Ryu controller program design and redirection

Chenkai Zhou

2033555
Information and computing science
Xi'an Jiaotong Liverpool University
Suzhou, China
Chenkai.ZHOU20@student.xjtlu.edu.cn

Haoyang Lin

2036741
Information and computing science
Xi'an Jiaotong Liverpool University
Suzhou, China
Haoyang.Lin20@student.xjtlu.edu.cn

Jiayue Wang

2034912
Information and computing science
Xi'an Jiaotong Liverpool University
Suzhou, China
Jiayue.Wang20@student.xjtlu.edu.cn

***Abstract*—Traditional routers make the data transfer between different subnets much easier, but the vertical integration architecture of the traditional router makes it less flexible and hard to change the router rules. An emerging technology called SDN controller helps to solve the problem with a centralized platform to install the flow table to each router, making the network programmable and more flexible. The project aims to design an SDN controller program through RYU to control the installation of flow tables. We succeed in installing the flow table to handle the packet transmission between the client side and the server side and redirection to the other server. The project explains the basic use of the SDN controller, providing a demo for future study.**

Keywords—computer network, SDN controller, IP protocol, network layer

I. INTRODUCTION

Software defined networking(SDN) is a new network innovation architecture proposed by Stanford's clean slate research group, whose core technology, OpenFlow, enables flexible control of network traffic by separating the control plane of network devices from the data plane, providing a good platform for innovation in core networks and applications[1]. SDN technology is one of the most promising network technologies as it can effectively reduce equipment load, help network operators better control their infrastructure, and reduce overall operating costs.

This project requires us to build an SDN network topology using Mininet and design a Ryu controller program to install flow entries to the switch implementing the forward and redirect functions. We are also required to calculate the average network latency of three handshakes at the first stage of the establishment of a TCP connection by Wireshark.

We meet serval challenges during finishing this project. Firstly, the Python ryu library is relatively complex, and we are unfamiliar with this library. Secondly, the TCP connection of redirection is hard to establish at the first stage, so the client can not receive the response packet from the server. However, we succeed in designing a correct SDN program to instruct the switch to send packets to the related destination host by looking up documents and testing the program.

Although our project is a simple demo for an SDN controller, but it also reveals the core principle of SDN controller, which might be used in many fields. The actions of the switch can be decided by the software, so the firewall can be implemented by the SDN controller by blocking some packets from specific IP addresses. Also, our proposal might be an extremely simple model of load balance application, because the redirect function can be combined with the flow monitoring and the SDN controller will choose a way that is low pressure to forward the packet.

II. RELATED WORK

In the process of finishing the project, we reviewed serval papers concerning the SDN controller, network redirection, and other applications of SDN. We found some examples that have certain guiding significance to our research.

Redirection can be utilized to reduce the maximum response time. The scientists install wildcard rules on the switch to implement a redirection function to minimize the maximum response of the SDN controller[2]. In addition, in Q. Yuan, R. Chai, and Q. Chen's study, they design caching to store flow forwarding policies and rules to achieve efficient data forwarding between two switches in an SDN network, because users flows are associated with a set of flow processing rules[3].

With regard to the performance testing of SDN, the researchers implement several scenario experiments on Mininet and Ryu controllers [4]. They test the scalability of the SDN controller by observing the throughput in dynamic networking conditions over Mesh topology by exponentially increasing the number of nodes.

III. DESIGN

The whole system includes 5 components: Ryu controller, switch, client host, server 1 host, and server 2 host. We focus on the design of the program on the Ryu controller.

A. *Network system design*

Figure 1 shows the basic network topology. The SDN controller is responsible for making the router rules and installing flow tables into the switch. The client and 2 servers are in different subnets respectively: 10.0.1.5/24, 10.0.1.2/24, and 10.0.1.3/24. The MAC address of the client and 2 servers are ten 0s prefixes(00:00:00:00:00) followed by 03, 01, and 02 respectively. The 3 hosts are linked with a switch, so all communication and data transmission must go through the switch. The client communicates with the server on TCP protocol, and it only knows the existence of server 1.

B. *System Workflow*

The Ryu controller first adds a table-miss flow entry to the switch, telling the switch to send the packet_in message to the Ryu controller if the packet does not match any one flow entry in the switch. Then Ryu controller will handle the packet by installing the flow entry to the switch.

Figure 2 illustrates the basic steps of communication between the client and server 1. First, the client sends the connection request to the server. When the request packet gets

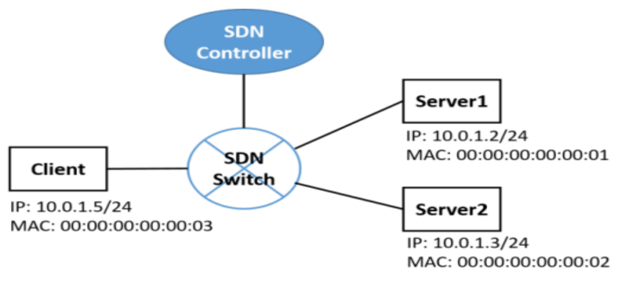


Fig. 1. A simple SDN network topology

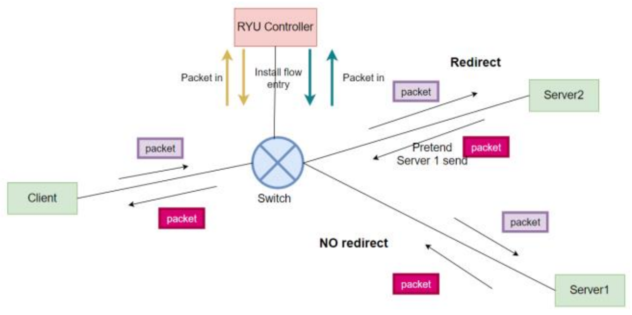


Fig. 2. Ryu workflow

Algorithm 1: Redirect Packet to Server 2

```
1:  mac_to_port={client: 1, server1: 2, server2: 3}
2:  dst_mac ,src_mac = (eth_dst,eth_src)
3:  If dst_mac == server1_mac
4:      out_port = mac_to_port[server2]
5:      set_flow_match(ip_dst=server2_ip)
6:      set_flow_match(eth_dst=server2_mac)
7:  If src_mac ==server2_mac
8:      out_port = mac_to_port[server1]
9:      set_flow_match(ip_src=server1_ip)
10:     set_flow_match(eth_src=server1_mac)
11:   packet out()
```

into the switch, there is no matched flow entry in the switch, so the switch will send the packet-in message to the Ryu controller. After receiving the packet-in message, the controller will analyze the protocol and record the in port. If the protocol is TCP/UDP/ICMP, the Ryu controller will start to create the related flow entry according to the source IP address(client IP), source MAC address(client MAC), destination IP Address(server 1 IP), destination MAC address(server 1 MAC), in-port, and out-port information in the packet-in message. After creating the flow entry, the Ryu controller will install the flow entry to the switch, so the switch can forward the request packet to server 1. In the process of response, the steps are similar to the steps of request. Server 1 will sends the response packet to the switch, and there is no flow entry matched, so the switch will send a packet-in message to the Ryu controller. Then the controller will analyze the information of the packet-in message and create the flow entry to install it into the switch to tell the switch how to forward the response packet.

After the first request and first response(not ARP protocol), the matched flow entry has already been installed in the switch, so the following packets(if the switch is idle less than 5 seconds) will be directly forwarded to the correct port without sending the packet-in message.

Algorithm 1 shows the critical steps of the redirect function. Firstly, set the *mac_to_port* dictionary with the mapping relationship with the existing MAC address of 3 hosts to 3 ports connected with the switch. Secondly, analyze the packet and get the source MAC address and destination MAC address of the packet. Then, redirect the packet from the client to server 2 by setting the flow entry destination IP and MAC address with server 2's IP. Finally, send back the response packet from server 2 to the client by changing the source IP and MAC address with those of server 1, pretending the packet is sent from server 1, which will help to establish the TCP connection successfully.

IV. IMPLEMENTATION

In this section, we report the detailed process of our algorithm and the actual implementation of the system workflow in Section 3.

Setup

The developed host is equipped with Intel(R) Core(TM) i5-1035G1 CPU and 16 GB RAM. For the sake of convenience, we develop our application on Windows 10 OS and use the remote connection (SSH) to test our code on virtual machines(Ubuntu OS). Our virtual machines are equipped with 4GB RAM.

The SDN Controller software is Ryu. All the source codes are implemented in Python 3.10 and developed on JetBrains PyCharm. These Python libraries below are used in our implementation: *ryu.base*, *ryu.controller*, *ryu.ofproto*, *ryu.li.packet*

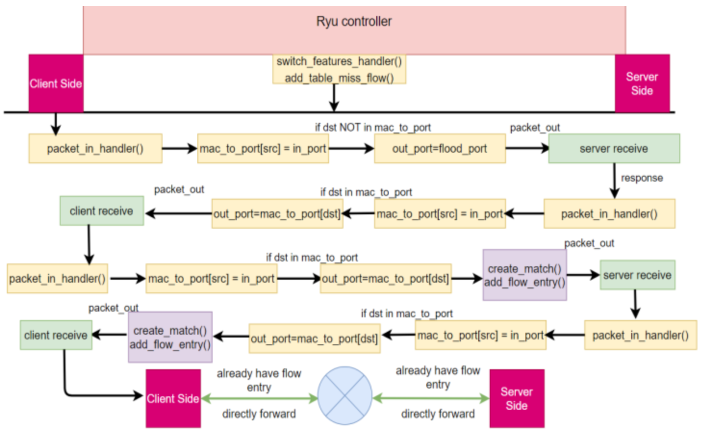


Fig. 3. Basic program flow chart (no redirect)

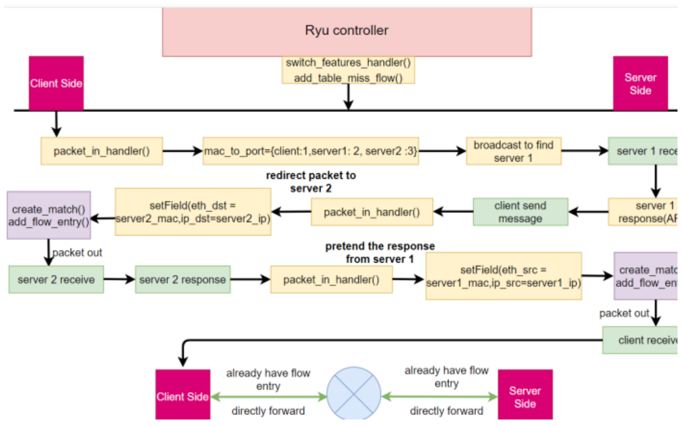
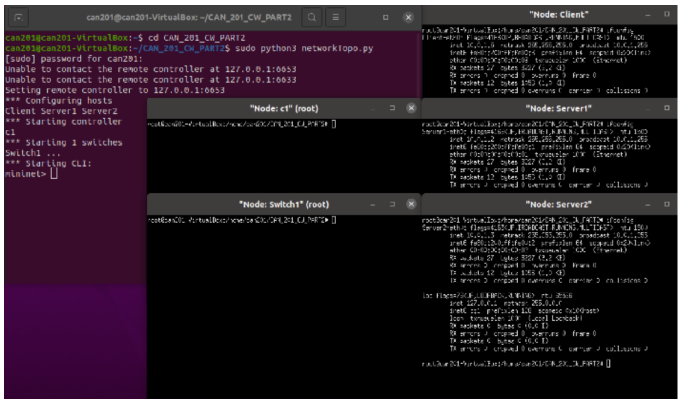


Fig. 4. Redirect program flow chart

Implementation Steps

Figure 3 is the program flow chart of the basic implementation of the Ryu controller program. The Ryu controller will firstly handler the *switch_feature* message, calling the *add_table_miss_flow()* method to add a table-miss flow entry to the switch. After installing the table-miss flow entry, the Ryu controller will start to wait for handling the *packet_in* message. The first request will trigger the *packet_in* event, and the Ryu controller will record the source MAC



```
can201@can201-VirtualBox: ~/CAN_201_CW_PART2
can201@can201-VirtualBox:~$ cd CAN_201_CW_PART2
can201@can201-VirtualBox:~/CAN_201_CW_PART2$ sudo python3 networkTopo.py
[sudo] password for can201:
Unable to contact the remote controller at 127.0.0.1:6653
Unable to contact the remote controller at 127.0.0.1:6633
Setting remote controller to 127.0.0.1:6653
*** Configuring hosts
Client Server1 Server2
*** Starting controller
c1
*** Starting 1 switches
Switch1 ...
*** Starting CLI:
mininet> 
```

```
"Node: Client"
root@can201-VirtualBox:/home/can201/CAN_201_CW_PART2# ifconfig
Client-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.1.5  netmask 255.255.255.0  broadcast 10.0.1.255
        inet6 [illegible]  prefixlen 64  scopeid 0x20<link>
        ether 00:00:00:00:00:03  txqueuelen 1000  (Ethernet)
        RX packets 27  bytes 3227 (3.2 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 12  bytes 1053 (1.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

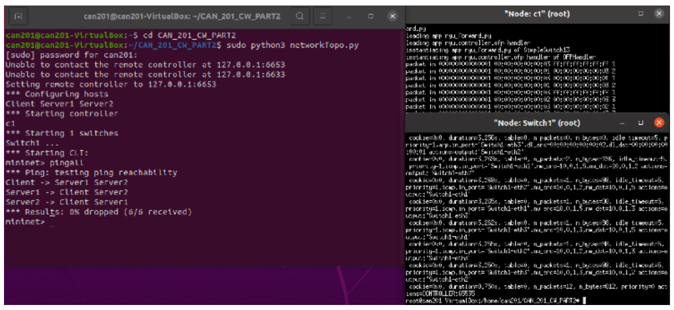
```
"Node: c1" (root)
root@can201-VirtualBox:/home/can201/CAN_201_CW_PART2# 
```

```
"Node: Server1"
root@can201-VirtualBox:/home/can201/CAN_201_CW_PART2# ifconfig
Server1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.1.2  netmask 255.255.255.0  broadcast 10.0.1.255
        inet6 [illegible]  prefixlen 64  scopeid 0x20<link>
        ether 00:00:00:00:00:01  txqueuelen 1000  (Ethernet)
        RX packets 27  bytes 3227 (3.2 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 12  bytes 1053 (1.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

```
"Node: Switch1" (root)
root@can201-VirtualBox:/home/can201/CAN_201_CW_PART2# 
```

```
"Node: Server2"
root@can201-VirtualBox:/home/can201/CAN_201_CW_PART2# ifconfig
Server2-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.1.3  netmask 255.255.255.0  broadcast 10.0.1.255
        inet6 [illegible]  prefixlen 64  scopeid 0x20<link>
        ether 00:00:00:00:00:02  txqueuelen 1000  (Ethernet)
        RX packets 27  bytes 3227 (3.2 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 12  bytes 1055 (1.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@can201-VirtualBox:/home/can201/CAN_201_CW_PART2# 
```

Fig. 5. Snapshot of Task 1(construct a SDN network topology)

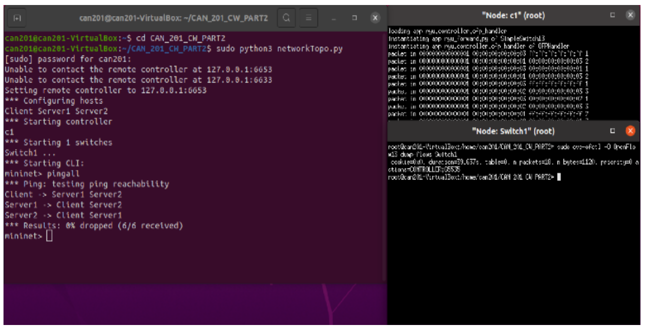


```
can201@can201-VirtualBox: ~/CAN_201_CW_PART2
can201@can201-VirtualBox:~$ cd CAN_201_CW_PART2
can201@can201-VirtualBox:~/CAN_201_CW_PART2$ sudo python3 networkTopo.py
[sudo] password for can201:
Unable to contact the remote controller at 127.0.0.1:6653
Unable to contact the remote controller at 127.0.0.1:6633
Setting remote controller to 127.0.0.1:6653
*** Configuring hosts
Client Server1 Server2
*** Starting controller
c1
*** Starting 1 switches
Switch1 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
Client -> Server1 Server2
Server1 -> Client Server2
Server2 -> Client Server1
*** Results: 0% dropped (6/6 received)
mininet> _
```

```
"Node: c1" (root)
ard.py
loading app ryu_forward.py
loading app ryu.controller.ofp_handler
instantiating app ryu_forward.py of SimpleSwitch13
instantiating app ryu.controller.ofp_handler of OFPHandler
packet in 0000000000000001 00:00:00:00:00:03 ff:ff:ff:ff:ff:ff 1
packet in 0000000000000001 00:00:00:00:00:01 00:00:00:00:00:03 2
packet in 0000000000000001 00:00:00:00:00:03 00:00:00:00:00:01 1
packet in 0000000000000001 00:00:00:00:00:01 00:00:00:00:00:03 2
packet in 0000000000000001 00:00:00:00:00:03 ff:ff:ff:ff:ff:ff 1
packet in 0000000000000001 00:00:00:00:00:02 00:00:00:00:00:03 3
packet in 0000000000000001 00:00:00:00:00:03 00:00:00:00:00:02 1
```

```
"Node: Switch1" (root)
 cookie=0x0, duration=3.256s, table=0, n_packets=0, n_bytes=0, idle_timeout=5, p
riority=1,arp,in_port="Switch1-eth3",dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00
:00:01 actions=output:"Switch1-eth2"
 cookie=0x0, duration=3.268s, table=0, n_packets=2, n_bytes=196, idle_timeout=5,
 priority=1,icmp,in_port="Switch1-eth1",nw_src=10.0.1.5,nw_dst=10.0.1.2 actions=
output:"Switch1-eth2"
 cookie=0x0, duration=3.268s, table=0, n_packets=1, n_bytes=98, idle_timeout=5,
priority=1,icmp,in_port="Switch1-eth2",nw_src=10.0.1.2,nw_dst=10.0.1.5 actions=o
utput:"Switch1-eth1"
 cookie=0x0, duration=3.265s, table=0, n_packets=1, n_bytes=98, idle_timeout=5,
priority=1,icmp,in_port="Switch1-eth1",nw_src=10.0.1.5,nw_dst=10.0.1.3 actions=o
utput:"Switch1-eth3"
 cookie=0x0, duration=3.262s, table=0, n_packets=1, n_bytes=98, idle_timeout=5,
priority=1,icmp,in_port="Switch1-eth3",nw_src=10.0.1.3,nw_dst=10.0.1.5 actions=o
utput:"Switch1-eth1"
 cookie=0x0, duration=3.256s, table=0, n_packets=1, n_bytes=98, idle_timeout=5,
priority=1,icmp,in_port="Switch1-eth2",nw_src=10.0.1.2,nw_dst=10.0.1.3 actions=o
utput:"Switch1-eth3"
 cookie=0x0, duration=3.250s, table=0, n_packets=1, n_bytes=98, idle_timeout=5,
priority=1,icmp,in_port="Switch1-eth3",nw_src=10.0.1.3,nw_dst=10.0.1.2 actions=o
utput:"Switch1-eth2"
 cookie=0x0, duration=0.750s, table=0, n_packets=12, n_bytes=812, priority=0 act
ions=CONTROLLER:65535
root@can201-VirtualBox:/home/can201/CAN_201_CW_PART2# 
```

Fig. 6. Snapshot of Task 2(each node is reachable from the others)



```
can201@can201-VirtualBox: ~/CAN_201_CW_PART2
can201@can201-VirtualBox:~$ cd CAN_201_CW_PART2
can201@can201-VirtualBox:~/CAN_201_CW_PART2$ sudo python3 networkTopo.py
[sudo] password for can201:
Unable to contact the remote controller at 127.0.0.1:6653
Unable to contact the remote controller at 127.0.0.1:6633
Setting remote controller to 127.0.0.1:6653
*** Configuring hosts
Client Server1 Server2
*** Starting controller
c1
*** Starting 1 switches
Switch1 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
Client -> Server1 Server2
Server1 -> Client Server2
Server2 -> Client Server1
*** Results: 0% dropped (6/6 received)
mininet> 
```

```
"Node: c1" (root)
loading app ryu.controller.ofp_handler
instantiating app ryu_forward.py of SimpleSwitch13
instantiating app ryu.controller.ofp_handler of OFPHandler
packet in 0000000000000001 00:00:00:00:00:03 ff:ff:ff:ff:ff:ff 1
packet in 0000000000000001 00:00:00:00:00:01 00:00:00:00:00:03 2
packet in 0000000000000001 00:00:00:00:00:03 00:00:00:00:00:01 1
packet in 0000000000000001 00:00:00:00:00:01 00:00:00:00:00:03 2
packet in 0000000000000001 00:00:00:00:00:03 ff:ff:ff:ff:ff:ff 1
packet in 0000000000000001 00:00:00:00:00:02 00:00:00:00:00:03 3
packet in 0000000000000001 00:00:00:00:00:03 00:00:00:00:00:02 1
packet in 0000000000000001 00:00:00:00:00:02 00:00:00:00:00:03 3
packet in 0000000000000001 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 2
```

```
"Node: Switch1" (root)
root@can201-VirtualBox:/home/can201/CAN_201_CW_PART2# sudo ovs-ofctl -O OpenFlo
w13 dump flows Switch1
 cookie=0x0, duration=39.637s, table=0, n_packets=18, n_bytes=1120, priority=0 a
ctions=CONTROLLER:65535
root@can201-VirtualBox:/home/can201/CAN_201_CW_PART2# 
```

Fig. 7. Snapshot of Task 2(every flow entry is set an idle timeout of 5 seconds)

address and look up the *mac_to_port* dictionary, then flood the packet. After flooding, server 1 will receive the first request and give the response. Afterward, the Ryu controller will handle the response packet and let the *out_port* be the client port because the destination MAC address of the client has already been recorded in the *mac_to_port* dictionary.

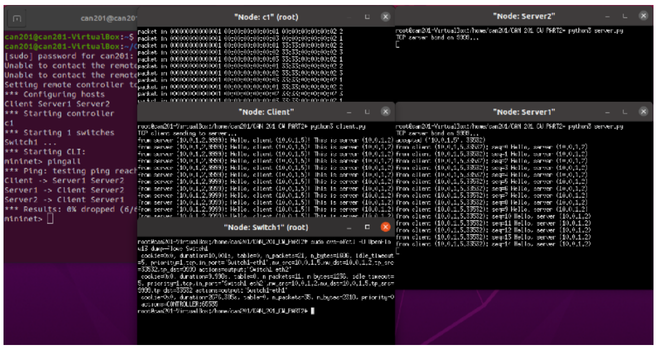
After receiving the confirmation message, the client knows server 1 is on and sends the first SYN to server 1. When receiving a packet-in message, the Ryu controller will prepare the source/destination IP address, source/destination port, in port, and forward action according to the packet information from parameters to call the *add_flow_entry* method to install a flow entry with idle time out in 5 seconds to the switch, so switch can forward the packet to server1. Server 1 will send SYN+ACK to the client to agree the connection request. Then Ryu controller will call the *add_flow_entry* method with the IP address, TCP port, and forward action to install a flow entry with idle time out in 5 seconds to the switch to instruct the switch to handle the packet from server 1. After installing the flow table, the packet will be sent to the client. On the grounds that there exist flow entries in the switch which can match the enter packet, the packet can be directly forwarded to the related host, and the switch will not send the *packet_in* message to the Ryu controller. The following communication between client and server 1 will keep going without the Ryu giving a new flow entry if the switch is not idle for longer than 5 seconds.

Figure 4 is the implementation of the redirect function. The starting flooding steps are same as before. It also uses ARP protocol to broadcast all the hosts to find server 1, and server 1 gives the response of broadcast. However, the first SYN packet from the client will be redirected to server 2. The program will first set the *mac_to_port* with 3 hosts' MAC addresses and 3 ports. When the first SYN packet enters the switch, the Ryu controller will handle the packet-in message and create a flow entry. It will prepare the action that set the flow entry field destination MAC address with the MAC address of server 2 and set the flow entry field destination IP address with the IP address of server 2, and it also lets the out port be the port connected with server 2. After preparing the actions, the Ryu controller will install the flow table by calling the *add_flow_entry* method. The switch will change the flow entry set according to the actions from the Ryu controller and forward the packet to server 2.

After receiving the SYN packet, server 2 will acknowledge the connection and respond back the SYN+ACK. However, the client only knows the existence of server 1, thus client will check the destination IP whether it is from server 1. As a result, direct forwarding the packet from server 2 to client is impossible. The Ryu controller will prepare actions which set the source IP and MAC address with server 1's IP and MAC address, pretending the response message is from server 1 which will help to pass the check of the client. After the flow entry is installed, the switch will forward the packet to the client. The packet will be received by the client. The following communication will be directly forwarded by the switch because the flow entries have been correctly installed. The client establishes the TCP connection with server 2, but the client also thinks it is communicating with server 1 and all the request messages are originally sent to server 1, so the redirect function is done.

Programming skills and difficulties

We use Object-Oriented Programming to implement our Ryu controller program. We encapsulate a class *SimpleSwitch13,* which extends the Python class *app_manager.RyuApp*. The class includes serval methods: the add_table_miss_flow() method helps to add table miss flow at the start of the program, the *add_flow_entry()* method helps Ryu controller install flow entry when handling the packet in the message, the *switch_feature_handler()* and *packet_in_handler()* method will handle the event switch feature and packet in respectively. We encountered some difficulties when we tried to redirect the packet from server 1 to server 2 because the client was hard to connect to server 2 at first. However, we overcame the problem by changing the *out_port* by the serve 2's MAC address and adding actions to set flow entry fields for server 2, and changing flow entry to pretend the response packet from server 1 to avoid being dropped when the client checks the IP address, finally making it redirect successfully. In addition, we are not extremely familiar with the use of the ryu library, but we solve this problem by looking up related documents on the Internet.



```
can201@can201-VirtualBox:~$
can201@can201-VirtualBox:~/
[sudo] password for can201:
Unable to contact the remote
Unable to contact the remote
Setting remote controller to
*** Configuring hosts
Client Server1 Server2
*** Starting controller
c1
*** Starting 1 switches
Switch1 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reach
Client -> Server1 Server2
Server1 -> Client Server2
Server2 -> Client Server1
*** Results: 0% dropped (6/6
mininet> 
```

"Node: c1" (root)

```
packet in 0000000000000001 00:00:00:00:00:01 00:00:00:00:00:02 2
packet in 0000000000000001 00:00:00:00:00:03 00:00:00:00:00:02 1
packet in 0000000000000001 00:00:00:00:00:01 33:33:00:00:00:02 2
packet in 0000000000000001 00:00:00:00:00:02 33:33:00:00:00:02 3
packet in 0000000000000001 00:00:00:00:00:03 33:33:00:00:00:02 1
packet in 0000000000000001 00:00:00:00:00:01 33:33:00:00:00:02 2
packet in 0000000000000001 00:00:00:00:00:02 33:33:00:00:00:02 3
packet in 0000000000000001 00:00:00:00:00:03 33:33:00:00:00:02 1
packet in 0000000000000001 00:00:00:00:00:01 33:33:00:00:00:02 2
packet in 0000000000000001 00:00:00:00:00:02 33:33:00:00:00:02 3
```

"Node: Server2"

```
root@can201-VirtualBox:/home/can201/CAN 201 CW PART2# python3 server.py
TCP server bind on 9999...
```

"Node: Client"

```
root@can201-VirtualBox:/home/can201/CAN 201 CW PART2# python3 client.py
TCP client sending to server...
from server (10,0,1,2,9999): Hello, client (10,0,1,5)! This is server (10,0,1,2)
from server (10,0,1,2,9999): Hello, client (10,0,1,5)! This is server (10,0,1,2)
from server (10,0,1,2,9999): Hello, client (10,0,1,5)! This is server (10,0,1,2)
from server (10,0,1,2,9999): Hello, client (10,0,1,5)! This is server (10,0,1,2)
from server (10,0,1,2,9999): Hello, client (10,0,1,5)! This is server (10,0,1,2)
from server (10,0,1,2,9999): Hello, client (10,0,1,5)! This is server (10,0,1,2)
from server (10,0,1,2,9999): Hello, client (10,0,1,5)! This is server (10,0,1,2)
from server (10,0,1,2,9999): Hello, client (10,0,1,5)! This is server (10,0,1,2)
from server (10,0,1,2,9999): Hello, client (10,0,1,5)! This is server (10,0,1,2)
from server (10,0,1,2,9999): Hello, client (10,0,1,5)! This is server (10,0,1,2)
from server (10,0,1,2,9999): Hello, client (10,0,1,5)! This is server (10,0,1,2)
```

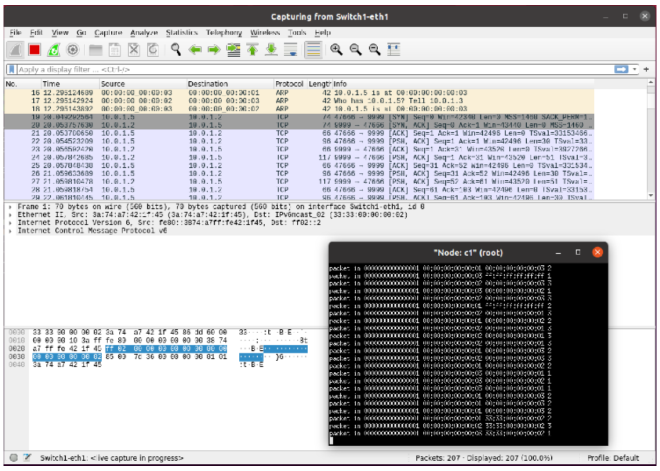
"Node: Server1"

```
root@can201-VirtualBox:/home/can201/CAN 201 CW PART2# python3 server.py
TCP server bind on 9999...
accepted ('10.0.1.5', 33532)
from client (10,0,1,5,33532): seq=0 Hello, server (10,0,1,2)
from client (10,0,1,5,33532): seq=1 Hello, server (10,0,1,2)
from client (10,0,1,5,33532): seq=2 Hello, server (10,0,1,2)
from client (10,0,1,5,33532): seq=3 Hello, server (10,0,1,2)
from client (10,0,1,5,33532): seq=4 Hello, server (10,0,1,2)
from client (10,0,1,5,33532): seq=5 Hello, server (10,0,1,2)
from client (10,0,1,5,33532): seq=6 Hello, server (10,0,1,2)
from client (10,0,1,5,33532): seq=7 Hello, server (10,0,1,2)
from client (10,0,1,5,33532): seq=8 Hello, server (10,0,1,2)
from client (10,0,1,5,33532): seq=9 Hello, server (10,0,1,2)
from client (10.0.1.5,33532): seq=10 Hello, server (10.0.1.2)
from client (10.0.1.5,33532): seq=11 Hello, server (10.0.1.2)
from client (10.0.1.5,33532): seq=12 Hello, server (10.0.1.2)
from client (10.0.1.5,33532): seq=13 Hello, server (10.0.1.2)
from client (10.0.1.5,33532): seq=14 Hello, server (10.0.1.2)
```

"Node: Switch1" (root)

```
root@can201-VirtualBox:/home/can201/CAN_201_CW_PART2# sudo ovs-ofctl -O OpenFlo
w13 dump-flows Switch1
 cookie=0x0, duration=10.001s, table=0, n_packets=21, n_bytes=1806, idle_timeout
=5, priority=1,tcp,in_port="Switch1-eth1",nw_src=10.0.1.5,nw_dst=10.0.1.2,tp_src
=33532,tp_dst=9999 actions=output:"Switch1-eth2"
 cookie=0x0, duration=9.990s, table=0, n_packets=11, n_bytes=1236, idle_timeout=
5, priority=1,tcp,in_port="Switch1-eth2",nw_src=10.0.1.2,nw_dst=10.0.1.5,tp_src=
9999,tp_dst=33532 actions=output:"Switch1-eth1"
 cookie=0x0, duration=2676.305s, table=0, n_packets=35, n_bytes=2310, priority=0
 actions=CONTROLLER:65535
root@can201-VirtualBox:/home/can201/CAN_201_CW_PART2# 
```

Fig. 8. Snapshot of Task 4(Apply the given socket programs to the SDN network topology)



Capturing from Switch1-eth1

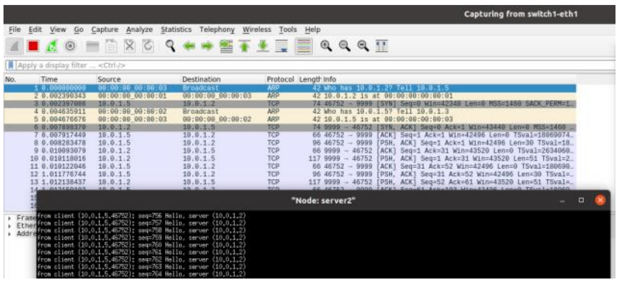
Apply a display filter ... <Ctrl-/>

```
No.  Time          Source             Destination        Protocol Length Info
  16 12.295124609  00:00:00_00:00:03  00:00:00_00:00:01  ARP       42 10.0.1.5 is at 00:00:00:00:00:03
  17 12.295142924  00:00:00_00:00:02  00:00:00_00:00:03  ARP       42 Who has 10.0.1.5? Tell 10.0.1.3
  18 12.295143892  00:00:00_00:00:03  00:00:00_00:00:02  ARP       42 10.0.1.5 is at 00:00:00:00:00:03
  19 20.049292564  10.0.1.5           10.0.1.2           TCP       74 47666 → 9999 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM=1…
  20 20.053757630  10.0.1.2           10.0.1.5           TCP       74 9999 → 47666 [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460 …
  21 20.053700650  10.0.1.5           10.0.1.2           TCP       66 47666 → 9999 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=33153466…
  22 20.054523209  10.0.1.5           10.0.1.2           TCP       96 47666 → 9999 [PSH, ACK] Seq=1 Ack=1 Win=42496 Len=30 TSval=33…
  23 20.058592420  10.0.1.2           10.0.1.5           TCP       66 9999 → 47666 [ACK] Seq=1 Ack=31 Win=43520 Len=0 TSval=3927286…
  24 20.057842685  10.0.1.2           10.0.1.5           TCP      117 9999 → 47666 [PSH, ACK] Seq=1 Ack=31 Win=43520 Len=51 TSval=3…
  25 20.057848430  10.0.1.5           10.0.1.2           TCP       66 47666 → 9999 [ACK] Seq=31 Ack=52 Win=42496 Len=0 TSval=331534…
  26 21.059633609  10.0.1.5           10.0.1.2           TCP       96 47666 → 9999 [PSH, ACK] Seq=31 Ack=52 Win=42496 Len=30 TSval=…
  27 21.059810478  10.0.1.2           10.0.1.5           TCP      117 9999 → 47666 [PSH, ACK] Seq=52 Ack=61 Win=43520 Len=51 TSval=…
  28 21.059818754  10.0.1.5           10.0.1.2           TCP       66 47666 → 9999 [ACK] Seq=61 Ack=103 Win=42496 Len=0 TSval=33153…
  29 22.061810445  10.0.1.5           10.0.1.2           TCP       96 47666 → 9999 [PSH, ACK] Seq=61 Ack=103 Win=42496 Len=30 TSval…

› Frame 1: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on interface Switch1-eth1, id 0
› Ethernet II, Src: 3a:74:a7:42:1f:45 (3a:74:a7:42:1f:45), Dst: IPv6mcast_02 (33:33:00:00:00:02)
› Internet Protocol Version 6, Src: fe80::3874:a7ff:fe42:1f45, Dst: ff02::2
› Internet Control Message Protocol v6

0000  33 33 00 00 00 02 3a 74  a7 42 1f 45 86 dd 60 00   33····:t ·B·E··`·
0010  00 00 00 10 3a ff fe 80  00 00 00 00 00 00 38 74   ····:··· ······8t
0020  a7 ff fe 42 1f 45 ff 02  00 00 00 00 00 00 00 00   ···B·E·· ········
0030  00 00 00 00 00 02 85 00  7c 36 00 00 00 00 01 01   ········ |6······
0040  3a 74 a7 42 1f 45                                  :t·B·E

Switch1-eth1: <live capture in progress>          Packets: 207 · Displayed: 207 (100.0%)          Profile: Default
```

"Node: c1" (root)

```
packet in 0000000000000001 00:00:00:00:00:01 00:00:00:00:00:03 2
packet in 0000000000000001 00:00:00:00:00:03 ff:ff:ff:ff:ff:ff 1
packet in 0000000000000001 00:00:00:00:00:02 00:00:00:00:00:03 3
packet in 0000000000000001 00:00:00:00:00:03 00:00:00:00:00:02 1
packet in 0000000000000001 00:00:00:00:00:02 00:00:00:00:00:03 3
packet in 0000000000000001 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 2
packet in 0000000000000001 00:00:00:00:00:02 00:00:00:00:00:01 3
packet in 0000000000000001 00:00:00:00:00:01 00:00:00:00:00:02 2
packet in 0000000000000001 00:00:00:00:00:02 00:00:00:00:00:01 3
packet in 0000000000000001 00:00:00:00:00:02 00:00:00:00:00:01 3
packet in 0000000000000001 00:00:00:00:00:02 00:00:00:00:00:01 3
packet in 0000000000000001 00:00:00:00:00:01 00:00:00:00:00:03 2
packet in 0000000000000001 00:00:00:00:00:02 00:00:00:00:00:03 3
packet in 0000000000000001 00:00:00:00:00:01 00:00:00:00:00:02 2
packet in 0000000000000001 00:00:00:00:00:03 00:00:00:00:00:01 1
packet in 0000000000000001 00:00:00:00:00:03 00:00:00:00:00:02 1
packet in 0000000000000001 00:00:00:00:00:03 00:00:00:00:00:01 1
packet in 0000000000000001 00:00:00:00:00:01 00:00:00:00:00:03 2
packet in 0000000000000001 00:00:00:00:00:01 00:00:00:00:00:02 2
packet in 0000000000000001 00:00:00:00:00:01 00:00:00:00:00:03 2
packet in 0000000000000001 00:00:00:00:00:01 33:33:00:00:00:02 2
packet in 0000000000000001 00:00:00:00:00:02 33:33:00:00:00:02 3
packet in 0000000000000001 00:00:00:00:00:03 33:33:00:00:00:02 1
```

Fig. 9. Snapshot of Task 4.2(capture packets via Wireshark)



Capturing from switch1-eth1

Apply a display filter ... <Ctrl-/>

```
No.  Time         Source             Destination        Protocol Length Info
   1 0.000000000  00:00:00_00:00:03  Broadcast          ARP       42 Who has 10.0.1.2? Tell 10.0.1.5
   2 0.002390343  00:00:00_00:00:01  00:00:00_00:00:03  ARP       42 10.0.1.2 is at 00:00:00:00:00:01
   3 0.002397086  10.0.1.5           10.0.1.2           TCP       74 46752 → 9999 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM=1…
   4 0.004635911  00:00:00_00:00:02  Broadcast          ARP       42 Who has 10.0.1.5? Tell 10.0.1.3
   5 0.004676676  00:00:00_00:00:03  00:00:00_00:00:02  ARP       42 10.0.1.5 is at 00:00:00:00:00:03
   6 0.007898370  10.0.1.2           10.0.1.5           TCP       74 9999 → 46752 [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460 …
   7 0.007917449  10.0.1.5           10.0.1.2           TCP       66 46752 → 9999 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=18069074…
   8 0.008283478  10.0.1.5           10.0.1.2           TCP       96 46752 → 9999 [PSH, ACK] Seq=1 Ack=1 Win=42496 Len=30 TSval=18…
   9 0.010093079  10.0.1.2           10.0.1.5           TCP       66 9999 → 46752 [ACK] Seq=1 Ack=31 Win=43520 Len=0 TSval=2634060…
  10 0.010118016  10.0.1.2           10.0.1.5           TCP      117 9999 → 46752 [PSH, ACK] Seq=1 Ack=31 Win=43520 Len=51 TSval=2…
  11 0.010122046  10.0.1.5           10.0.1.2           TCP       66 46752 → 9999 [ACK] Seq=31 Ack=52 Win=42496 Len=0 TSval=180698…
  12 1.011776744  10.0.1.5           10.0.1.2           TCP       96 46752 → 9999 [PSH, ACK] Seq=31 Ack=52 Win=42496 Len=30 TSval=…
  13 1.012138437  10.0.1.2           10.0.1.5           TCP      117 9999 → 46752 [PSH, ACK] Seq=52 Ack=61 Win=43520 Len=51 TSval=…
```

"Node: server2"

```
from client (10,0,1,5,46752): seq=756 Hello, server (10,0,1,2)
from client (10,0,1,5,46752): seq=757 Hello, server (10,0,1,2)
from client (10,0,1,5,46752): seq=758 Hello, server (10,0,1,2)
from client (10,0,1,5,46752): seq=759 Hello, server (10,0,1,2)
from client (10,0,1,5,46752): seq=760 Hello, server (10,0,1,2)
from client (10,0,1,5,46752): seq=761 Hello, server (10,0,1,2)
from client (10,0,1,5,46752): seq=762 Hello, server (10,0,1,2)
from client (10,0,1,5,46752): seq=763 Hello, server (10,0,1,2)
from client (10,0,1,5,46752): seq=764 Hello, server (10,0,1,2)
```

Fig. 10. Snapshot of Task 5.2(capture packets via Wireshark)

V. TESTING AND RESULTS

The testing environment is the same as the implementation environment. Especially, the Operating System is Ubuntu(64-bit), the RAM is 4096MB, and the Python version is 3.10.

Firstly, regarding Task 1, we construct an SDN network topology using the Mininet Python library, which contains an SDN Switch, an SDN Controller, a Client, and two Servers Server1 and Server2. The appropriate IP and Mac addresses are configured for the Servers and Client, respectively, which is demonstrated in Figure 5.

Regarding Task 2, through the Ryu framework, we successfully developed an SDN controller application, and each node is reachable from the others. In addition, every flow entry has been set with an idle timeout of 5 seconds. Figure 6 shows several flow entries produced by the pingall command, while Figure 7 shows 5 seconds after the pingall command was executed, those flow entries were cleared.

Regarding Task 3, In this SDN network topology, we successfully implement the provided socket client program (client.py) and socket server program (server.py). Start by running server.py on Server 1 and Server 2 respectively, then 5 seconds after ping, run the client.py on Client. Figure 8 displays the information printed by the Client and Server, along with the flow entries.

Regarding Task 4, we accomplished the prerequisites in Task 4.1. First, the controller was able to send packets as required and install process entries into the SDN switch, and all the following traffic was forwarded to Server 1. Figure 9 displays the outcomes of the packets capture using Wireshark.

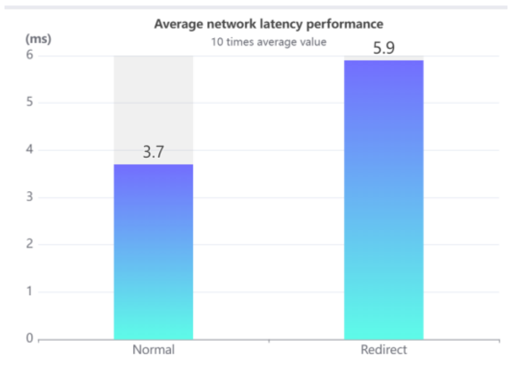


Fig. 11. Average network latency performance

Regarding Task 5, we accomplished the redirection function and test the function by running the client.py on the client and running server.py on both server 1 and server 2. Server 2 successfully prints the result and server 1 has no results, which means the redirect function is correctly implemented. Task 5.2, we capture the packet on the client side with Wireshark, Figure 10 shows the outcomes of capturing packets and server 2 results.

Figure 11 shows the average network latency performance. We calculate the latency between the client sending the first SYN and it sending the first ACK(the third handshake). We measure the first three handshake latency 10 times and calculate the average value to avoid the negative network influence. The final latency result of normal forward is 3.7 ms and that of redirect is 5.9 ms. The redirect costs more time, but the latency is also in the acceptable range.

VI. CONCLUSION

In this project, we developed a Ryu controller program to implement the basic forward and redirect function in the network topology. We succeed in connecting the client with the server and deciding the action of the switch.

However, our program still has some drawbacks that can be improved in the future. Firstly, the program only solves the redirect problem, but the SDN controller can have many kinds of actions like block, flood, and so on. In the future, we can implement other functions to make our program robust and play more roles in the network. Secondly, the operations of the program all depend on the command line input, which is too complex for a nonprofessional person, so future research can focus on the easy use of the program.

ACKNOWLEDGMENT

Chenkai Zhou(2033555), Haoyang Lin （2036741） Jiayue Wang（2034912） contribute same to the project

REFERENCES

[1] N. McKeown et al., "OpenFlow: Enabling innovation in campus networks", ACM SIGCOMM Comput. Commun. Rev., vol. 38, no. 2, pp. 69-74, 2008.

[2] P. Wang, H. Xu, L. Huang, C. Qian, S. Wang, and Y. Sun, "Minimizing Controller Response Time Through Flow Redirecting in SDNs," in IEEE/ACM Transactions on Networking, vol. 26, no. 1, pp. 562-575, Feb. 2018

[3] Q. Yuan, R. Chai, and Q. Chen, "End-to-End Delay Minimization-based Joint Rule Caching and Flow Forwarding Algorithm for SDN," 2019 11th International Conference on Wireless Communications and Signal Processing (WCSP), 2019, pp. 1-6

[4] S. Asadollahi, B. Goswami and M. Sameer, "Ryu controller's scalability experiment on software defined networks," 2018 IEEE International Conference on Current Trends in Advanced Computing (ICCTAC), 2018, pp. 1-5