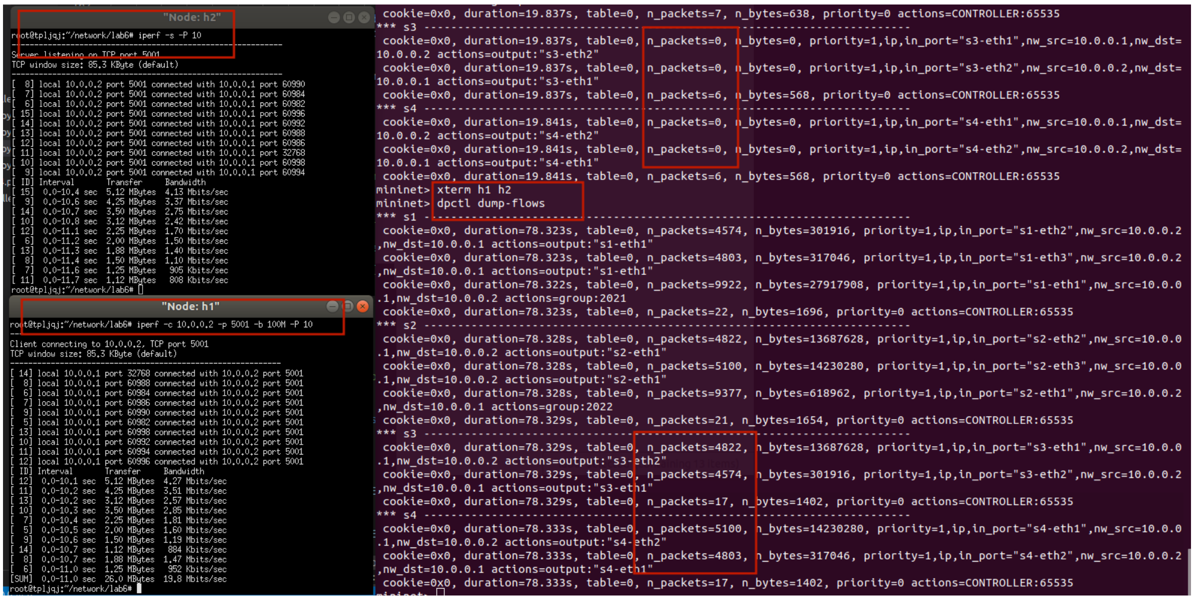


What's more,we can use the following commands to force the two paths to be used to transmit the packets.

```
mininet> xterm h1 h2
root@tpljqj:~/network/lab6# iperf -s -P 10
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.



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 **OFPGT_FF** rather than **OFPGT_SELECT**. The **OFPGT_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.

```
           s3
          /  \
h1 - s1        s2 - h2
          \  /
           s4
```

- 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 between **s1** and **s3** is broken.

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

code

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

```

from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER,
MAIN_DISPATCHER,DEAD_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet,ipv4,arp,ether_types
import time
from ryu.lib.packet import  in_proto as inet


class ExampleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(ExampleSwitch13, self).__init__(*args, **kwargs)
        # initialize mac address table.
        self.mac_to_port = {}
        self.datapaths = {}

    @set_ev_cls(ofp_event.EventOFPStateChange,
                [MAIN_DISPATCHER, DEAD_DISPATCHER])
    def _state_change_handler(self, ev):  #get all switches
        datapath = ev.datapath
        if ev.state == MAIN_DISPATCHER:
            if datapath.id not in self.datapaths:
                self.logger.debug('register datapath: %016x', datapath.id)
                self.datapaths[datapath.id] = datapath
        elif ev.state == DEAD_DISPATCHER:
            if datapath.id in self.datapaths:
                self.logger.debug('unregister datapath: %016x',
datapath.id)
                del self.datapaths[datapath.id]



    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        # install the table-miss flow entry.
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions,0)
```

```python

        #add flows
        #s1
        if datapath.id == 1:
            #s3-s1 get the packet from h2
            kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
                    ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            self.add_flow(datapath,1,match,actions,0)

            #s4-s1 get packets from h1
            kwargs = dict(in_port = 3, eth_type=ether_types.ETH_TYPE_IP,
                    ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            self.add_flow(datapath,1,match,actions,0)


            # add a group, the first patch  is s1-s3, the second path is s1-s4
            kwargs = dict(in_port = 1, eth_type=ether_types.ETH_TYPE_IP,
                  ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
            #kwargs = dict(in_port = 1)
            match = parser.OFPMatch(**kwargs)
            outport1 = 2
            outport2 = 3
            weight1 = 0
            weight2 = 0
            watch_port1 = 2
            watch_group1 = ofproto.OFPG_ANY
            watch_port2 = 3
            watch_group2 = ofproto.OFPG_ANY
            group_id = 2021


            self.add_group(datapath,match,outport1,outport2,weight1,weight2,watch_port1,watch_port2,watch_group1,watch_group2,group_id)




            #s3-s1 when link between s3 and s2 down , the packet in s3 will fowarding the packet to the s1, so s1 need transmit the packet to the s4
            # s1 forwaring the packet to the port 3(s4)

            kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
                    ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(3)]
            self.add_flow(datapath,1,match,actions,0)

            # when link between s4 and s2 down , the packet in s4 will fowarding the packet to the s1, so s1 need transmit the packet to the s3
            # s1 forwaring the packet to the port 2(s3)
            kwargs = dict(in_port = 3, eth_type=ether_types.ETH_TYPE_IP,
                    ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
            match = parser.OFPMatch(**kwargs)
```

```
            actions = [parser.OFPActionOutput(2)]
            self.add_flow(datapath,1,match,actions,0)

        #s2
        if datapath.id == 2:
            #s3-s2
            kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
                    ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            self.add_flow(datapath,1,match,actions,0)


            #s4-s2
            kwargs = dict(in_port = 3, eth_type=ether_types.ETH_TYPE_IP,
                    ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            self.add_flow(datapath,1,match,actions,0)

            #group s2-s3,s2-s4
            kwargs = dict(in_port = 1, eth_type=ether_types.ETH_TYPE_IP,
                    ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
            match = parser.OFPMatch(**kwargs)
            outport1 = 2
            outport2 = 3
            weight1 =  0
            weight2 = 0
            watch_port1 = 2
            watch_group1 = ofproto.OFPG_ANY
            watch_port2 = 3
            watch_group2 = ofproto.OFPG_ANY
            group_id = 2022
            self.add_group(datapath,match,outport1,outport2,weight1,weight2,watch_port1,watch_port2,watch_group1,watch_group2,group_id)

            #when the link down between s1 and s3
            kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
                    ipv4_src="10.0.0.2",ipv4_dst="10.0.0.1")
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(3)]
            self.add_flow(datapath,1,match,actions,0)

             #when the link down between s1 and s4
            kwargs = dict(in_port = 3, eth_type=ether_types.ETH_TYPE_IP,
                    ipv4_src="10.0.0.2",ipv4_dst="10.0.0.1")
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(2)]
            self.add_flow(datapath,1,match,actions,0)

        #s3
        if datapath.id == 3:


            #group when get the packet from in_port 1, it will forwarding it to the port 2 firstly
```

```
            # 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

            kwargs = dict(in_port = 1, eth_type=ether_types.ETH_TYPE_IP,
                  ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
            #kwargs = dict(in_port = 1)
            match = parser.OFPMatch(**kwargs)
            outport1 = 2
            outport2 = ofproto.OFPP_IN_PORT # when port 1 down, forwarding to the port2
            weight1 = 0
            weight2 = 0
            watch_port1 = 2
            watch_group1 = ofproto.OFPG_ANY
            watch_port2 = 1
            watch_group2 = ofproto.OFPG_ANY
            group_id = 2023
            self.add_group(datapath,match,outport1,outport2,weight1,weight2,watch_port1,watch_port2,watch_group1,watch_group2,group_id)


            #group when get the packet from in_port 2, if the link between s1 and s3 down, forwarding the packet to the s2
            kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
                  ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
            #kwargs = dict(in_port = 1)
            match = parser.OFPMatch(**kwargs)
            outport1 = 1
            outport2 = ofproto.OFPP_IN_PORT # when port 1 down, forwarding to the port2
            weight1 = 0
            weight2 = 0
            watch_port1 = 1
            watch_group1 = ofproto.OFPG_ANY
            watch_port2 = 2
            watch_group2 = ofproto.OFPG_ANY
            group_id = 2024
            self.add_group(datapath,match,outport1,outport2,weight1,weight2,watch_port1,watch_port2,watch_group1,watch_group2,group_id)





        #s4
        if datapath.id == 4:


            #group when get the packet from in_port 1, if the link between s2 and s3 down, forwarding the packet to the s1

            kwargs = dict(in_port = 1, eth_type=ether_types.ETH_TYPE_IP,
                  ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
            #kwargs = dict(in_port = 1)
            match = parser.OFPMatch(**kwargs)
            outport1=2
```

```
            outport2 = ofproto.OFPP_IN_PORT# when port 2 down, forwarding to the port1
            weight1 = 0
            weight2 = 0
            watch_port1 = 2
            watch_group1 = ofproto.OFPG_ANY
            watch_port2 = 1
            watch_group2 = ofproto.OFPG_ANY

            group_id = 2025

            self.add_group(datapath,match,outport1,outport2,weight1,weight2,watch_port1,watch_port2,watch_group1,watch_group2,group_id)


            #group when get the packet from in_port 2, if the link between s1 and s3 down, forwarding the packet to the s2
            kwargs = dict(in_port = 2, eth_type=ether_types.ETH_TYPE_IP,
                  ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
            #kwargs = dict(in_port = 1)
            match = parser.OFPMatch(**kwargs)
            outport1 = 1
            outport2 = ofproto.OFPP_IN_PORT
            weight1 = 0
            weight2 = 0
            watch_port1 = 1
            watch_group1 = ofproto.OFPG_ANY
            watch_port2 = 2
            watch_group2 = ofproto.OFPG_ANY
            group_id = 2026
            self.add_group(datapath,match,outport1,outport2,weight1,weight2,watch_port1,watch_port2,watch_group1,watch_group2,group_id)




    def add_group(self,datapath,match,outport1,outport2,weight1,weight2,watch_port1,watch_port2,watch_group1,watch_group2,group_id):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        actions1 = [parser.OFPActionOutput(outport1)]
        actions2 = [parser.OFPActionOutput(outport2)]
        buckets = [parser.OFPBucket(weight1, watch_port1, watch_group1,
                                        actions1),
                    parser.OFPBucket(weight2, watch_port2, watch_group2,
                                        actions2)]

        req = parser.OFPGroupMod(datapath, ofproto.OFPGC_ADD,
                                ofproto.OFPGT_FF, group_id, buckets)
        datapath.send_msg(req)
        actions = [parser.OFPActionGroup(group_id=group_id)]
        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                                actions)]
        mod = parser.OFPFlowMod(datapath=datapath, priority=1,
                                match=match, instructions=inst)
```

```python
        datapath.send_msg(mod)


    def add_flow(self, datapath, priority, match, actions,timeout):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        # construct flow_mod message and send it.
        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                             actions)]
        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                match=match, instructions=inst,hard_timeout=timeout)
        datapath.send_msg(mod)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        # get Datapath ID to identify OpenFlow switches.
        dpid = datapath.id
        self.mac_to_port.setdefault(dpid, {})

        # analyse the received packets using the packet library.
        pkt = packet.Packet(msg.data)
        eth_pkt = pkt.get_protocol(ethernet.ethernet)
        ipv4_pkt = pkt.get_protocol(ipv4.ipv4)
        #arp_pkt = pkt.get_protocol(arp.arp)
        dst = eth_pkt.dst
        src = eth_pkt.src
        ethtype = eth_pkt.ethertype

        in_port = msg.match['in_port']

        # get the received port number from packet_in message.


        self.logger.info("packet in %s  %s %s %s %s", dpid, src, dst, in_port,ethtype)
        if ipv4_pkt:
            ipv4dst = ipv4_pkt.dst
            ipv4src = ipv4_pkt.src
            self.logger.info("packet ipv4 in %s  %s %s %s %s", dpid, ipv4src, ipv4dst, in_port,ethtype)
        if ethtype == ether_types.ETH_TYPE_LLDP:  #ingore the LLDP packet
            return

        if ethtype == ether_types.ETH_TYPE_ARP or ethtype == ether_types.ETH_TYPE_IP: #deal with the arp packet
            if in_port == 1:  #send to both output1 and 2
                actions = [parser.OFPActionOutput(ofproto.OFPP_FLOOD)]
            if in_port == 2 or in_port==3:
                actions = [parser.OFPActionOutput(1)]


```

```
                # construct packet_out message and send it.
                out = parser.OFPPacketOut(datapath=datapath,
                                          buffer_id=ofproto.OFP_NO_BUFFER,
                                          in_port=in_port, actions=actions,
                                          data=msg.data)
                datapath.send_msg(out)
```

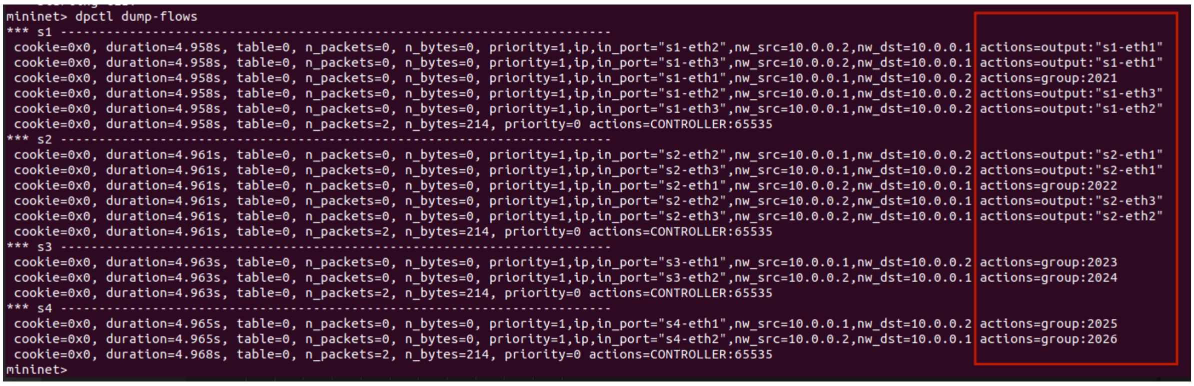
- In the **switch_features_handler()**, we add all needed flow entries and group entries to all switches. Here the weights of all buckets must be 0 because the group type is **OFPGT_FF** and we need choose the right watch_port. And because we needn't watch_group, we can set its value to be **OFPG_ANY**.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

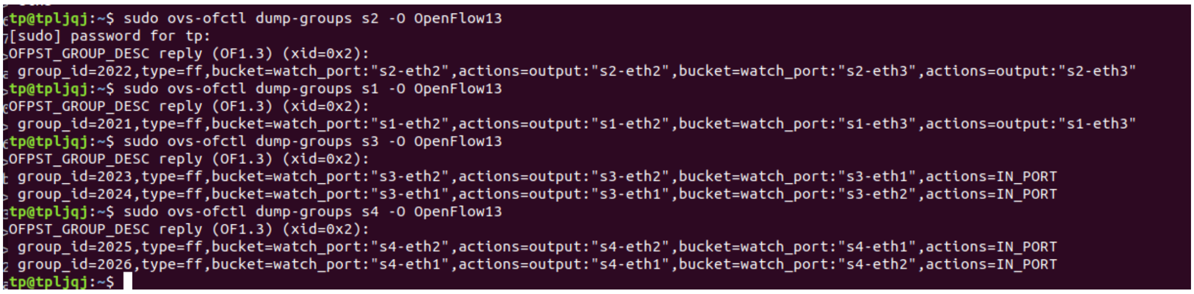
Result

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

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

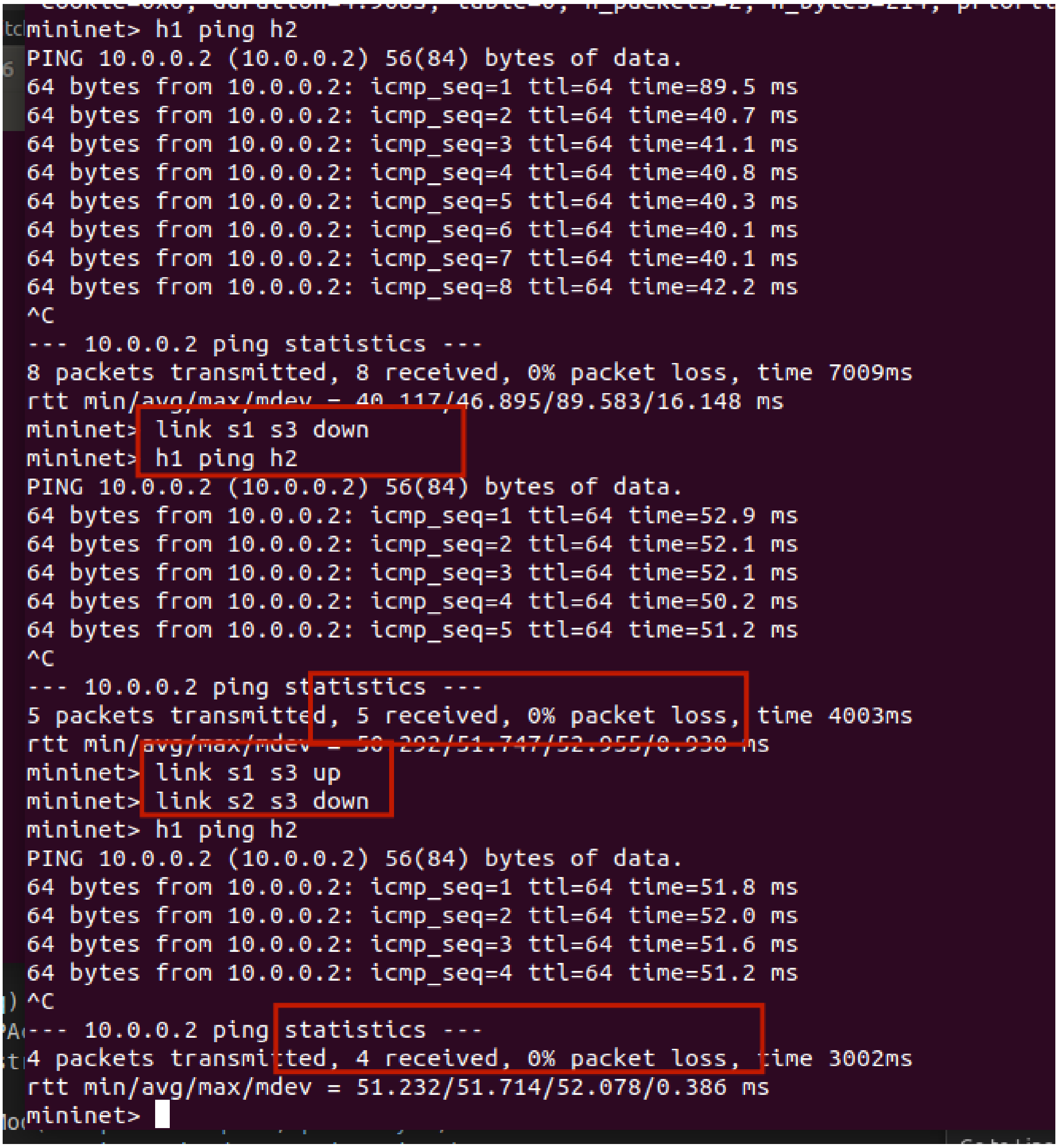
After running above commands, we can see the flow table of switches. The following figure shows that.





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

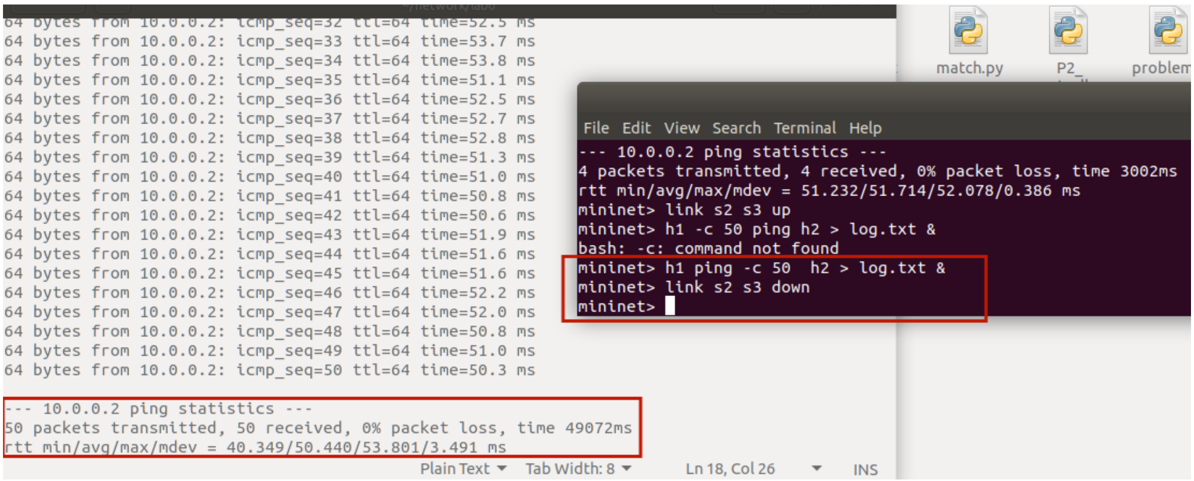
Then we can use **h1** ping **h2** and break down some links to see the results.



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.

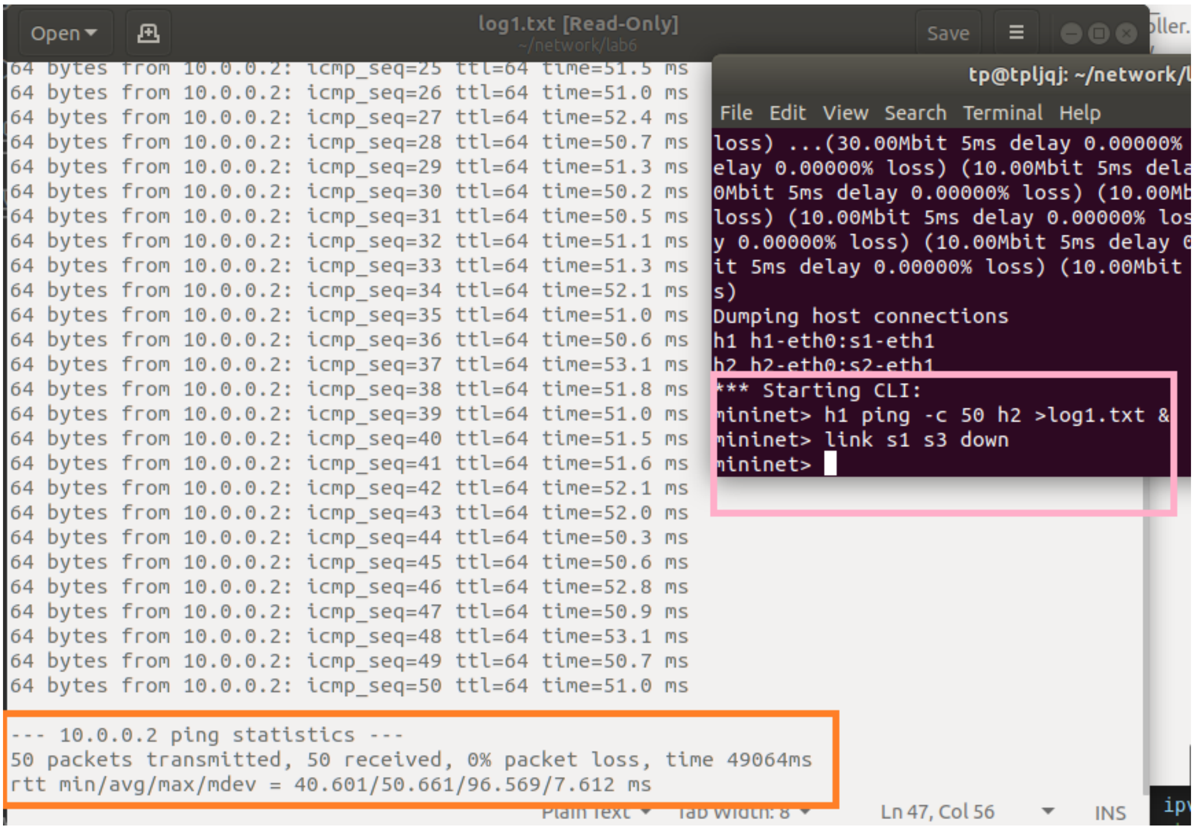
```
mininet> h1 ping -c 50  h2 > log.txt &
mininet> link s2 s3 down
```

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.



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

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



Conclusion

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