SDN Lab Report

1. Introduction

> For more details, you can find all the.py files in the attachment

Ryu provides software components with well defined API's that make it easy for developers to create new network management and control applications. Ryu supports various protocols for managing network devices, such as OpenFlow, Netconf, OF-config, etc.

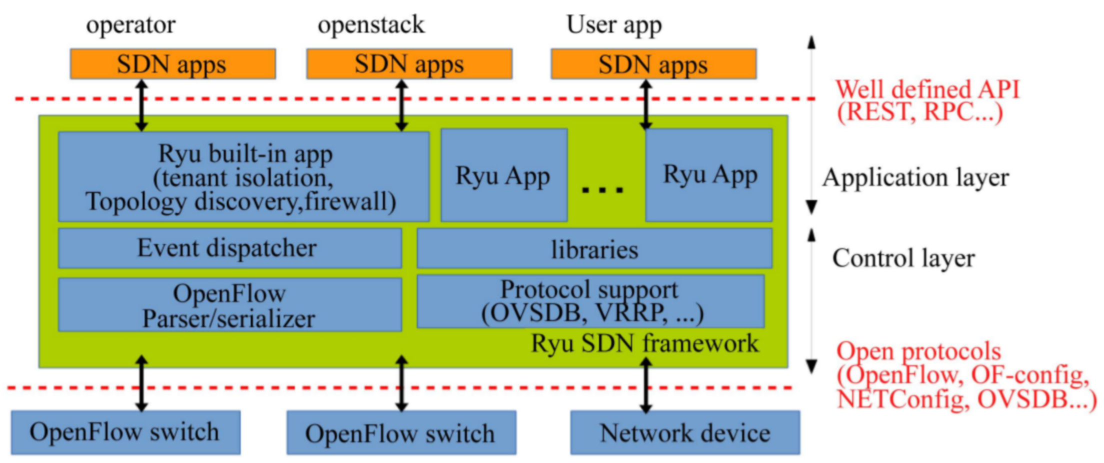
In this lab, we use Ryu to customize an OpenFlow Controller to achieve path switching, load balancing, faliover functions respectively.

And in the following report, we will introduce the eight parts of RYU analysis, Network Topology, Tool App, path switching, load balancing, fast failover, problems encountered and solutions, summary.

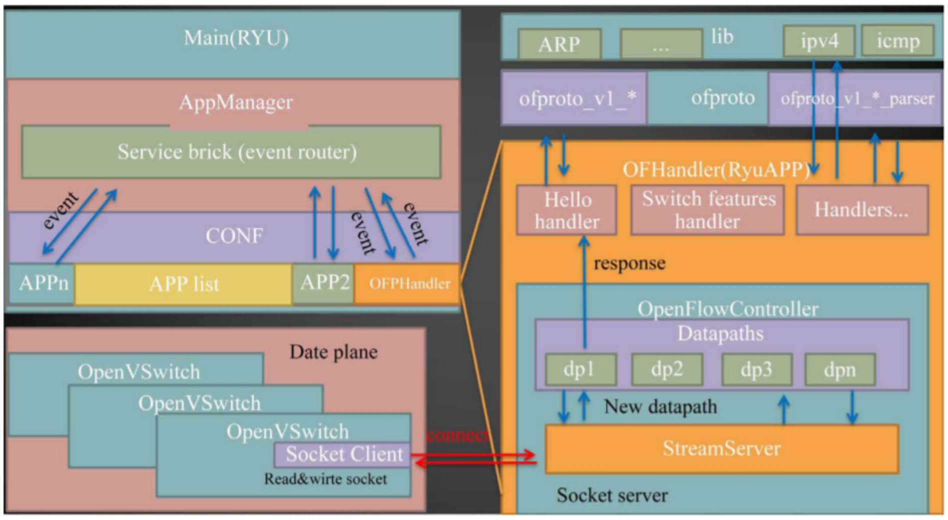
2. RYU Analysis

Ryu is a very well designed OpenFlow framework. An in-depth understanding of it will be very helpful in designing our own communication framework in the future. So I think it's worth taking a look at the source code of it and analyzing how it works.

Framework



- Modules

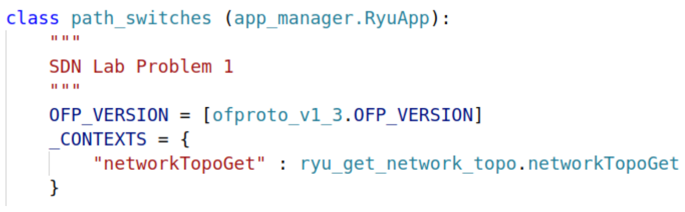


App start-up procedure

1) load apps

```
# /cmd/manager.py main()
def main(args=None, prog=None):
    # Omit some code
    app_lists = CONF.app_lists + CONF.app
        # keep old behavior, run ofp if no application is specified.
        if not app_lists:
            app_lists = ['ryu.controller.ofp_handler']

        app_mgr = AppManager.get_instance()
        app_mgr.load_apps(app_lists)
```

We usually don't instantiate one app but a app list, because Ryu allows us to instantiate other apps in one APP to take advantage of services they provide. We can declare up front what app we're going to use by defining in `_CONTEXTS`. Such as:


```
class path_switches (app_manager.RyuApp):
    """
    SDN Lab Problem 1
    """
    OFP_VERSION = [ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {
        "networkTopoGet" : ryu_get_network_topo.networkTopoGet
    }
```

But this invocation relationship can usually have only two levels.

2) Initialize the app

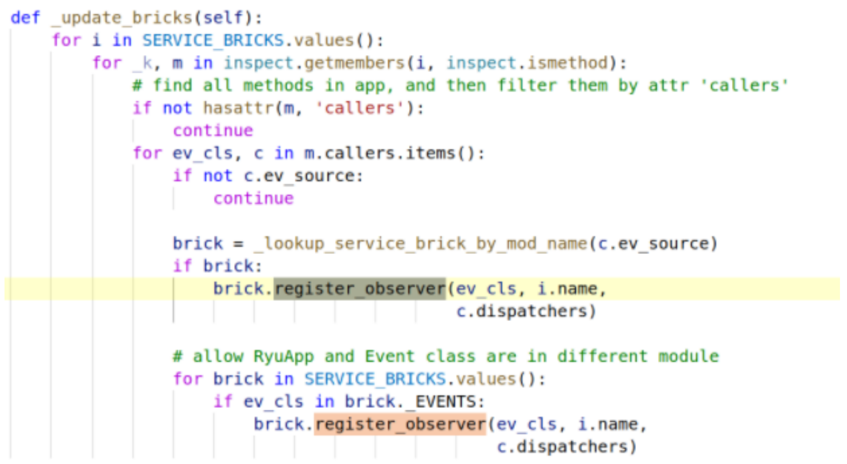
This step follows the previous one and is responsible for completing the **registration**, which is at the heart of Ryu. The reason why we can receive the corresponding type of event in the app and process it with our custom event handler is because we have completed two different types of registration during the startup process.

> `ofp_handler` is a per_designed app by ryu, it defines all the basic events and all the other apps depend on it for their work. As we will see later, it is used as a **message source** for all other apps

- App registers with the `ofp_handler`

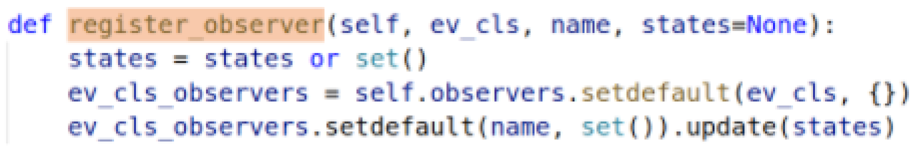
  We can think of this process as an app subscribing to an event to `ofp_handler`. `ofp_handler` has an attribute named `observers`.

  Our app call `_update_bricks` after load process



```
def _update_bricks(self):
    for i in SERVICE_BRICKS.values():
        for _k, m in inspect.getmembers(i, inspect.ismethod):
            # find all methods in app, and then filter them by attr 'callers'
            if not hasattr(m, 'callers'):
                continue
            for ev_cls, c in m.callers.items():
                if not c.ev_source:
                    continue

                brick = _lookup_service_brick_by_mod_name(c.ev_source)
                if brick:
                    brick.register_observer(ev_cls, i.name,
                                            c.dispatchers)

                # allow RyuApp and Event class are in different module
                for brick in SERVICE_BRICKS.values():
                    if ev_cls in brick._EVENTS:
                        brick.register_observer(ev_cls, i.name,
                                                c.dispatchers)
```

  and `register_observer` function :

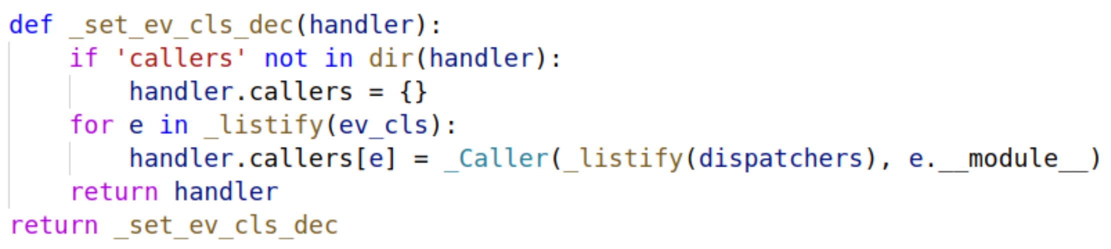


```
def register_observer(self, ev_cls, name, states=None):
    states = states or set()
    ev_cls_observers = self.observers.setdefault(ev_cls, {})
    ev_cls_observers.setdefault(name, set()).update(states)
```

  This work allows us to distribute events received from ovs-Switch of the corresponding type to specific apps when we receive them.

- The handler function registers with the app containing it

  Every handler function has an attribute named `callers` , it is assigned by the decorator `set_ev_cls`.

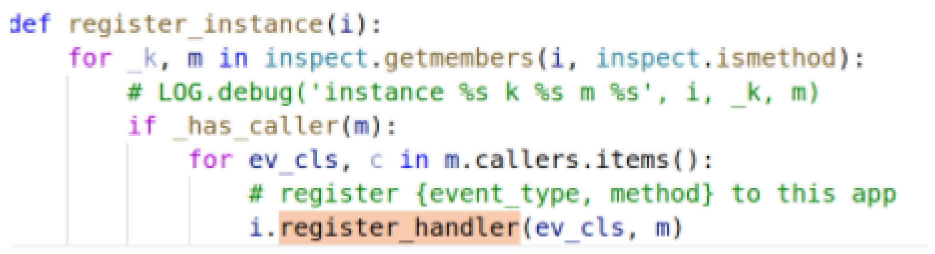


```
def _set_ev_cls_dec(handler):
    if 'callers' not in dir(handler):
        handler.callers = {}
    for e in _listify(ev_cls):
        handler.callers[e] = _Caller(_listify(dispatchers), e.__module__)
    return handler
return _set_ev_cls_dec
```

  Each app has an data member `self.event_handlers`, it is a dict like :

```
event_handlers[event_i] = {handler1, handler2}
```

  In the initilization process, we call `register_instance` to regist the handler to app by its `caller` attribute.



```
def register_instance(i):
    for _k, m in inspect.getmembers(i, inspect.ismethod):
        # LOG.debug('instance %s k %s m %s', i, _k, m)
        if _has_caller(m):
            for ev_cls, c in m.callers.items():
                # register {event_type, method} to this app
                i.register_handler(ev_cls, m)
```

The purpose of this step is to distribute the event to the Hander that handles the corresponding event according to the event type when the app receives the event sent by `ofp_handler`.

And after finish these steps, each app went to `_event_loop` , that is, polling message queues to distribute events.

```python
# app_manager.py
def _event_loop(self):
    while self.is_active or not self.events.empty():
    ev, state = self.events.get()
    if ev == self._event_stop:
        continue
    handlers = self.get_handlers(ev, state)
    for handler in handlers:
        handler(ev)
```

3) Establish a TCP connection with the Switch

- process

  `OpenFlowController.__call__()`

  -> `server_loop()`

  -> `StreamServer`

  -> `serve_forever()`

  -> listen for socket connection requests from switch and create a new thread to accept socket connection

  -> `datapath_connection_factory` (Instantiate a datapath object)

  -> `datapath.serve()`

```python
def serve(self):
        send_thr = hub.spawn(self._send_loop)
        hello = self.ofproto_parser.OFPHello(self)
        self.send_msg(hello)                    # send hello message
immediately
        try:
            self._recv_loop()                   # receive msg
        finally:
            hub.kill(send_thr)
            hub.joinall([send_thr])
```

4) Receive messages from the switch and distribute them

The message is captured by `_recv_loop()` function.

```
# core code of _recv_loop() function
                self.ofp_brick.send_event_to_observers(ev, self.state)
                dispatchers = lambda x: x.callers[ev.__class__].dispatchers
                handlers = [handler for handler in
                            self.ofp_brick.get_handlers(ev) if
                            self.state in dispatchers(handler)]
                for handler in handlers:
                    handler(ev)
```

We can clearly see that all events are sent to other apps through `ofp_handler` app. So we find that ryu adopted the message source design pattern.

5) Message sent

Each datapath maintains a fixed depth send queue. When we call `send_msg` in our app, the message is serialized and then add to the send queue, and waiting for `send_loop` method to send it finally.

```
def _send_loop(self):
        try:
            while self.state != DEAD_DISPATCHER:
                buf, close_socket = self.send_q.get()
                self._send_q_sem.release()
                self.socket.sendall(buf)
                if close_socket:
                    break
        # Omit some code
def send_msg(self, msg, close_socket=False):
        assert isinstance(msg, self.ofproto_parser.MsgBase)
        if msg.xid is None:
            self.set_xid(msg)
        msg.serialize()                                 # serialize the msg
        return self.send(msg.buf, close_socket=close_socket)
def send(self, buf, close_socket=False):
        msg_enqueued = False
        self._send_q_sem.acquire()
        if self.send_q:
            self.send_q.put((buf, close_socket))    # add the message to
send queue
            msg_enqueued = True
        else:
            self._send_q_sem.release()
        return msg_enqueued
```

6) Summary

Message distribution in Ryu uses a subscription and push mechanism, and the message push is app as minimal **granularity** rather than the handler function. And after it is pushed to the app, it is then distributed to the handler through polling mode.

3. Network Topology

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

```python
# ryu_lab_mininet.py
def NetworkTopo():
    net = Mininet()
    info("Create host nodes.\n")
    h1 = net.addHost("h1")
    h2 = net.addHost("h2")
    info("Create switch node.\n")
    s1 = net.addSwitch("switch_1", dpid='1', switch=OVSSwitch, protocols=OF_PROTOCOL, failMode="secure")
    s2 = net.addSwitch("switch_2", dpid='2', switch=OVSSwitch, protocols=OF_PROTOCOL, failMode="secure")
    s3 = net.addSwitch("switch_3", dpid='3', switch=OVSSwitch, protocols=OF_PROTOCOL, failMode="secure")
    s4 = net.addSwitch("switch_4", dpid='4', switch=OVSSwitch, protocols=OF_PROTOCOL, failMode="secure")

    info("Connect to controller node.\n")
    net.addController(name='ryu_c1',controller=RemoteController,ip='127.0.0.1',port=6653)

    info("Create Links.\n")
    net.addLink(h1, s1)
    net.addLink(s1, s3)
    net.addLink(s1, s4)
    net.addLink(s2, s3)
    net.addLink(s2, s4)
    net.addLink(s2, h2)

    info("build and start.\n")
    net.build()
    net.start()
    CLI(net)
```

4. A Tool App

> For more details, Please see the `ryu_get_network_topo.py`

Consider ryu's ability to instantiate other apps in one APP, I decided to implement the functions that will be used in the next three apps and the additional functions I added in a separate APP, named `ryu_get_network_topo.py`. And then instantiate it directly in other apps to improve my code reuse.

In this APP, I implement following functions :

- Set up a table-miss match flow entry, which is sent to the controller (in function `switch_feature_handler`)
- Obtain the topology of the entire network at this time
- Process received ARP packets (Perform Mac learning, flooding, and ARP storm processing)
- Draw the network topology and the current packet transmission path using `networkx` lib (Mainly for Problem 1)

  > You can see the renderings later

5. Problem_1 : Path Switch

> 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.

Solving Ideas

Get network Topo

I use the API provided by Ryu to capture the network topology in real time (by `lldp` packets), mainly :

`get_link` and `get_switch` , and obtain host connection information by analyzing `ARP` packets by ourself.

Mainly data structures uesd :

```
# ryu_get_network_topo.py
self.portsOfSwitch = {}            # {dpid -> [ports]}
self.allSwitches = set()           # (dpid)
self.onlineSwitches = set()        # (dpid)
self.linksBetweenSwitch = {}       # {(src_dpid, port) : {dst_dpid, port}}
self.linksToHost = {}              # {(src_dpid, port) : host_ip}
self.hostToSwitch = {}             # {host_ip : [(src_dpid, port)]}
self.hosts = {}                    # {host_ip -> mac}
self.multicast_record = {}         # {(dpid, src_ip, dst_ip) : in_port}
self.mac_to_port = {}              # {dpid : {mac : port}}
```

Mainly four functions :

```
# ryu_get_network_topo.py
# Obtain new link information whenever the network structure changes
@set_ev_cls(switch_modify_event_list, None)
regetNetTopo(self, event):

# Assign `onlineSwitches` `allSwitches` `portsOfSwitch` `portsOfSwitch`
constructor(self, switches, links)

# Get Host-Switch information
handle_arp_packet(self, msg)

# Assign `hosts` `linksToHost` `hostToSwitch`
host_register(self, ip, mac, datapath, port)
```

Set timeout

To achieve the goal of switching a path every 5 seconds，I set the `hard_time` for the flow table to 5s. That is, all flow tables related to the delivered path become invalid every 5s.

So every five seconds, the controller will get table-miss messages, and then I select a new path and deliver the flow table.

And at the same time, the flow table may be out of sync due to network latency and different times of distribution，I set the priority of the newly delivered flow table to increase by one to solve this problem to some extent.

Find a new path

Considering I can get the network topology in real time, So I used **DFS** to find new path.

The data structure that saves the path:

```
# SDN_Lab_P1.py
self.path = []                  # [(dpid, in_port, out_port)]
self.pathForCompare = []        # [(dpid, in_port)]
```

Because that source and destination nodes are all hosts which we can resolve it from packet received in advance, I found that a routing path can be uniquely defined by a list of `[switch + in_port + out_port]`, here the switch is the switch through which packet passes in the whole transmission process.

Besides, the `pathForCompare` is used to make sure we get a different path.

```
# SDN_Lab_P1.py/get_new_path(self, src_ip, dst_ip)
# DFS process (Only the core code)
        first_hop = hostToSwitch[src_ip]
        stack = []
        stack.append(first_hop)
        children_node = {}              # {dpid, []}
        new_path = []                   # record the path
        visited = {}                    # {dpid : bool}

        while (len(stack)):
```

```python
            print("stack:  ",stack)
            temp_node = stack[-1]       # (dpid, in_port)
            if not children_node.__contains__(temp_node[0]):
                children_node[temp_node[0]] = []
            visited[temp_node[0]] = True
            if temp_node[0] == dst_ip:
                if self.pathForCompare != stack:        # Make sure to choose a different a path
                    break

            Flag = False
            if temp_node[0] != dst_ip:
                for out_port in portsOfSwitch[temp_node[0]]:
                    if out_port == temp_node[1]:
                        continue
                    if linksBetweenSwitch.__contains__((temp_node[0], out_port)):
                        next_hop = linksBetweenSwitch[(temp_node[0], out_port)]
                    elif linksToHost.__contains__((temp_node[0], out_port)):
                        next_hop = (linksToHost[(temp_node[0], out_port)], 0)
                    if not visited. __contains__(next_hop[0]) or visited[next_hop[0]] == False:
                        stack.append(next_hop)
                        children_node[temp_node[0]].append(next_hop[0])
                        Flag = True
                        break

            if not Flag:
                stack.pop(-1)
                for switch in children_node[temp_node[0]]:
                    visited[switch] = False

        if len(stack) == 0:
            print("No Usable Path")
            return False
```

Distributed flow table

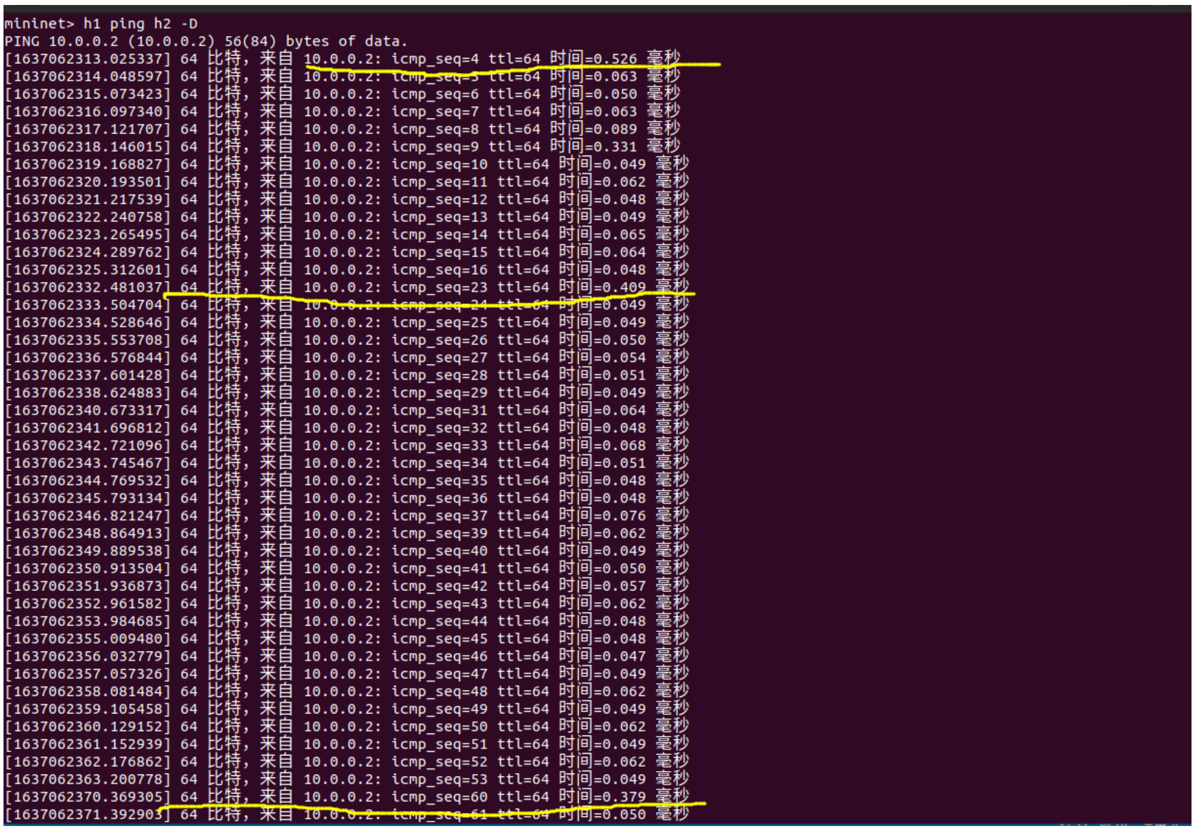
Add a flow table for each switch based on the calculated path.

```python
def set_new_flow_tables(self, src_ip, dst_ip):
        while not self.get_new_path(src_ip, dst_ip):
            self.get_new_path(src_ip, dst_ip)

        priority = self.priority + 1
        for node in self.path:
            dpid = node[0]
            in_port = node[1]
            out_port = node[2]
            datapath = self.datapaths[dpid]
            ofp_parser = datapath.ofproto_parser
            # A -> B
            actions_1 = [ofp_parser.OFPActionOutput(out_port)]
            match_1 = ofp_parser.OFPMatch(in_port=in_port, eth_type=0x0800, ipv4_src=src_ip, ipv4_dst=dst_ip)
            self.add_flow(datapath, _priority=priority, _match=match_1, _actions=actions_1, _hard_time=self.INTERVAL)

            # B -> A
            actions_2 = [ofp_parser.OFPActionOutput(in_port)]
            match_2 = ofp_parser.OFPMatch(in_port=out_port, eth_type=0x0800, ipv4_src=dst_ip, ipv4_dst=src_ip)
            self.add_flow(datapath, _priority=priority, _match=match_2, _actions=actions_2, _hard_time=self.INTERVAL)

        self.priority = priority
        path_for_draw = [(src_ip, 0, 0)] + self.path + [(dst_ip, 0, 0)]
        self.draw_topo(path_for_draw)                      # Real-time path visualization
```

Testing Result

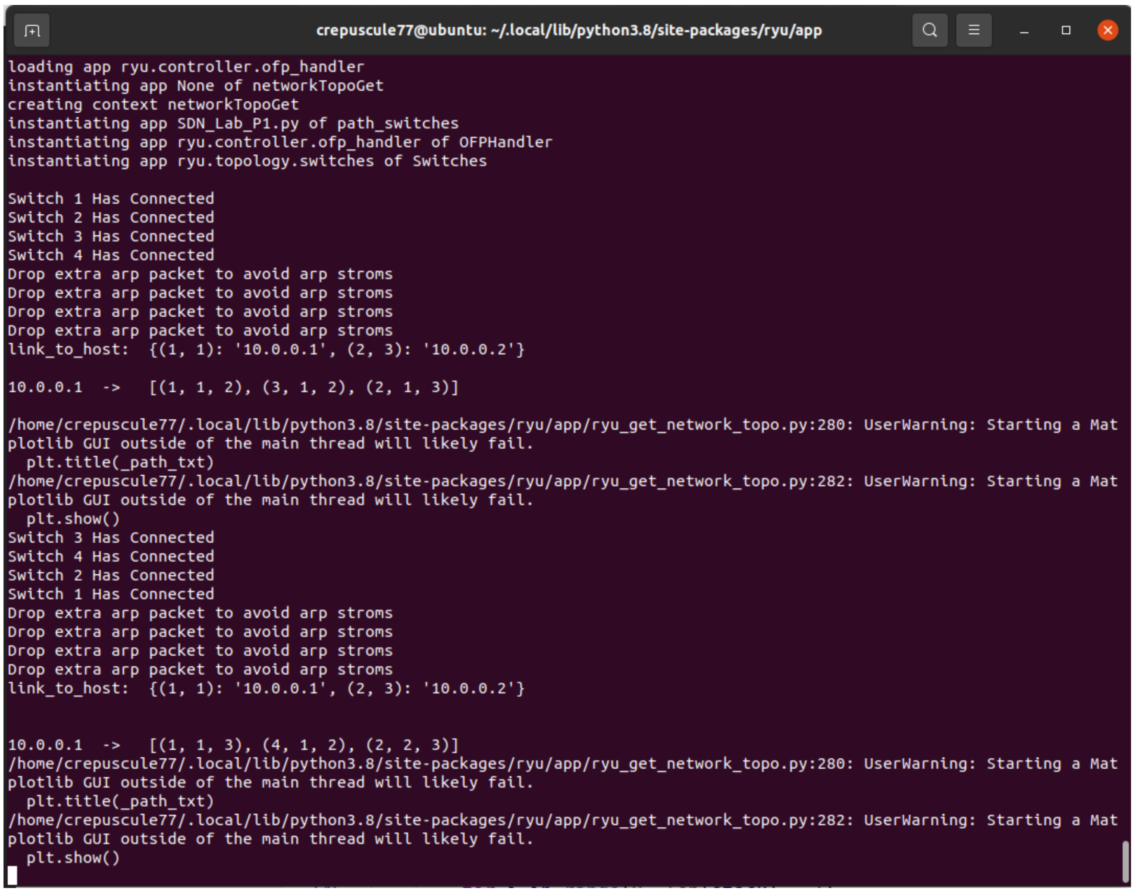
- Mininet CLI

> We can obviously see that every once in a while, several packets are lost and the time interval suddenly increases.
>
> And this is the moment to switch paths.



```
mininet> h1 ping h2 -D
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
[1637062313.025337] 64 比特，来自 10.0.0.2: icmp_seq=4 ttl=64 时间=0.526 毫秒
[1637062314.048597] 64 比特，来自 10.0.0.2: icmp_seq=5 ttl=64 时间=0.063 毫秒
[1637062315.073423] 64 比特，来自 10.0.0.2: icmp_seq=6 ttl=64 时间=0.050 毫秒
[1637062316.097340] 64 比特，来自 10.0.0.2: icmp_seq=7 ttl=64 时间=0.063 毫秒
[1637062317.121707] 64 比特，来自 10.0.0.2: icmp_seq=8 ttl=64 时间=0.089 毫秒
[1637062318.146015] 64 比特，来自 10.0.0.2: icmp_seq=9 ttl=64 时间=0.331 毫秒
[1637062319.168827] 64 比特，来自 10.0.0.2: icmp_seq=10 ttl=64 时间=0.049 毫秒
[1637062320.193501] 64 比特，来自 10.0.0.2: icmp_seq=11 ttl=64 时间=0.062 毫秒
[1637062321.217539] 64 比特，来自 10.0.0.2: icmp_seq=12 ttl=64 时间=0.048 毫秒
[1637062322.240758] 64 比特，来自 10.0.0.2: icmp_seq=13 ttl=64 时间=0.049 毫秒
[1637062323.265495] 64 比特，来自 10.0.0.2: icmp_seq=14 ttl=64 时间=0.065 毫秒
[1637062324.289762] 64 比特，来自 10.0.0.2: icmp_seq=15 ttl=64 时间=0.064 毫秒
[1637062325.312601] 64 比特，来自 10.0.0.2: icmp_seq=16 ttl=64 时间=0.048 毫秒
[1637062332.481037] 64 比特，来自 10.0.0.2: icmp_seq=23 ttl=64 时间=0.409 毫秒
[1637062333.504704] 64 比特，来自 10.0.0.2: icmp_seq=24 ttl=64 时间=0.049 毫秒
[1637062334.528646] 64 比特，来自 10.0.0.2: icmp_seq=25 ttl=64 时间=0.049 毫秒
[1637062335.553708] 64 比特，来自 10.0.0.2: icmp_seq=26 ttl=64 时间=0.050 毫秒
[1637062336.576844] 64 比特，来自 10.0.0.2: icmp_seq=27 ttl=64 时间=0.054 毫秒
[1637062337.601428] 64 比特，来自 10.0.0.2: icmp_seq=28 ttl=64 时间=0.051 毫秒
[1637062338.624883] 64 比特，来自 10.0.0.2: icmp_seq=29 ttl=64 时间=0.049 毫秒
[1637062340.673317] 64 比特，来自 10.0.0.2: icmp_seq=31 ttl=64 时间=0.064 毫秒
[1637062341.696812] 64 比特，来自 10.0.0.2: icmp_seq=32 ttl=64 时间=0.048 毫秒
[1637062342.721096] 64 比特，来自 10.0.0.2: icmp_seq=33 ttl=64 时间=0.068 毫秒
[1637062343.745467] 64 比特，来自 10.0.0.2: icmp_seq=34 ttl=64 时间=0.051 毫秒
[1637062344.769532] 64 比特，来自 10.0.0.2: icmp_seq=35 ttl=64 时间=0.048 毫秒
[1637062345.793134] 64 比特，来自 10.0.0.2: icmp_seq=36 ttl=64 时间=0.048 毫秒
[1637062346.821247] 64 比特，来自 10.0.0.2: icmp_seq=37 ttl=64 时间=0.076 毫秒
[1637062348.864913] 64 比特，来自 10.0.0.2: icmp_seq=39 ttl=64 时间=0.062 毫秒
[1637062349.889538] 64 比特，来自 10.0.0.2: icmp_seq=40 ttl=64 时间=0.049 毫秒
[1637062350.913504] 64 比特，来自 10.0.0.2: icmp_seq=41 ttl=64 时间=0.050 毫秒
[1637062351.936873] 64 比特，来自 10.0.0.2: icmp_seq=42 ttl=64 时间=0.057 毫秒
[1637062352.961582] 64 比特，来自 10.0.0.2: icmp_seq=43 ttl=64 时间=0.062 毫秒
[1637062353.984685] 64 比特，来自 10.0.0.2: icmp_seq=44 ttl=64 时间=0.048 毫秒
[1637062355.009480] 64 比特，来自 10.0.0.2: icmp_seq=45 ttl=64 时间=0.048 毫秒
[1637062356.032779] 64 比特，来自 10.0.0.2: icmp_seq=46 ttl=64 时间=0.047 毫秒
[1637062357.057326] 64 比特，来自 10.0.0.2: icmp_seq=47 ttl=64 时间=0.049 毫秒
[1637062358.081484] 64 比特，来自 10.0.0.2: icmp_seq=48 ttl=64 时间=0.062 毫秒
[1637062359.105458] 64 比特，来自 10.0.0.2: icmp_seq=49 ttl=64 时间=0.049 毫秒
[1637062360.129152] 64 比特，来自 10.0.0.2: icmp_seq=50 ttl=64 时间=0.062 毫秒
[1637062361.152939] 64 比特，来自 10.0.0.2: icmp_seq=51 ttl=64 时间=0.049 毫秒
[1637062362.176862] 64 比特，来自 10.0.0.2: icmp_seq=52 ttl=64 时间=0.062 毫秒
[1637062363.200778] 64 比特，来自 10.0.0.2: icmp_seq=53 ttl=64 时间=0.049 毫秒
[1637062370.369305] 64 比特，来自 10.0.0.2: icmp_seq=60 ttl=64 时间=0.379 毫秒
[1637062371.392903] 64 比特，来自 10.0.0.2: icmp_seq=61 ttl=64 时间=0.050 毫秒
```

- Ryu CLI

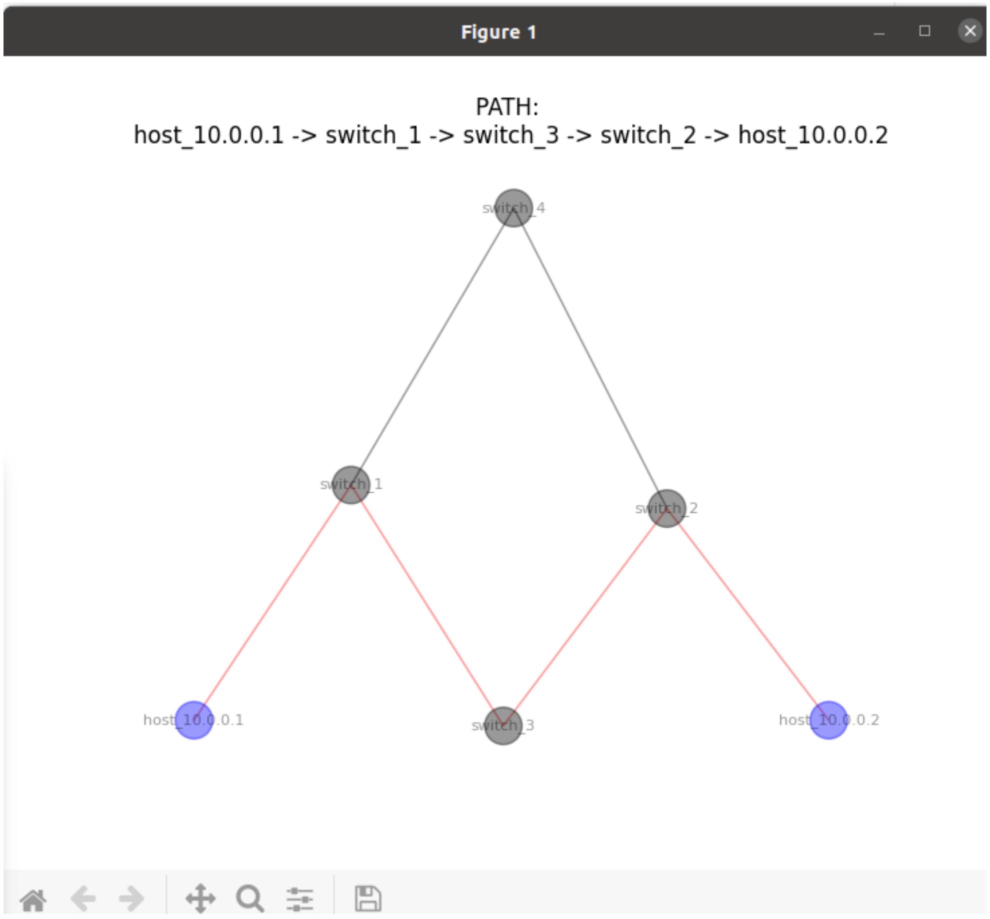


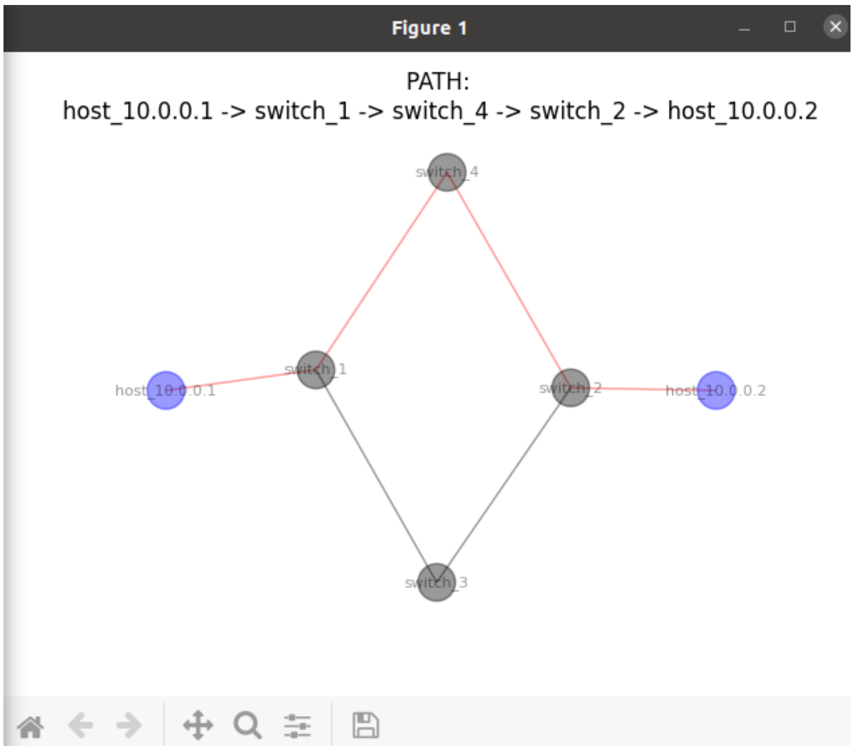
crepuscule77@ubuntu: ~/.local/lib/python3.8/site-packages/ryu/app

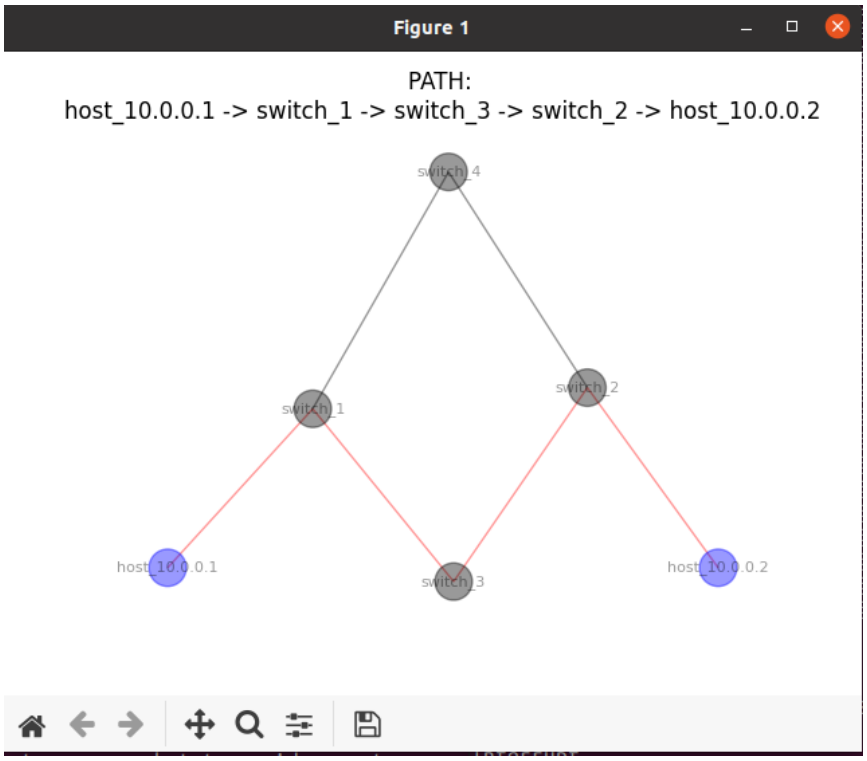
```
loading app ryu.controller.ofp_handler
instantiating app None of networkTopoGet
creating context networkTopoGet
instantiating app SDN_Lab_P1.py of path_switches
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu.topology.switches of Switches

Switch 1 Has Connected
Switch 2 Has Connected
Switch 3 Has Connected
Switch 4 Has Connected
Drop extra arp packet to avoid arp stroms
Drop extra arp packet to avoid arp stroms
Drop extra arp packet to avoid arp stroms
Drop extra arp packet to avoid arp stroms
link_to_host:  {(1, 1): '10.0.0.1', (2, 3): '10.0.0.2'}

10.0.0.1  ->   [(1, 1, 2), (3, 1, 2), (2, 1, 3)]

/home/crepuscule77/.local/lib/python3.8/site-packages/ryu/app/ryu_get_network_topo.py:280: UserWarning: Starting a Mat
plotlib GUI outside of the main thread will likely fail.
  plt.title(_path_txt)
/home/crepuscule77/.local/lib/python3.8/site-packages/ryu/app/ryu_get_network_topo.py:282: UserWarning: Starting a Mat
plotlib GUI outside of the main thread will likely fail.
  plt.show()
Switch 3 Has Connected
Switch 4 Has Connected
Switch 2 Has Connected
Switch 1 Has Connected
Drop extra arp packet to avoid arp stroms
Drop extra arp packet to avoid arp stroms
Drop extra arp packet to avoid arp stroms
Drop extra arp packet to avoid arp stroms
link_to_host:  {(1, 1): '10.0.0.1', (2, 3): '10.0.0.2'}


10.0.0.1  ->   [(1, 1, 3), (4, 1, 2), (2, 2, 3)]
/home/crepuscule77/.local/lib/python3.8/site-packages/ryu/app/ryu_get_network_topo.py:280: UserWarning: Starting a Mat
plotlib GUI outside of the main thread will likely fail.
  plt.title(_path_txt)
/home/crepuscule77/.local/lib/python3.8/site-packages/ryu/app/ryu_get_network_topo.py:282: UserWarning: Starting a Mat
plotlib GUI outside of the main thread will likely fail.
  plt.show()
```

- Roadmap with `networkx`

> Red lines represent the current path.
>
> We can see that we're constantly switching between the top path and the bottom path.




PATH:
host_10.0.0.1 -> switch_1 -> switch_4 -> switch_2 -> host_10.0.0.2


PATH:
host_10.0.0.1 -> switch_1 -> switch_3 -> switch_2 -> host_10.0.0.2

6. Problem 2 : Load Balance

> Write a RYU controller that uses both paths to forward packets from h1 to h2.

Solving Ideas

We use group table with `OFPGT_SELECT` type to achieve it.

First we should set two buckets, and assign some weight to each of them. Each bucket represents a set of actions.

```python
# SDN_Lab_P2.py/switch_feature_handler
port_2 = 2
weight_for_port_2 = 50
actions_to_port_2 = [ofp_parser.OFPActionOutput(port_2)]

# port_3
port_3 = 3
weight_for_port_3 = 50
actions_to_port_3 = [ofp_parser.OFPActionOutput(port_3)]

# Will not be used in select group
watch_port = ofproto_v1_3.OFPP_ANY
watch_group = ofproto_v1_3.OFPQ_ALL

buckets = [
    ofp_parser.OFPBucket(weight_for_port_2, watch_port, watch_group, actions_to_port_2),
    ofp_parser.OFPBucket(weight_for_port_3, watch_port, watch_group, actions_to_port_3)
]
```

Then, we use `OFPGroupMod` to add a group for a switch and set a match rule for it

```python
_group_id = 77
msg = ofp_parser.OFPGroupMod(datapath, ofproto.OFPGC_ADD, ofproto.OFPGT_SELECT, _group_id, buckets)
datapath.send_msg(msg)

# add flow entry make group_table apply to packets from host_1
actions = [ofp_parser.OFPActionGroup(group_id=77)]
match = ofp_parser.OFPMatch(in_port=1, eth_type=0x0800, ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
self.add_flow(datapath, 10, match, actions)
```

And then do the same for Switch 2.

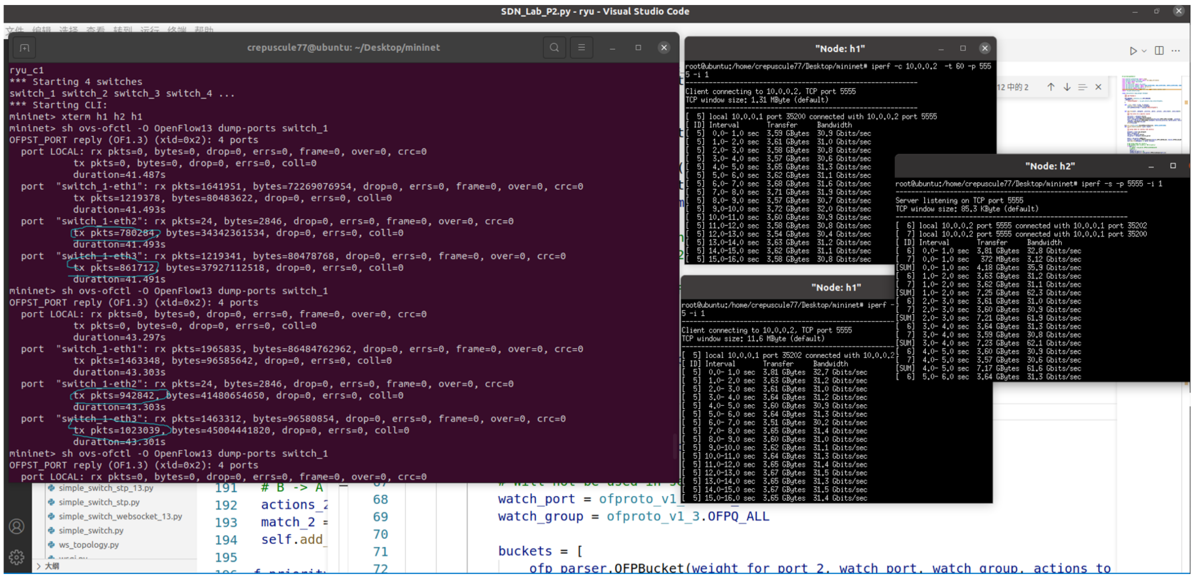
Testing Result

I use `iperf` to maintain continuous data transfer between H1 and H2.

H1 as client and H2 as server.

And then we use `sh ovs-ofctl -O OpenFlow13 dump-ports  switch_1` command to check port packet statistics.

- Screenshot



```
ryu_c1
*** Starting 4 switches
switch_1 switch_2 switch_3 switch_4 ...
*** Starting CLI:
mininet> xterm h1 h2 h1
mininet> sh ovs-ofctl -O OpenFlow13 dump-ports switch_1
OFPST_PORT reply (OF1.3) (xid=0x2): 4 ports
  port LOCAL: rx pkts=0, bytes=0, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=0, bytes=0, drop=0, errs=0, coll=0
           duration=41.487s
  port  "switch_1-eth1": rx pkts=1641951, bytes=72269076954, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=1219378, bytes=80483622, drop=0, errs=0, coll=0
           duration=41.493s
  port  "switch_1-eth2": rx pkts=24, bytes=2846, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=780284, bytes=34342361534, drop=0, errs=0, coll=0
           duration=41.493s
  port  "switch_1-eth3": rx pkts=1219341, bytes=80478768, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=861712, bytes=37927112518, drop=0, errs=0, coll=0
           duration=41.491s
mininet> sh ovs-ofctl -O OpenFlow13 dump-ports switch_1
OFPST_PORT reply (OF1.3) (xid=0x2): 4 ports
  port LOCAL: rx pkts=0, bytes=0, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=0, bytes=0, drop=0, errs=0, coll=0
           duration=43.297s
  port  "switch_1-eth1": rx pkts=1965835, bytes=86484762962, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=1463348, bytes=96585642, drop=0, errs=0, coll=0
           duration=43.303s
  port  "switch_1-eth2": rx pkts=24, bytes=2846, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=942842, bytes=41480654650, drop=0, errs=0, coll=0
           duration=43.303s
  port  "switch_1-eth3": rx pkts=1463312, bytes=96580854, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=1023039, bytes=45004441820, drop=0, errs=0, coll=0
           duration=43.301s
mininet> sh ovs-ofctl -O OpenFlow13 dump-ports switch_1
OFPST_PORT reply (OF1.3) (xid=0x2): 4 ports
  port LOCAL: rx pkts=0, bytes=0, drop=0, errs=0, frame=0, over=0, crc=0
```

```
root@ubuntu:/home/crepuscule77/Desktop/mininet# iperf -c 10.0.0.2  -t 60 -p 555
5 -i 1
------------------------------------------------------------
Client connecting to 10.0.0.2, TCP port 5555
TCP window size: 1.31 MByte (default)
------------------------------------------------------------
[  5] local 10.0.0.1 port 35200 connected with 10.0.0.2 port 5555
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0- 1.0 sec  3.59 GBytes  30.9 Gbits/sec
[  5]  1.0- 2.0 sec  3.61 GBytes  31.0 Gbits/sec
[  5]  2.0- 3.0 sec  3.58 GBytes  30.8 Gbits/sec
[  5]  3.0- 4.0 sec  3.57 GBytes  30.6 Gbits/sec
[  5]  4.0- 5.0 sec  3.65 GBytes  31.3 Gbits/sec
[  5]  5.0- 6.0 sec  3.62 GBytes  31.1 Gbits/sec
[  5]  6.0- 7.0 sec  3.68 GBytes  31.6 Gbits/sec
[  5]  7.0- 8.0 sec  3.71 GBytes  31.9 Gbits/sec
[  5]  8.0- 9.0 sec  3.57 GBytes  30.7 Gbits/sec
[  5]  9.0-10.0 sec  3.72 GBytes  32.0 Gbits/sec
[  5] 10.0-11.0 sec  3.60 GBytes  30.9 Gbits/sec
[  5] 11.0-12.0 sec  3.58 GBytes  30.8 Gbits/sec
[  5] 12.0-13.0 sec  3.54 GBytes  30.4 Gbits/sec
[  5] 13.0-14.0 sec  3.63 GBytes  31.2 Gbits/sec
[  5] 14.0-15.0 sec  3.62 GBytes  31.1 Gbits/sec
[  5] 15.0-16.0 sec  3.58 GBytes  30.8 Gbits/sec
```

```
root@ubuntu:/home/crepuscule77/Desktop/mininet# iperf -s -p 5555 -i 1
------------------------------------------------------------
Server listening on TCP port 5555
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  6] local 10.0.0.2 port 5555 connected with 10.0.0.1 port 35202
[  7] local 10.0.0.2 port 5555 connected with 10.0.0.1 port 35200
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0- 1.0 sec  3.81 GBytes  32.8 Gbits/sec
[  7]  0.0- 1.0 sec   372 MBytes  3.12 Gbits/sec
[SUM]  0.0- 1.0 sec  4.18 GBytes  35.9 Gbits/sec
[  6]  1.0- 2.0 sec  3.63 GBytes  31.2 Gbits/sec
[  7]  1.0- 2.0 sec  3.62 GBytes  31.1 Gbits/sec
[SUM]  1.0- 2.0 sec  7.25 GBytes  62.3 Gbits/sec
[  6]  2.0- 3.0 sec  3.61 GBytes  31.0 Gbits/sec
[  7]  2.0- 3.0 sec  3.60 GBytes  30.9 Gbits/sec
[SUM]  2.0- 3.0 sec  7.21 GBytes  61.9 Gbits/sec
[  6]  3.0- 4.0 sec  3.64 GBytes  31.3 Gbits/sec
[  7]  3.0- 4.0 sec  3.59 GBytes  30.8 Gbits/sec
[SUM]  3.0- 4.0 sec  7.23 GBytes  62.1 Gbits/sec
[  6]  4.0- 5.0 sec  3.60 GBytes  30.9 Gbits/sec
[  7]  4.0- 5.0 sec  3.57 GBytes  30.6 Gbits/sec
[SUM]  4.0- 5.0 sec  7.17 GBytes  61.6 Gbits/sec
[  6]  5.0- 6.0 sec  3.64 GBytes  31.3 Gbits/sec
```

```
root@ubuntu:/home/crepuscule77/Desktop/mininet# iperf -
5 -i 1
------------------------------------------------------------
Client connecting to 10.0.0.2, TCP port 5555
TCP window size: 11.6 MByte (default)
------------------------------------------------------------
[  5] local 10.0.0.1 port 35202 connected with 10.0.0.2
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0- 1.0 sec  3.81 GBytes  32.7 Gbits/sec
[  5]  1.0- 2.0 sec  3.63 GBytes  31.2 Gbits/sec
[  5]  2.0- 3.0 sec  3.61 GBytes  31.0 Gbits/sec
[  5]  3.0- 4.0 sec  3.64 GBytes  31.2 Gbits/sec
[  5]  4.0- 5.0 sec  3.60 GBytes  30.9 Gbits/sec
[  5]  5.0- 6.0 sec  3.64 GBytes  31.3 Gbits/sec
[  5]  6.0- 7.0 sec  3.51 GBytes  30.2 Gbits/sec
[  5]  7.0- 8.0 sec  3.65 GBytes  31.4 Gbits/sec
[  5]  8.0- 9.0 sec  3.60 GBytes  31.0 Gbits/sec
[  5]  9.0-10.0 sec  3.62 GBytes  31.1 Gbits/sec
[  5] 10.0-11.0 sec  3.64 GBytes  31.3 Gbits/sec
[  5] 11.0-12.0 sec  3.65 GBytes  31.4 Gbits/sec
[  5] 12.0-13.0 sec  3.67 GBytes  31.5 Gbits/sec
[  5] 13.0-14.0 sec  3.65 GBytes  31.3 Gbits/sec
[  5] 14.0-15.0 sec  3.67 GBytes  31.5 Gbits/sec
[  5] 15.0-16.0 sec  3.65 GBytes  31.4 Gbits/sec
```

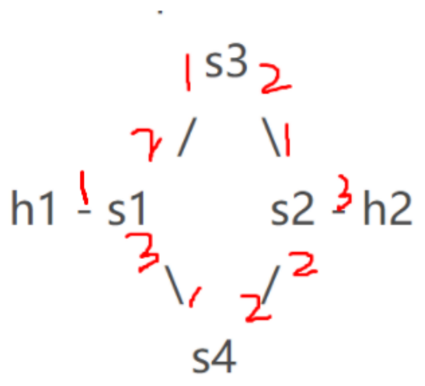
We can easily find that when host_1 sends data to host_2， the two outgoing ports of Switch 1 are allocated almost the same number of packets, and both paths are being used at the same time. (Here we set the weights of both buckets to 50%)

7. Problem_3 : Fast Failover

Solving ideas

We use group table with `OFPGT_FF` to achieve it.

For the topology below, we define flow rules as follows :



```
# Flow rules
Switch_1:
① when port 2 of switch_1 down, we choose to send packets by port 3.
  when port 3 of switch_1 down, we choose to send packets by port 2.

Switch_2:
② when port 1 of switch_2 down, we choose to send packets by port 2.
  when port 2 of switch_2 down, we choose to send packets by port 1.
  for packets from port 1, if dst_ip == h2, send to port 3; if dst_ip == h1, send to port_2
  for packets from port 2, if dst_ip == h2, send to port 3; if dst_ip == h1, send to port_1

Switch_3:
```

```
③ when port 1 of switch_3 down, we choose to send packets by port 2.
  when port 2 of switch_3 down, we choose to send packets by port 1.

Switch_4:
④ when port 1 of switch_4 down, we choose to send packets by port 2.
  when port 2 of switch_4 down, we choose to send packets by port 1.
```

Core Code

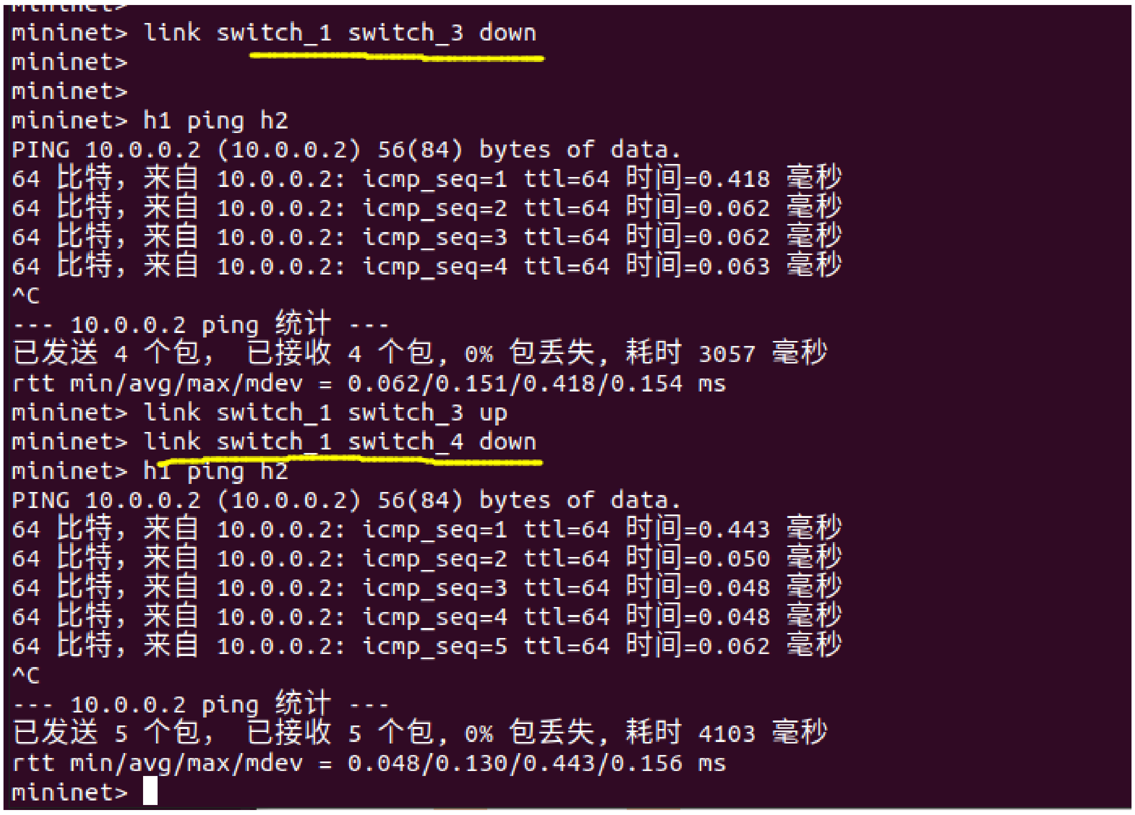
```
# SDN_Lab_P3.py/switch_feature_handler
    if dpid == 1:
        weight = 0
        # port_2
        port_2 = 2
        actions_to_port_2 = [ofp_parser.OFPActionOutput(port_2)]

        # port_3
        port_3 = 3
        actions_to_port_3 = [ofp_parser.OFPActionOutput(port_3)]

        watch_group = ofproto_v1_3.OFPQ_ALL

        buckets = [
            ofp_parser.OFPBucket(weight, port_2, watch_group, actions_to_port_2),
            ofp_parser.OFPBucket(weight, port_3, watch_group, actions_to_port_3)
        ]
        # fast failover
        _group_id = 77
        msg = ofp_parser.OFPGroupMod(datapath, ofproto.OFPGC_ADD, ofproto.OFPGT_FF, _group_id, buckets)
        datapath.send_msg(msg)
        sleep(1)

        actions = [ofp_parser.OFPActionGroup(group_id=77)]
        match = ofp_parser.OFPMatch(in_port=1, eth_type=0x0800, ipv4_src="10.0.0.1")
        self.add_flow(datapath, 10, match, actions)

        # #add the return flow for h1 in s1.
        actions = [ofp_parser.OFPActionOutput(1)]
        match = ofp_parser.OFPMatch(in_port=2)
        self.add_flow(datapath, 10, match, actions)

        actions = [ofp_parser.OFPActionOutput(1)]
        match = ofp_parser.OFPMatch(in_port=3)
        self.add_flow(datapath, 10, match, actions)

    elif dpid == 2:
        port_1 = 1
        port_2 = 2
        port_3 = 3
        weight = 0

```

```python
        actions_to_port_1 = [ofp_parser.OFPActionOutput(port_1)]
        actions_to_port_2 = [ofp_parser.OFPActionOutput(port_2)]

        watch_group = ofproto_v1_3.OFPQ_ALL

        buckets = [
            ofp_parser.OFPBucket(weight, port_1, watch_group, actions_to_port_1),
            ofp_parser.OFPBucket(weight, port_2, watch_group, actions_to_port_2)
        ]
        # fast failover
        _group_id = 77
        msg = ofp_parser.OFPGroupMod(datapath, ofproto.OFPGC_ADD, ofproto.OFPGT_FF, _group_id, buckets)
        datapath.send_msg(msg)
        sleep(1)

        actions = [ofp_parser.OFPActionGroup(group_id=77)]
        match = ofp_parser.OFPMatch(in_port=port_3, eth_type=0x0800, ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
        self.add_flow(datapath, 10, match, actions)

        # normal condition
        actions = [ofp_parser.OFPActionOutput(port_3)]
        match = ofp_parser.OFPMatch(in_port=port_1, eth_type=0x0800, ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
        self.add_flow(datapath, 10, match, actions)

        actions = [ofp_parser.OFPActionOutput(port_3)]
        match = ofp_parser.OFPMatch(in_port=port_2, eth_type=0x0800, ipv4_src="10.0.0.1", ipv4_dst="10.0.0.2")
        self.add_flow(datapath, 10, match, actions)

        # we need to deal with the emergencies
        # such as there is a break in the middle of the link
        actions = [ofp_parser.OFPActionOutput(port_2)]
        match = ofp_parser.OFPMatch(in_port=port_1, eth_type=0x0800, ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
        self.add_flow(datapath, 10, match, actions)

        actions = [ofp_parser.OFPActionOutput(port_1)]
        match = ofp_parser.OFPMatch(in_port=port_2, eth_type=0x0800, ipv4_src="10.0.0.2", ipv4_dst="10.0.0.1")
        self.add_flow(datapath, 10, match, actions)

    elif dpid == 3 or dpid == 4:
        # If we want to send the packet to the input port
        # we must use ofproto_v1_3.OFPP_IN_PORT to represent the in_put port
        port_1 = 1
        port_2 = 2
        weight = 0

        actions_to_port_1_BACK = [ofp_parser.OFPActionOutput(ofproto_v1_3.OFPP_IN_PORT)]
        actions_to_port_2 = [ofp_parser.OFPActionOutput(port_2)]

        actions_to_port_1 = [ofp_parser.OFPActionOutput(port_1)]
        actions_to_port_2_BACK = [ofp_parser.OFPActionOutput(ofproto_v1_3.OFPP_IN_PORT)]

        watch_group = ofproto_v1_3.OFPQ_ALL

        buckets_for_port_1 = [
            ofp_parser.OFPBucket(weight, port_2, watch_group, actions_to_port_2),       # out to port_2
            ofp_parser.OFPBucket(weight, port_1, watch_group, actions_to_port_1_BACK)   # out to input port
        ]

        buckets_for_port_2 = [
            ofp_parser.OFPBucket(weight, port_1, watch_group, actions_to_port_1),       # out to port_1
            ofp_parser.OFPBucket(weight, port_2, watch_group, actions_to_port_2_BACK)   # out to input port
        ]

        # fast failover for port_1
        msg = ofp_parser.OFPGroupMod(datapath, ofproto.OFPGC_ADD, ofproto.OFPGT_FF, 1, buckets_for_port_1)
        datapath.send_msg(msg)
        sleep(1)

        actions = [ofp_parser.OFPActionGroup(group_id=1)]
        match = ofp_parser.OFPMatch(in_port=1, eth_type=0x0800)
        self.add_flow(datapath, 10, match, actions)

        # fast failover for port_2
        msg = ofp_parser.OFPGroupMod(datapath, ofproto.OFPGC_ADD, ofproto.OFPGT_FF, 2, buckets_for_port_2)
        datapath.send_msg(msg)
        sleep(1)

        actions = [ofp_parser.OFPActionGroup(group_id=2)]
        match = ofp_parser.OFPMatch(in_port=2, eth_type=0x0800)
        self.add_flow(datapath, 10, match, actions)


```

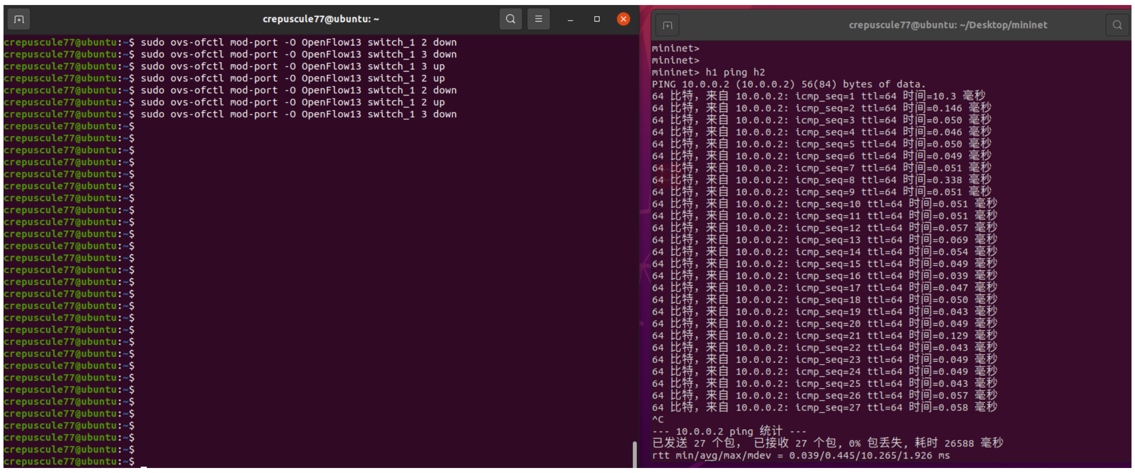
Testing Result

- We first disconnect the connection between Switch_1 and Switch_3

  And then disconnect the connection between Switch_1 and Switch_4



```
mininet>
mininet> link switch_1 switch_3 down
mininet>
mininet>
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 比特，来自 10.0.0.2: icmp_seq=1 ttl=64 时间=0.418 毫秒
64 比特，来自 10.0.0.2: icmp_seq=2 ttl=64 时间=0.062 毫秒
64 比特，来自 10.0.0.2: icmp_seq=3 ttl=64 时间=0.062 毫秒
64 比特，来自 10.0.0.2: icmp_seq=4 ttl=64 时间=0.063 毫秒
^C
--- 10.0.0.2 ping 统计 ---
已发送 4 个包， 已接收 4 个包, 0% 包丢失, 耗时 3057 毫秒
rtt min/avg/max/mdev = 0.062/0.151/0.418/0.154 ms
mininet> link switch_1 switch_3 up
mininet> link switch_1 switch_4 down
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 比特，来自 10.0.0.2: icmp_seq=1 ttl=64 时间=0.443 毫秒
64 比特，来自 10.0.0.2: icmp_seq=2 ttl=64 时间=0.050 毫秒
64 比特，来自 10.0.0.2: icmp_seq=3 ttl=64 时间=0.048 毫秒
64 比特，来自 10.0.0.2: icmp_seq=4 ttl=64 时间=0.048 毫秒
64 比特，来自 10.0.0.2: icmp_seq=5 ttl=64 时间=0.062 毫秒
^C
--- 10.0.0.2 ping 统计 ---
已发送 5 个包， 已接收 5 个包, 0% 包丢失, 耗时 4103 毫秒
rtt min/avg/max/mdev = 0.048/0.130/0.443/0.156 ms
mininet>
```



crepuscule77@ubuntu: ~

```
crepuscule77@ubuntu:~$ sudo ovs-ofctl mod-port -O OpenFlow13 switch_1 2 down
crepuscule77@ubuntu:~$ sudo ovs-ofctl mod-port -O OpenFlow13 switch_1 3 down
crepuscule77@ubuntu:~$ sudo ovs-ofctl mod-port -O OpenFlow13 switch_1 3 up
crepuscule77@ubuntu:~$ sudo ovs-ofctl mod-port -O OpenFlow13 switch_1 2 up
crepuscule77@ubuntu:~$ sudo ovs-ofctl mod-port -O OpenFlow13 switch_1 2 down
crepuscule77@ubuntu:~$ sudo ovs-ofctl mod-port -O OpenFlow13 switch_1 2 up
crepuscule77@ubuntu:~$ sudo ovs-ofctl mod-port -O OpenFlow13 switch_1 3 down
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
crepuscule77@ubuntu:~$
```

crepuscule77@ubuntu: ~/Desktop/mininet

```
mininet>
mininet>
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 比特，来自 10.0.0.2: icmp_seq=1 ttl=64 时间=10.3 毫秒
64 比特，来自 10.0.0.2: icmp_seq=2 ttl=64 时间=0.146 毫秒
64 比特，来自 10.0.0.2: icmp_seq=3 ttl=64 时间=0.050 毫秒
64 比特，来自 10.0.0.2: icmp_seq=4 ttl=64 时间=0.046 毫秒
64 比特，来自 10.0.0.2: icmp_seq=5 ttl=64 时间=0.050 毫秒
64 比特，来自 10.0.0.2: icmp_seq=6 ttl=64 时间=0.049 毫秒
64 比特，来自 10.0.0.2: icmp_seq=7 ttl=64 时间=0.051 毫秒
64 比特，来自 10.0.0.2: icmp_seq=8 ttl=64 时间=0.338 毫秒
64 比特，来自 10.0.0.2: icmp_seq=9 ttl=64 时间=0.051 毫秒
64 比特，来自 10.0.0.2: icmp_seq=10 ttl=64 时间=0.051 毫秒
64 比特，来自 10.0.0.2: icmp_seq=11 ttl=64 时间=0.051 毫秒
64 比特，来自 10.0.0.2: icmp_seq=12 ttl=64 时间=0.057 毫秒
64 比特，来自 10.0.0.2: icmp_seq=13 ttl=64 时间=0.069 毫秒
64 比特，来自 10.0.0.2: icmp_seq=14 ttl=64 时间=0.054 毫秒
64 比特，来自 10.0.0.2: icmp_seq=15 ttl=64 时间=0.049 毫秒
64 比特，来自 10.0.0.2: icmp_seq=16 ttl=64 时间=0.039 毫秒
64 比特，来自 10.0.0.2: icmp_seq=17 ttl=64 时间=0.047 毫秒
64 比特，来自 10.0.0.2: icmp_seq=18 ttl=64 时间=0.050 毫秒
64 比特，来自 10.0.0.2: icmp_seq=19 ttl=64 时间=0.043 毫秒
64 比特，来自 10.0.0.2: icmp_seq=20 ttl=64 时间=0.049 毫秒
64 比特，来自 10.0.0.2: icmp_seq=21 ttl=64 时间=0.129 毫秒
64 比特，来自 10.0.0.2: icmp_seq=22 ttl=64 时间=0.043 毫秒
64 比特，来自 10.0.0.2: icmp_seq=23 ttl=64 时间=0.049 毫秒
64 比特，来自 10.0.0.2: icmp_seq=24 ttl=64 时间=0.049 毫秒
64 比特，来自 10.0.0.2: icmp_seq=25 ttl=64 时间=0.043 毫秒
64 比特，来自 10.0.0.2: icmp_seq=26 ttl=64 时间=0.057 毫秒
64 比特，来自 10.0.0.2: icmp_seq=27 ttl=64 时间=0.058 毫秒
^C
--- 10.0.0.2 ping 统计 ---
已发送 27 个包， 已接收 27 个包, 0% 包丢失, 耗时 26588 毫秒
rtt min/avg/max/mdev = 0.039/0.445/10.265/1.926 ms
```

We can see that when we change the state of the port, there is no impact on the data transfer .

And observing ICMP SEP, we can find that no packet loss occurs.

8. Problems encountered and solutions

Arp storms

In addition to specifying the path directly, I also implemented automatic connectivity through ARP. And since there are loops in the topology of our lab, we need to deal with ARP storms.

I use the policy that a switch can only receive a specific ARP packet from one port. And the second time it receives the same ARP packet, Just drop it. This avoids the formation of a ring.

Data Structure:

> Use (src_ip, dst_ip) to uniquely identify arp packets
>
> Use in_port to record the port from which the packet was first received

```
self.multicast_record       # {(dpid, src_ip, dst_ip) : in_port}
```

Core Code

```
# ryu_get_network_topo.py/flood
def flood(self, msg, datapath, in_port, arp_packet):
        ofproto = datapath.ofproto
        ofp_parser = datapath.ofproto_parser
        dpid = datapath.id

        if self.multicast_record. __contains__((dpid, arp_packet.src_ip, arp_packet.dst_ip)):
            if self.multicast_record[(dpid, arp_packet.src_ip, arp_packet.dst_ip)] != in_port:
                # just drop it
                _msg = ofp_parser.OFPPacketOut(
                    datapath=datapath,
                    buffer_id=datapath.ofproto.OFP_NO_BUFFER,
                    in_port=in_port,
                    actions=[], data=None
                )
                datapath.send_msg(_msg)
                self.logger.info("Drop extra arp packet to avoid arp stroms")
                return

        self.multicast_record[(dpid, arp_packet.src_ip, arp_packet.dst_ip)] = in_port
        for out_port in self.portsOfSwitch[dpid]:
            if out_port != in_port:
                actions = [ofp_parser.OFPActionOutput(out_port)]
                _msg = ofp_parser.OFPPacketOut(
                    datapath = datapath,
                    buffer_id = datapath.ofproto.OFP_NO_BUFFER,
                    in_port = in_port,
                    actions = actions,
                    data = msg.data
                )
                datapath.send_msg(_msg)
        return
```

Send packet from input port

In fast failover problem, for Switch_3 and Switch_4, we need to send the received packet from the input port when one of the ports becomes down.

But when I try to write as belows, it doesn't work.

```
actions_to_port_1_BACK = [ofp_parser.OFPActionOutput(port_1)]
actions_to_port_2 = [ofp_parser.OFPActionOutput(port_2)]

buckets_for_port_1 = [
            ofp_parser.OFPBucket(weight, port_2, watch_group, actions_to_port_2),  # out to port_2
            ofp_parser.OFPBucket(weight, port_1, watch_group, actions_to_port_1)   # out to input port
        ]
```

I used Wireshark to capture packets and found that Switch_3 did not send packets from the input port. It's not what I expected, and I'm confused. After consulting a lot of information, it still failed to solve effectively. And I found that many of my classmates had the same problem.

Finally, one student found that we can't send packets directly from the input port using the port number, instead, openFlow's predefined `ofproto_v1_3.OFPP_IN_PORT` should be used. So it should be like :

```
actions_to_port_1_BACK = [ofp_parser.OFPActionOutput(ofproto_v1_3.OFPP_IN_PORT)]
actions_to_port_2 = [ofp_parser.OFPActionOutput(port_2)]

buckets_for_port_1 = [
            ofp_parser.OFPBucket(weight, port_2, watch_group, actions_to_port_2),                     # out to port_2
            ofp_parser.OFPBucket(weight, ofproto_v1_3.OFPP_IN_PORT, watch_group, actions_to_port_1)   # out to input port
        ]
```

This type of problem is very difficult to debug unless you are very familiar with the protocol ......

9. Summary

Through this lab, I have a deeper understanding of SDN, realize how difficult it is to realize the ideal SDN we learnt in the classroom. And by reading RYU source code, I learnt how is a good communication framework designed.

In addition, I am more proficient in python, mininet, OVS-CTL and other tools. And my debug ability has been greatly improved. In a word, I believe that what I have learned in this lab will benefit me a lot in my future study and scientific research life.

At last, Thanks to all the teachers and students who have helped me.