CS339 Lab SDN Design Report

Student Name: Zhou Haoquan

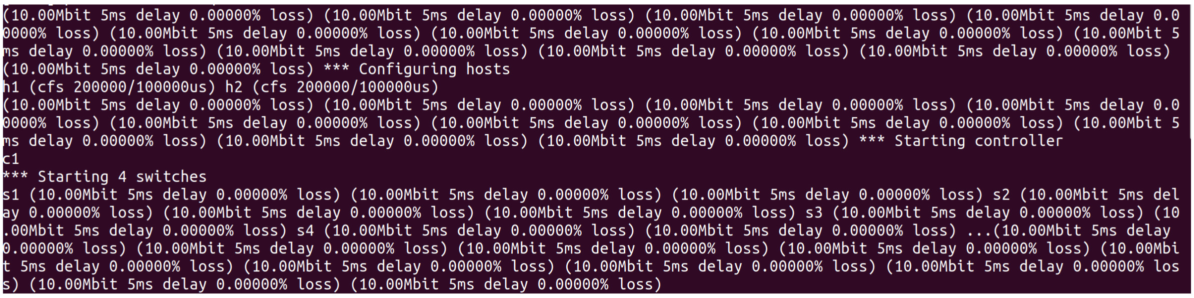
Student ID: 519370910059

1. Build Topology

The topology we built in this lab is the same as given by the sample code. Namely, having two hosts attached to the network composed to four routers, with two of them being intermediate ones.

```python
#!/usr/bin/python

"""

Sample Code

"""

from mininet.topo import Topo

from mininet.net import Mininet

from mininet.node import OVSBridge, OVSSwitch, OVSKernelSwitch

from mininet.node import CPULimitedHost

from mininet.node import RemoteController

from mininet.link import TCLink

from mininet.util import dumpNodeConnections

from mininet.log import setLogLevel, info

from mininet.cli import CLI

from sys import argv

# It would be nice if we didn't have to do this:

# pylint: disable=arguments-differ

def Test():

  "Create network and run simple performance test"

  net = Mininet( switch=OVSSwitch, host=CPULimitedHost, link=TCLink, autoStaticArp=False, controller=RemoteController)

  switch1 = net.addSwitch('s1')

  switch2 = net.addSwitch('s2')
  switch3 = net.addSwitch('s3')

  switch4 = net.addSwitch('s4')

  host1 = net.addHost('h1', cpu=.25)

  host2 = net.addHost('h2', cpu=.25)

  net.addLink(host1, switch1, bw=10, delay='5ms', loss=0, use_htb=True)

  net.addLink(host2, switch2, bw=10, delay='5ms', loss=0, use_htb=True)

  net.addLink(switch1, switch3, bw=10, delay='5ms', loss=0, use_htb=True)

  net.addLink(switch1, switch4, bw=10, delay='5ms', loss=0, use_htb=True)

  net.addLink(switch2, switch3, bw=10, delay='5ms', loss=0, use_htb=True)

  net.addLink(switch2, switch4, bw=10, delay='5ms', loss=0, use_htb=True)

  c1 = net.addController('c1', controller=RemoteController, ip="127.0.0.1", port=6653)

  net.build()

  c1.start()

  s1, s2, s3, s4 = net.getNodeByName('s1', 's2', 's3', 's4')

  s1.start([c1])

  s2.start([c1])

  s3.start([c1])

  s4.start([c1])

  net.start()

  info( "Dumping host connections\n" )

  dumpNodeConnections(net.hosts)

  h1, h2 = net.getNodeByName('h1', 'h2')

  CLI(net)

  net.stop()

if __name__ == '__main__':

  # setLogLevel( 'debug' )

  setLogLevel('info')

  Test()
```

The result of running Mininet is shown in the following figure, which is what we desired:



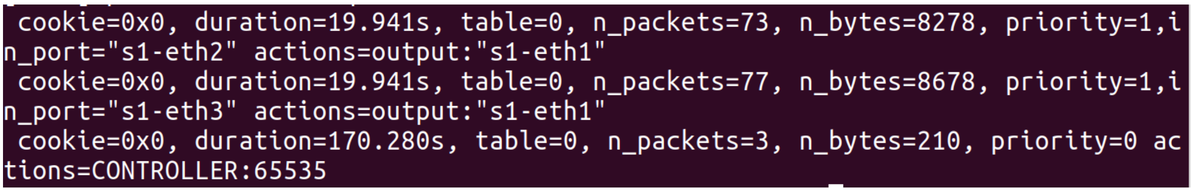
```
(10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.0
0000% loss) (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5
ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss)
(10.00Mbit 5ms delay 0.00000% loss) *** Configuring hosts
h1 (cfs 200000/100000us) h2 (cfs 200000/100000us)
(10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.0
0000% loss) (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5
ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss) *** Starting controller
c1
*** Starting 4 switches
s1 (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss) s2 (10.00Mbit 5ms del
ay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss) s3 (10.00Mbit 5ms delay 0.00000% loss) (10
.00Mbit 5ms delay 0.00000% loss) s4 (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss) ...(10.00Mbit 5ms delay
0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbi
t 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% los
s) (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss)
```

2. Path Switching

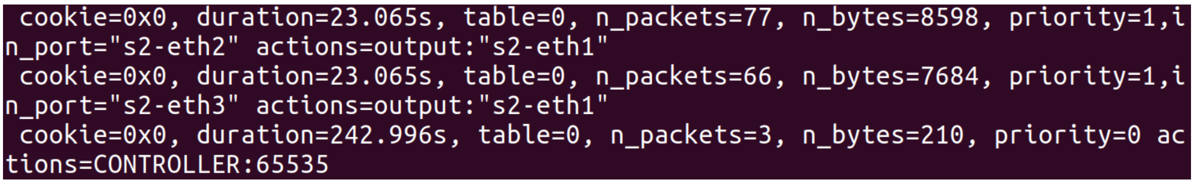
This part requires us to build two paths with each of them begin used for five seconds sequentially. And we can use the technique of putting parameter of `hard_timeout = 5` to the information sent to the switches from `h1` and `h2`. In this way the flow table will change every 5 seconds. Also, we set a global variable called `INIT_PORT`, which is initialized to be `2`. Every 5 seconds, the `INIT_PORT` is set to be `5-INIT_PORT` so that the output ports of `s1` and `s2` are switched from the upper path and lower path back and forth. The code is shown in the following part.

```
# Copyright (C) 2011 Nippon Telegraph and Telephone Corporation.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.lib.hub import Timeout
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types


class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    INIT_PORT = 2

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}

    @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 table-miss flow entry
        #
        # We specify NO BUFFER to max_len of the output action due to
        # OVS bug. At this moment, if we specify a lesser number, e.g.,
        # 128, OVS will send Packet-In with invalid buffer_id and
        # truncated packet data. In that case, we cannot output packets
        # correctly.  The bug has been fixed in OVS v2.1.0.
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

    def add_flow(self, datapath, priority, match, actions, buffer_id=None):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                             actions)]
        if buffer_id:
            mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
                                    priority=priority, match=match,
                                    instructions=inst)
        else:
            mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                    match=match, instructions=inst)
        datapath.send_msg(mod)

        if(datapath.id == 1):
            kwargs = dict(in_port=1)
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(2)]
            inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
            mod = parser.OFPFlowMod(datapath=datapath, priority=1, hard_timeout =
5,match=match, instructions=inst)
            datapath.send_msg(mod)

            kwargs = dict(in_port=2)
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
            mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
            datapath.send_msg(mod)

            kwargs = dict(in_port=3)
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
            mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
            datapath.send_msg(mod)
        if(datapath.id == 2 ):
            kwargs = dict(in_port=1)
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(2)]
            inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
            mod = parser.OFPFlowMod(datapath=datapath, priority=1, hard_timeout =
5,match=match, instructions=inst)
            datapath.send_msg(mod)

            kwargs = dict(in_port=2)
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
            mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
            datapath.send_msg(mod)

            kwargs = dict(in_port=3)
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
            mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
            datapath.send_msg(mod)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        # If you hit this you might want to increase
        # the "miss_send_length" of your switch
        if ev.msg.msg_len < ev.msg.total_len:
            self.logger.debug("packet truncated: only %s of %s bytes",
                              ev.msg.msg_len, ev.msg.total_len)
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        in_port = msg.match['in_port']

        pkt = packet.Packet(msg.data)
        eth = pkt.get_protocols(ethernet.ethernet)[0]

        if eth.ethertype == ether_types.ETH_TYPE_LLDP:
            # ignore lldp packet
            return
        dst = eth.dst
        src = eth.src

        dpid = format(datapath.id, "d").zfill(16)
        self.mac_to_port.setdefault(dpid, {})

        self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)

        # learn a mac address to avoid FLOOD next time.
        self.mac_to_port[dpid][src] = in_port
        if dst in self.mac_to_port[dpid]:
            out_port = self.mac_to_port[dpid][dst]
        else:
            out_port = ofproto.OFPP_FLOOD


        if(datapath.id == 1):
            kwargs = dict(in_port=1)
            match = parser.OFPMatch(**kwargs)
            self.INIT_PORT = 5 - self.INIT_PORT
            actions = [parser.OFPActionOutput(self.INIT_PORT)]
            inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
            mod = parser.OFPFlowMod(datapath=datapath, priority=1, hard_timeout =
5,match=match, instructions=inst)
            datapath.send_msg(mod)

            kwargs = dict(in_port=2)
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
            mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
            datapath.send_msg(mod)

            kwargs = dict(in_port=3)
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
            mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
            datapath.send_msg(mod)

        if(datapath.id == 2 ):
            kwargs = dict(in_port=1)
            match = parser.OFPMatch(**kwargs)
            self.INIT_PORT = 5 - self.INIT_PORT
            actions = [parser.OFPActionOutput(self.INIT_PORT)]
            inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
            mod = parser.OFPFlowMod(datapath=datapath, priority=1, hard_timeout =
5,match=match, instructions=inst)
            datapath.send_msg(mod)

            kwargs = dict(in_port=2)
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
            mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
            datapath.send_msg(mod)

            kwargs = dict(in_port=3)
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
            mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
            datapath.send_msg(mod)


        # elif(datapath.id == 3):
        #     out_port = ofproto.OFPP_FLOOD
        # else:
        #     out_port = ofproto.OFPP_FLOOD

        actions = [parser.OFPActionOutput(out_port)]

        # install a flow to avoid packet_in next time
        if out_port != ofproto.OFPP_FLOOD:
            match = parser.OFPMatch(in_port=in_port, eth_dst=dst, eth_src=src)
            # verify if we have a valid buffer_id, if yes avoid to send both
            # flow_mod & packet_out
            if msg.buffer_id != ofproto.OFP_NO_BUFFER:
                self.add_flow(datapath, 1, match, actions, msg.buffer_id)
                return
            else:
                self.add_flow(datapath, 1, match, actions)
        data = None
        if msg.buffer_id == ofproto.OFP_NO_BUFFER:
            data = msg.data

        out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                                  in_port=in_port, actions=actions, data=data)
        datapath.send_msg(out)
```

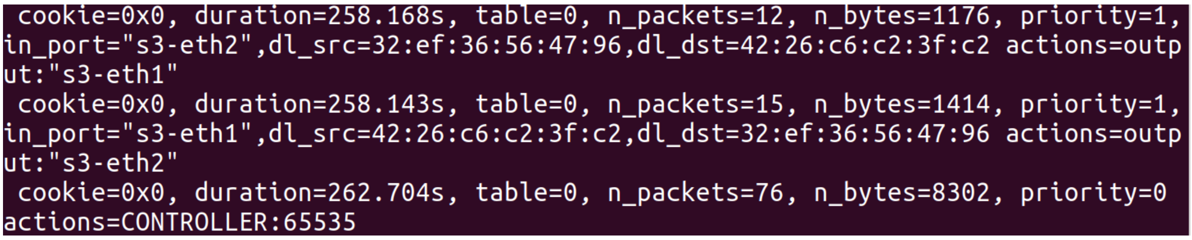
And we check the information from the `s1` to `s4`, the result is shown in the following figure.



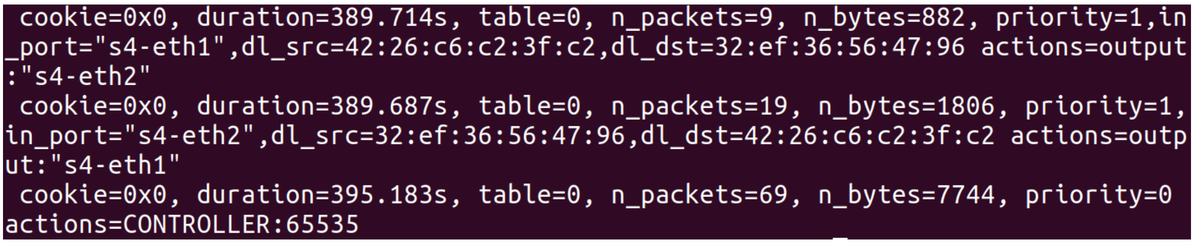
```
 cookie=0x0, duration=19.941s, table=0, n_packets=73, n_bytes=8278, priority=1,i
n_port="s1-eth2" actions=output:"s1-eth1"
 cookie=0x0, duration=19.941s, table=0, n_packets=77, n_bytes=8678, priority=1,i
n_port="s1-eth3" actions=output:"s1-eth1"
 cookie=0x0, duration=170.280s, table=0, n_packets=3, n_bytes=210, priority=0 ac
tions=CONTROLLER:65535
```



```
 cookie=0x0, duration=23.065s, table=0, n_packets=77, n_bytes=8598, priority=1,i
n_port="s2-eth2" actions=output:"s2-eth1"
 cookie=0x0, duration=23.065s, table=0, n_packets=66, n_bytes=7684, priority=1,i
n_port="s2-eth3" actions=output:"s2-eth1"
 cookie=0x0, duration=242.996s, table=0, n_packets=3, n_bytes=210, priority=0 ac
tions=CONTROLLER:65535
```

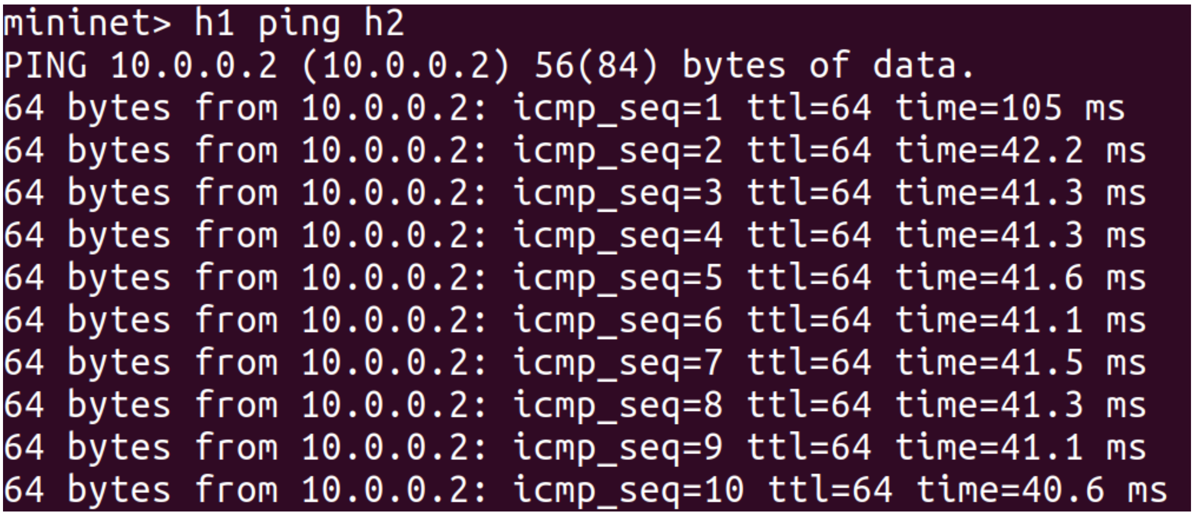


```
 cookie=0x0, duration=258.168s, table=0, n_packets=12, n_bytes=1176, priority=1,
in_port="s3-eth2",dl_src=32:ef:36:56:47:96,dl_dst=42:26:c6:c2:3f:c2 actions=outp
ut:"s3-eth1"
 cookie=0x0, duration=258.143s, table=0, n_packets=15, n_bytes=1414, priority=1,
in_port="s3-eth1",dl_src=42:26:c6:c2:3f:c2,dl_dst=32:ef:36:56:47:96 actions=outp
ut:"s3-eth2"
 cookie=0x0, duration=262.704s, table=0, n_packets=76, n_bytes=8302, priority=0
actions=CONTROLLER:65535
```



```
 cookie=0x0, duration=389.714s, table=0, n_packets=9, n_bytes=882, priority=1,in
_port="s4-eth1",dl_src=42:26:c6:c2:3f:c2,dl_dst=32:ef:36:56:47:96 actions=output
:"s4-eth2"
 cookie=0x0, duration=389.687s, table=0, n_packets=19, n_bytes=1806, priority=1,
in_port="s4-eth2",dl_src=32:ef:36:56:47:96,dl_dst=42:26:c6:c2:3f:c2 actions=outp
ut:"s4-eth1"
 cookie=0x0, duration=395.183s, table=0, n_packets=69, n_bytes=7744, priority=0
actions=CONTROLLER:65535
```

And we see that both two links are working and they share the same throughput, since the packets sending on both links are roughly the same. The result of `h1 ping h2` is shown in the 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=105 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=41.3 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=41.3 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=41.6 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=41.1 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=41.5 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=41.3 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=41.1 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=40.6 ms
```

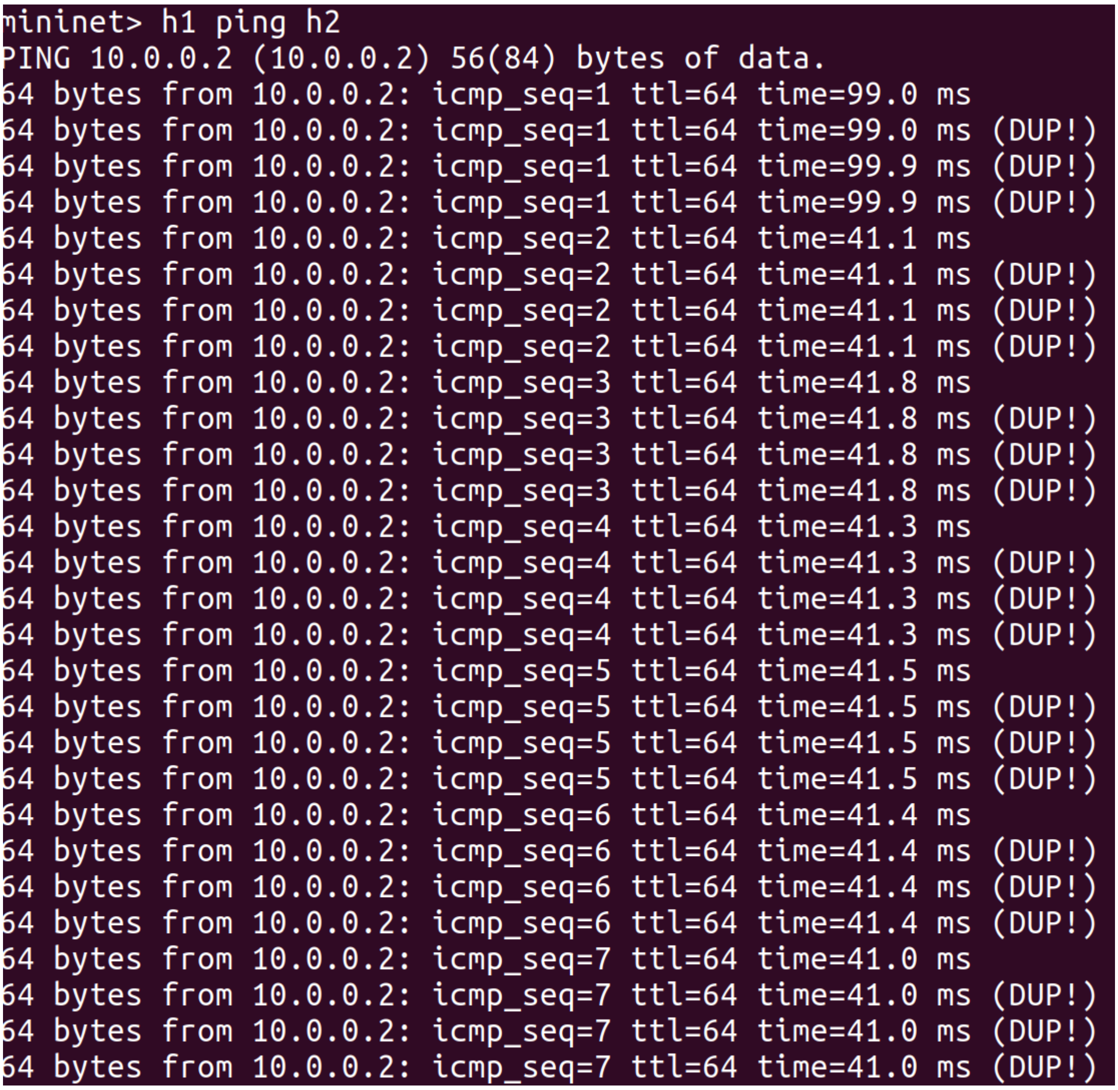
3. Dual Path

This time, we want both paths to be working at the same time, we can just make the `s1` and `s2` sending packets both to `port 2` and `port 3` concurrently. The code is shown as follows

```python
# Copyright (C) 2011 Nippon Telegraph and Telephone Corporation.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.lib.hub import Timeout
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types
```

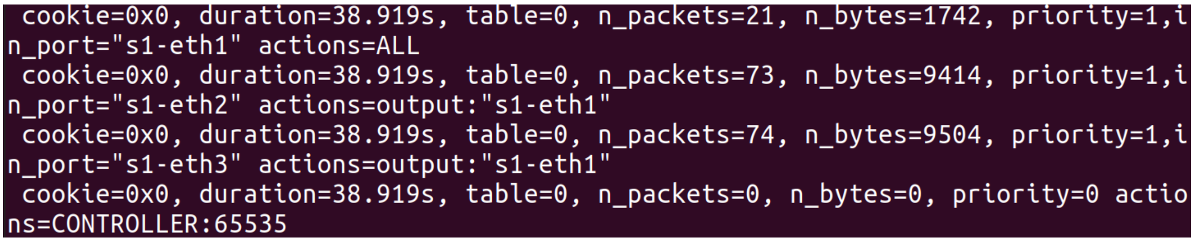
```
class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    INIT_PORT = 2

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}

    @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 table-miss flow entry
        #
        # We specify NO BUFFER to max_len of the output action due to
        # OVS bug. At this moment, if we specify a lesser number, e.g.,
        # 128, OVS will send Packet-In with invalid buffer_id and
        # truncated packet data. In that case, we cannot output packets
        # correctly.  The bug has been fixed in OVS v2.1.0.
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

    def add_flow(self, datapath, priority, match, actions, buffer_id=None):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                             actions)]
        if buffer_id:
            mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
                                    priority=priority, match=match,
                                    instructions=inst)
        else:
            mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                    match=match, instructions=inst)
        datapath.send_msg(mod)

        if(datapath.id == 1):
            kwargs = dict(in_port=1)
            match = parser.OFPMatch(**kwargs)
            actions1 = [parser.OFPActionOutput(ofproto_v1_3.OFPP_ALL)]
            inst1 =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions1)]
            mod1 = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst1)
            datapath.send_msg(mod1)
            # actions2 = [parser.OFPActionOutput(3)]
            # inst2=
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions2)]
            # mod2 = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst2)
            # datapath.send_msg(mod2)

            kwargs = dict(in_port=2)
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
            mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
            datapath.send_msg(mod)

            kwargs = dict(in_port=3)
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
            mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
            datapath.send_msg(mod)

        if(datapath.id == 2 ):
            kwargs = dict(in_port=1)
            match = parser.OFPMatch(**kwargs)
            actions1 = [parser.OFPActionOutput(ofproto_v1_3.OFPP_ALL)]
            inst1 =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions1)]
            mod1 = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst1)
            datapath.send_msg(mod1)
            # actions2 = [parser.OFPActionOutput(3)]
            # inst2 =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions2)]
            # mod2 = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst2)
            # datapath.send_msg(mod2)

            kwargs = dict(in_port=2)
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
            mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
            datapath.send_msg(mod)

            kwargs = dict(in_port=3)
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
            mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
            datapath.send_msg(mod)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        # If you hit this you might want to increase
        # the "miss_send_length" of your switch
        if ev.msg.msg_len < ev.msg.total_len:
            self.logger.debug("packet truncated: only %s of %s bytes",
                              ev.msg.msg_len, ev.msg.total_len)
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        in_port = msg.match['in_port']

        pkt = packet.Packet(msg.data)
        eth = pkt.get_protocols(ethernet.ethernet)[0]

        if eth.ethertype == ether_types.ETH_TYPE_LLDP:
            # ignore lldp packet
            return
        dst = eth.dst
        src = eth.src

        dpid = format(datapath.id, "d").zfill(16)
        self.mac_to_port.setdefault(dpid, {})

        self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)

        # learn a mac address to avoid FLOOD next time.
        self.mac_to_port[dpid][src] = in_port

        if dst in self.mac_to_port[dpid]:
            out_port = self.mac_to_port[dpid][dst]
        else:
            out_port = ofproto.OFPP_FLOOD


        if(datapath.id == 1):
            kwargs = dict(in_port=1)
            match = parser.OFPMatch(**kwargs)
            actions1 = [parser.OFPActionOutput(ofproto_v1_3.OFPP_ALL)]
            inst1 =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions1)]
            mod1 = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst1)
            datapath.send_msg(mod1)
            # actions2 = [parser.OFPActionOutput(3)]
            # inst2=
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions2)]
            # mod2 = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst2)
            # datapath.send_msg(mod2)

            kwargs = dict(in_port=2)
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
            mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
            datapath.send_msg(mod)

            kwargs = dict(in_port=3)
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
            mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
            datapath.send_msg(mod)

        if(datapath.id == 2 ):
            kwargs = dict(in_port=1)
            match = parser.OFPMatch(**kwargs)
            actions1 = [parser.OFPActionOutput(ofproto_v1_3.OFPP_ALL)]
            inst1 =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions1)]
            mod1 = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst1)
            datapath.send_msg(mod1)
            # actions2 = [parser.OFPActionOutput(3)]
            # inst2 =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions2)]
            # mod2 = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst2)
            # datapath.send_msg(mod2)

            kwargs = dict(in_port=2)
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
            mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
            datapath.send_msg(mod)

            kwargs = dict(in_port=3)
            match = parser.OFPMatch(**kwargs)
            actions = [parser.OFPActionOutput(1)]
            inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
            mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
            datapath.send_msg(mod)



        # elif(datapath.id == 3):
        #     out_port = ofproto.OFPP_FLOOD
        # else:
        #     out_port = ofproto.OFPP_FLOOD

        actions = [parser.OFPActionOutput(out_port)]

        # install a flow to avoid packet_in next time
        if out_port != ofproto.OFPP_FLOOD:
            match = parser.OFPMatch(in_port=in_port, eth_dst=dst, eth_src=src)
            # verify if we have a valid buffer_id, if yes avoid to send both
            # flow_mod & packet_out
            if msg.buffer_id != ofproto.OFP_NO_BUFFER:
                self.add_flow(datapath, 1, match, actions, msg.buffer_id)
                return
            else:
                self.add_flow(datapath, 1, match, actions)
        data = None
        if msg.buffer_id == ofproto.OFP_NO_BUFFER:
            data = msg.data

        out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                                  in_port=in_port, actions=actions, data=data)
        datapath.send_msg(out)
```

Note that we use the `ofproto_v1_3.OFPP_ALL` to ensure that the packet from `s1` and `s2` are sent to both links. The result is shown in the following figure.

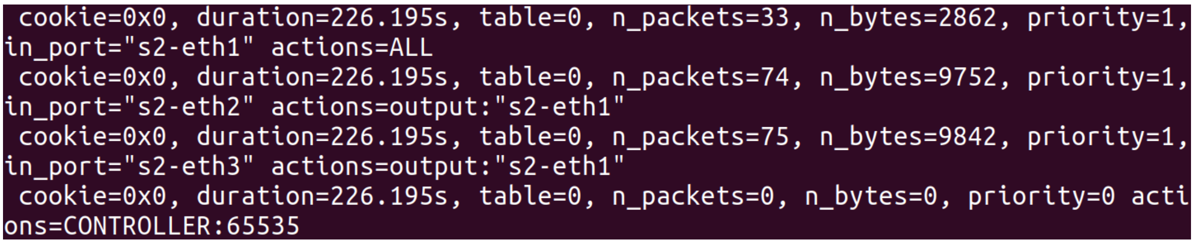


```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=99.0 ms
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=99.0 ms (DUP!)
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=99.9 ms (DUP!)
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=99.9 ms (DUP!)
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=41.1 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=41.1 ms (DUP!)
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=41.1 ms (DUP!)
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=41.1 ms (DUP!)
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=41.8 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=41.8 ms (DUP!)
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=41.8 ms (DUP!)
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=41.8 ms (DUP!)
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=41.3 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=41.3 ms (DUP!)
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=41.3 ms (DUP!)
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=41.3 ms (DUP!)
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=41.5 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=41.5 ms (DUP!)
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=41.5 ms (DUP!)
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=41.5 ms (DUP!)
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=41.4 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=41.4 ms (DUP!)
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=41.4 ms (DUP!)
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=41.4 ms (DUP!)
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=41.0 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=41.0 ms (DUP!)
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=41.0 ms (DUP!)
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=41.0 ms (DUP!)
```

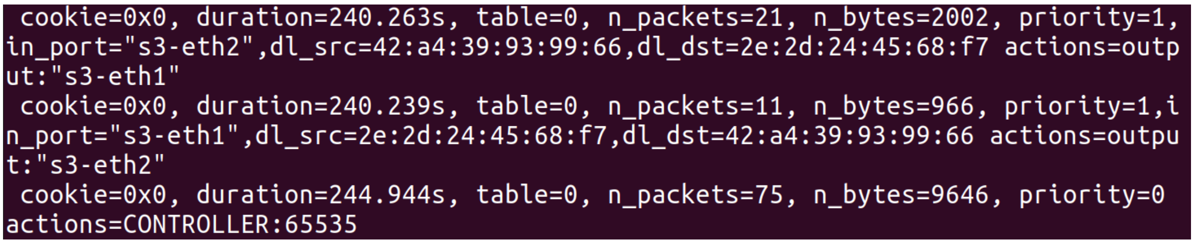
Here we see that there are in total four packets received by `h1` from `h2`, this is because the request packets are sent from both links, and `h2` will send two responses for each of the request message. By `2x2 = 4` there are in total four packets sent back. The configuration results are shown in the following figures. The links they show are what we expect.



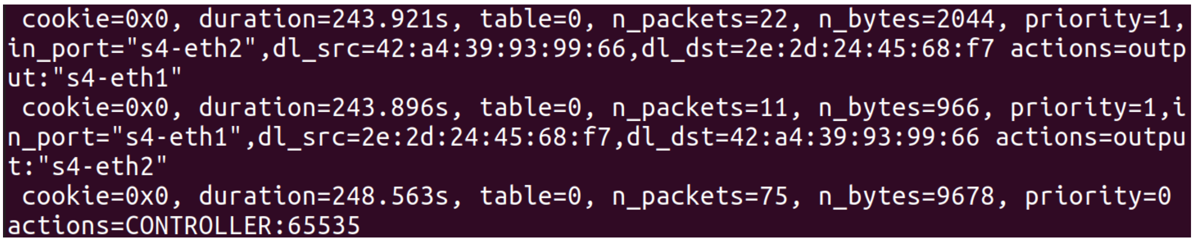
```
 cookie=0x0, duration=38.919s, table=0, n_packets=21, n_bytes=1742, priority=1,i
n_port="s1-eth1" actions=ALL
 cookie=0x0, duration=38.919s, table=0, n_packets=73, n_bytes=9414, priority=1,i
n_port="s1-eth2" actions=output:"s1-eth1"
 cookie=0x0, duration=38.919s, table=0, n_packets=74, n_bytes=9504, priority=1,i
n_port="s1-eth3" actions=output:"s1-eth1"
 cookie=0x0, duration=38.919s, table=0, n_packets=0, n_bytes=0, priority=0 actio
ns=CONTROLLER:65535
```



```
 cookie=0x0, duration=226.195s, table=0, n_packets=33, n_bytes=2862, priority=1,
in_port="s2-eth1" actions=ALL
 cookie=0x0, duration=226.195s, table=0, n_packets=74, n_bytes=9752, priority=1,
in_port="s2-eth2" actions=output:"s2-eth1"
 cookie=0x0, duration=226.195s, table=0, n_packets=75, n_bytes=9842, priority=1,
in_port="s2-eth3" actions=output:"s2-eth1"
 cookie=0x0, duration=226.195s, table=0, n_packets=0, n_bytes=0, priority=0 acti
ons=CONTROLLER:65535
```



```
 cookie=0x0, duration=240.263s, table=0, n_packets=21, n_bytes=2002, priority=1,
in_port="s3-eth2",dl_src=42:a4:39:93:99:66,dl_dst=2e:2d:24:45:68:f7 actions=outp
ut:"s3-eth1"
 cookie=0x0, duration=240.239s, table=0, n_packets=11, n_bytes=966, priority=1,i
n_port="s3-eth1",dl_src=2e:2d:24:45:68:f7,dl_dst=42:a4:39:93:99:66 actions=outpu
t:"s3-eth2"
 cookie=0x0, duration=244.944s, table=0, n_packets=75, n_bytes=9646, priority=0
actions=CONTROLLER:65535
```



```
 cookie=0x0, duration=243.921s, table=0, n_packets=22, n_bytes=2044, priority=1,
in_port="s4-eth2",dl_src=42:a4:39:93:99:66,dl_dst=2e:2d:24:45:68:f7 actions=outp
ut:"s4-eth1"
 cookie=0x0, duration=243.896s, table=0, n_packets=11, n_bytes=966, priority=1,i
n_port="s4-eth1",dl_src=2e:2d:24:45:68:f7,dl_dst=42:a4:39:93:99:66 actions=outpu
t:"s4-eth2"
 cookie=0x0, duration=248.563s, table=0, n_packets=75, n_bytes=9678, priority=0
actions=CONTROLLER:65535
```

4. Solve Link Failure

The group table is constructed so that the entry have two flow options for the flow from `h1` to `h2` so that when one link is down, the newly constructed table will automatically select the second link that is valid.

```
# Copyright (C) 2011 Nippon Telegraph and Telephone Corporation.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.lib.hub import Timeout
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types


class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    INIT_PORT = 2
    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}

    @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 table-miss flow entry
        #
        # We specify NO BUFFER to max_len of the output action due to
        # OVS bug. At this moment, if we specify a lesser number, e.g.,
        # 128, OVS will send Packet-In with invalid buffer_id and
        # truncated packet data. In that case, we cannot output packets
        # correctly.  The bug has been fixed in OVS v2.1.0.
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

    def add_flow(self, datapath, priority, match, actions, buffer_id=None):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                             actions)]
        if buffer_id:
            mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
                                    priority=priority, match=match,
                                    instructions=inst)
        else:
            mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                    match=match, instructions=inst)
        datapath.send_msg(mod)

        # if(datapath.id == 1):
        #     kwargs = dict(in_port=1)
        #     match = parser.OFPMatch(**kwargs)
        #     actions1 = [parser.OFPActionOutput(2)]
        #     inst1 =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions1)]
        #     mod1 = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst1)
        #     datapath.send_msg(mod1)
        #     # actions2 = [parser.OFPActionOutput(3)]
        #     # inst2=
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions2)]
        #     # mod2 = parser.OFPFlowMod(datapath=datapath,
priority=1,match=match, instructions=inst2)
        #     # datapath.send_msg(mod2)

        #     kwargs = dict(in_port=2)
        #     match = parser.OFPMatch(**kwargs)
        #     actions = [parser.OFPActionOutput(1)]
        #     inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
        #     mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
        #     datapath.send_msg(mod)

        #     kwargs = dict(in_port=3)
        #     match = parser.OFPMatch(**kwargs)
        #     actions = [parser.OFPActionOutput(1)]
        #     inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
        #     mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
        #     datapath.send_msg(mod)

        # if(datapath.id == 2 ):
        #     kwargs = dict(in_port=1)
        #     match = parser.OFPMatch(**kwargs)
        #     actions1 = [parser.OFPActionOutput(2)]
        #     inst1 =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions1)]
        #     mod1 = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst1)
        #     datapath.send_msg(mod1)
        #     # actions2 = [parser.OFPActionOutput(3)]
        #     # inst2 =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions2)]
        #     # mod2 = parser.OFPFlowMod(datapath=datapath,
priority=1,match=match, instructions=inst2)
        #     # datapath.send_msg(mod2)

        #     kwargs = dict(in_port=2)
        #     match = parser.OFPMatch(**kwargs)
        #     actions = [parser.OFPActionOutput(1)]
        #     inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
        #     mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
        #     datapath.send_msg(mod)

        #     kwargs = dict(in_port=3)
        #     match = parser.OFPMatch(**kwargs)
        #     actions = [parser.OFPActionOutput(1)]
        #     inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
        #     mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
        #     datapath.send_msg(mod)

        if(datapath == 1):
            ofp_parser = datapath.ofproto_parser
            port = 2
            actions1 = [ofp_parser.OFPActionOutput(port)]
            port = 1
            actions2 = [ofp_parser.OFPActionOutput(port)]
            weight1 = 50
            weight2 = 50
            buckets = [ofp_parser.OFPBucket(weight1, 1, 1,actions1),
ofp_parser.OFPBucket(weight1, 2, 1,actions2),ofp_parser.OFPBucket(weight1, 3,
1,actions2)]
            group_id = 1
            req = ofp_parser.OFPGroupMod(datapath,
ofproto.OFPGC_ADD,ofproto.OFPGT_SELECT, 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)
            datapath.send_msg(mod)
        if(datapath == 2):
            ofp_parser = datapath.ofproto_parser
            port = 2
            actions1 = [ofp_parser.OFPActionOutput(port)]
            port = 1
            actions2 = [ofp_parser.OFPActionOutput(port)]
            weight1 = 50
            weight2 = 50
            watch_port = 0
            watch_group = 0
            buckets = [ofp_parser.OFPBucket(weight1, 1,
1,actions1),ofp_parser.OFPBucket(weight1, 2,
1,actions2),ofp_parser.OFPBucket(weight1, 3, 1,actions2)]
            group_id = 1
            req = ofp_parser.OFPGroupMod(datapath,
ofproto.OFPGC_ADD,ofproto.OFPGT_SELECT, 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=priority,match=match, instructions=inst)
            datapath.send_msg(mod)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        # If you hit this you might want to increase
        # the "miss_send_length" of your switch
        if ev.msg.msg_len < ev.msg.total_len:
            self.logger.debug("packet truncated: only %s of %s bytes",
                              ev.msg.msg_len, ev.msg.total_len)
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        in_port = msg.match['in_port']

        pkt = packet.Packet(msg.data)
        eth = pkt.get_protocols(ethernet.ethernet)[0]

        if eth.ethertype == ether_types.ETH_TYPE_LLDP:
            # ignore lldp packet
            return
        dst = eth.dst
        src = eth.src
        dpid = format(datapath.id, "d").zfill(16)
        self.mac_to_port.setdefault(dpid, {})

        self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)

        # learn a mac address to avoid FLOOD next time.
        self.mac_to_port[dpid][src] = in_port

        if dst in self.mac_to_port[dpid]:
            out_port = self.mac_to_port[dpid][dst]
        else:
            out_port = ofproto.OFPP_FLOOD


        # if(datapath.id == 1):
        #     kwargs = dict(in_port=1)
        #     match = parser.OFPMatch(**kwargs)
        #     actions1 = [parser.OFPActionOutput(2)]
        #     inst1 =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions1)]
        #     mod1 = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst1)
        #     datapath.send_msg(mod1)
        #     # actions2 = [parser.OFPActionOutput(3)]
        #     # inst2=
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions2)]
        #     # mod2 = parser.OFPFlowMod(datapath=datapath,
priority=1,match=match, instructions=inst2)
        #     # datapath.send_msg(mod2)

        #     kwargs = dict(in_port=2)
        #     match = parser.OFPMatch(**kwargs)
        #     actions = [parser.OFPActionOutput(1)]
        #     inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
        #     mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
        #     datapath.send_msg(mod)

        #     kwargs = dict(in_port=3)
        #     match = parser.OFPMatch(**kwargs)
        #     actions = [parser.OFPActionOutput(1)]
        #     inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
        #     mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
        #     datapath.send_msg(mod)

        # if(datapath.id == 2 ):
        #     kwargs = dict(in_port=1)
        #     match = parser.OFPMatch(**kwargs)
        #     actions1 = [parser.OFPActionOutput(2)]
        #     inst1 =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions1)]
        #     mod1 = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst1)
        #     datapath.send_msg(mod1)
        #     # actions2 = [parser.OFPActionOutput(3)]
        #     # inst2 =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions2)]
        #     # mod2 = parser.OFPFlowMod(datapath=datapath,
priority=1,match=match, instructions=inst2)
        #     # datapath.send_msg(mod2)

        #     kwargs = dict(in_port=2)
        #     match = parser.OFPMatch(**kwargs)
        #     actions = [parser.OFPActionOutput(1)]
        #     inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
        #     mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
        #     datapath.send_msg(mod)

        #     kwargs = dict(in_port=3)
        #     match = parser.OFPMatch(**kwargs)
        #     actions = [parser.OFPActionOutput(1)]
        #     inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
        #     mod = parser.OFPFlowMod(datapath=datapath, priority=1,match=match,
instructions=inst)
        #     datapath.send_msg(mod)

        match = parser.OFPMatch()
        if(datapath == 1):
            ofp_parser = datapath.ofproto_parser
            port = 2
            actions1 = [ofp_parser.OFPActionOutput(port)]
            port = 1
            actions2 = [ofp_parser.OFPActionOutput(port)]
            weight1 = 50
            weight2 = 50
            watch_port = 0
            watch_group = 0
            buckets = [ofp_parser.OFPBucket(weight1, 1,
1,actions1),ofp_parser.OFPBucket(weight1, 2,
1,actions2),ofp_parser.OFPBucket(weight1, 3, 1,actions2)]
            group_id = 1
            req = ofp_parser.OFPGroupMod(datapath,
ofproto.OFPGC_ADD,ofproto.OFPGT_SELECT, 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)
            datapath.send_msg(mod)
        if(datapath == 2):
            ofp_parser = datapath.ofproto_parser
            port = 2
            actions1 = [ofp_parser.OFPActionOutput(port)]
            port = 1
            actions2 = [ofp_parser.OFPActionOutput(port)]
            weight1 = 50
            weight2 = 50
            watch_port = 0
            watch_group = 0
            buckets = [ofp_parser.OFPBucket(weight1, 1, 1,actions1),
            ofp_parser.OFPBucket(weight2, 2, 1,actions2),
ofp_parser.OFPBucket(weight1, 3, 1,actions2)]
            group_id = 1
            req = ofp_parser.OFPGroupMod(datapath,
ofproto.OFPGC_ADD,ofproto.OFPGT_SELECT, 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)
            datapath.send_msg(mod)

        # elif(datapath.id == 3):
        #     out_port = ofproto.OFPP_FLOOD
        # else:
        #     out_port = ofproto.OFPP_FLOOD

        actions = [parser.OFPActionOutput(out_port)]

        # install a flow to avoid packet_in next time
        if out_port != ofproto.OFPP_FLOOD:
            match = parser.OFPMatch(in_port=in_port, eth_dst=dst, eth_src=src)
            # verify if we have a valid buffer_id, if yes avoid to send both
            # flow_mod & packet_out
            if msg.buffer_id != ofproto.OFP_NO_BUFFER:
                self.add_flow(datapath, 1, match, actions, msg.buffer_id)
                return
            else:
                self.add_flow(datapath, 1, match, actions)
        data = None
        if msg.buffer_id == ofproto.OFP_NO_BUFFER:
            data = msg.data

        out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                                  in_port=in_port, actions=actions, data=data)
        datapath.send_msg(out)
```