Traffic Forwarding and Redirection in Mininet SDN Topology Using the Ryu Framework

1st Ruiyang.Wu
School of Advanced Technology
Xi'an Jiaotong - Liverpool University
Suzhou, China
2257475

1st Junhao.Huang
School of Advanced Technology
Xi'an Jiaotong - Liverpool University
Suzhou, China
2256792

1st Daiyan.Wu
School of Advanced Technology
Xi'an Jiaotong - Liverpool University
Suzhou, China
2257387

1st Yuxiao.Tang
School of Advanced Technology
Xi'an Jiaotong - Liverpool University
Suzhou, China
2252554

1st Weizhe.Sun
School of Advanced Technology
Xi'an Jiaotong - Liverpool University
Suzhou, China
1927150

***Abstract*—This project utilizes the Mininet and Ryu frameworks to design a simple Software-Defined Network (SDN) addressing traditional network challenges. It involves creating an SDN topology, simulating traffic control, and developing a controller for targeted traffic forwarding and service redirection. Key tasks include creating flow entries, analyzing traffic, and calculating latency for seamless client-server redirection. Diagrams, pseudo-code, and flow charts illustrate the design and implementation, while testing confirms the system's efficiency. Finally, we put forward some prospects for future improvement.**

Keywords*—*SDN, Ryu controller, Mininet, Redirecting, TCP/IP, Load balancing

I. INTRODUCTION

A. Background

Software-Defined Networking (SDN) is an emerging kind of network architecture as it separates the control plane from the data plane of the network which changes how traditional networks are designed and managed. As a fact, traditional network devices integrate data with control logic but SDN introduces a centralized controller that allows network administrators to manage network traffic through software and adjust policies in real time dynamically.

B. Project Tasks

Specifically, the project can be divided into three parts which are:

1) Use Mininet to build the network topology, configure the correct IP and MAC address, and ensure basic network connectivity.
2) Develop an SDN controller application by using the Ryu framework to implement traffic forwarding and flow table management functionalities.
3) Implement functions such as traffic redirection and network latency measurement to analyze and compare test results.

C. Project challenges

Meanwhile, when achieving project objectives, there also exists some challenges remain to be solved:

1) It is necessary to generate, update and delete flow table entries dynamically which are based on real-time network traffic with avoiding performance degradation caused by frequent updates.
2) Use Wireshark to capture TCP handshake packets and measure the latency from the first SYN packet to the ACK packet accurately.
3) Develop a controller program capable of real-time traffic redirection from Server1 to Server2 while ensuring uninterrupted traffic and minimal latency.

D. Contribution

To completing the project successfully, our group members should understand and master the core concepts of SDN. Besides, we learn about the Ryu controller programming framework, master the implementation of event handling and dynamically deploying flow table entries. Overall, this project integrates SDN theory, network simulation, controller programming, algorithm design, and network performance analysis, comprehensively enhancing the understanding and practical skills in SDN technologies.

II. RELATED WORK

Software Defined Networking (SDN) is an emerging architecture that can simplify network management; its main advantages are the separation of the control and data layers, allowing dynamic management and optimization of network resources, improving flexibility and programmability, and enabling traffic engineering [1].

Nginx [2] is a open source web server reverse proxy server project. It has a similar idea of redirecting network traffic: Nginx servers can distribute and balance different client requests to cross different servers in a computing cluster. This is an example of traffic redirection.

With the rapid growth of streaming multimedia services, traditional Content Delivery Networks (CDNs) are facing challenges such as the inability to guarantee a high user experience and increased infrastructure costs. The emergence of SDN provides new approaches to address these challenges, such as locating the nearest content server to users and redirecting them to that server, which improves energy efficiency and user experience according to Yang et al. [3].

This redirection process is invisible to the user, who doesn't know which server is handling the request. They can only see the server responding to their request as usual, just as the client is communicating with server1 but is actually communicating with server2.

III. Design

This section will focus on the design of the project, including the architecture of an SDN network, the process of the solution of the project and the algorithm of the SDN controller.

A. Architecture of SDN Network

The SDN topology is composed of a centralized SDN controller, an SDN switch, and three hosts—one acting as a client and the other two as servers. The network structure is shown in Fig.1. The following is about the explanation of main elements:

1) SDN Controller: The SDN controller is the centralized control core of the network and it is implemented by using the open source Ryu framework. It utilizes a three-layer architecture which can provide essential functionalities through its northbound API to manage traffic flows dynamically. The controller interacts with the switch through the OpenFlow protocol to handle key events such as `'Packet_In'` and `'Packet_Out'`. The interaction mechanism enables the controller to create flow table entries in the switch to manage, forward or redirect traffic.
2) SDN Switch: SDN switch is responsible for maintaining flow tables defined by the OpenFlow standard. Each flow table entry contains several key elements such as match fields, priority and action. Match fields can be configured to filter traffic based on IPv4 address or TCP port while the associated action determines how to handle the packet. If the packet doesn't match an existing rule, the switch sends a `'Packet_In'` message to the controller to request further instruction. After receiving the Packet-Out response, the switch will update the flow table entries.
3) Hosts: The network host consists one client and two servers which is assigned unique IP and MAC addresses for the interfaces. The client is responsible for initiating traffic to the server, and two-way communication is achieved through socket programming. The host serves as a terminal for testing traffic forwarding and redirection mechanisms and it will be managed by the SDN controller uniformly.

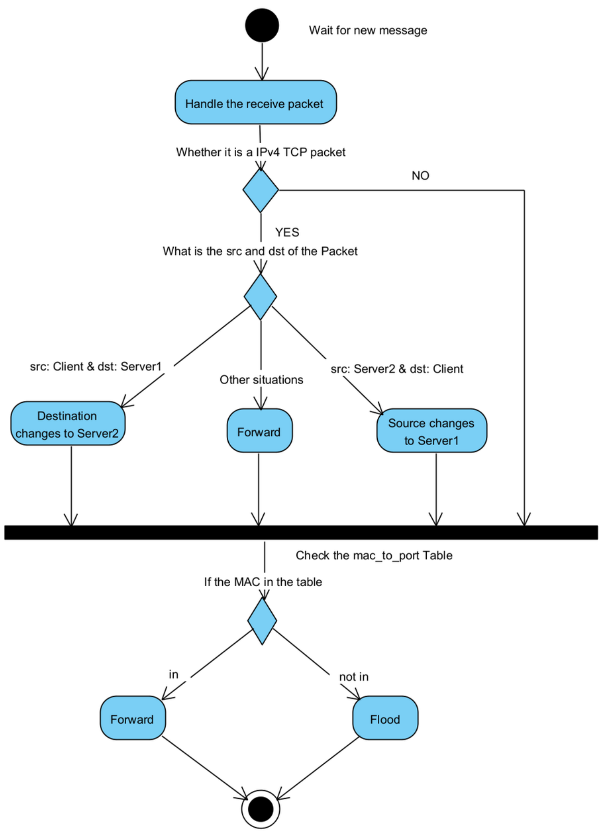

Wait for new message
Handle the receive packet
Whether it is a IPv4 TCP packet
NO
YES
What is the src and dst of the Packet
src: Client & dst: Server1
Other situations
src: Server2 & dst: Client
Destination changes to Server2
Forward
Source changes to Server1
Check the mac_to_port Table
If the MAC in the table
in
not in
Forward
Flood

Fig. 1. Redirecting Activity Diagram

B. Workflow of the project

The following outlines the essential solution workflow:

1) At the initialization stage, the SDN controller instructs all connected switches to add a persistent default entry which is called the table-miss flow entry to their flow tables.
2) For packets that don't match any pre-existing entries, the controller will parse the `'Packet_In'` to extract necessary information such as source and destination MAC addresses. If the destination MAC address is known, the controller then will install a flow entry to the forward packet to the specific output port which is associated with the destination.
3) To handle those flows with specific destinations, we employ different strategies based on the protocol type of traffic such as UDP, TCP, ICMP and ARP while each strategy is tailored to ensure optimal handling.
4) Therefore, the controller can send `'Flow_Mod'` to the switch to add these flow entries to its flow table. Once installed, the switch can handle subsequent packets of similar flows without controller intervention autonomously.

Algorithm 1: Forwarding Algorithm

```
Input: message packet in event
pkt ← ev.msg
eth ← pkt.get_protocols(ethernet)
dst_mac ← eth.dst  # MAC Address
src_mac ← eth.src
if mac address is found) then
    out_port = mac_to_port[dpid][dst_mac]  # Dictionary
else
    out_port = FLOOD
end
match ← in_port=in_port, eth_dst=dst_mac, eth_src=src_mac)
actions ← out_port
add_flow ← match, actions, idle_timeout = 5
send_msg()
```

Algorithm 2: Redirecting Algorithm

```
Input: message packet in event
pkt ← ev.msg
eth ← pkt.get_protocols(ethernet)
dst_mac ← eth.dst
src_mac ← eth.src
match ← in_port=in_port, eth_dst=dst_mac
if is_IPv4 and is_TCP then
    ipv4_src ← ipv4_pkt.src
    ipv4_dst ← ipv4_pkt.dst
    # change the dst from server1 to server2
    if src_mac == CLIENT['mac'] and dst_mac == SERVER1['mac'] then
        match ← ipv4_src=ipv4_src, ipv4_dst=ipv4_dst
        actions ← eth_dst=SERVER2['mac'], ipv4_dst=SERVER2['ip'], out_port
    end
    # change the src from server2 to server1
    if src_mac == SERVER2['mac'] and dst_mac == CLIENT['mac'] then
        match ← ipv4_src=ipv4_src, ipv4_dst=ipv4_dst
        actions ← eth_src=SERVER1['mac'], ipv4_src=SERVER1['ip'], out_port
    end
end
if mac address is found then
    out_port = mac_to_port[dpid][dst_mac]
else
    out_port = FLOOD
end
actions ← out_port
add_flow ← match, actions, idle_timeout = 5
send_msg()
```

C. Algorithm

IV. Implementation

A. Development Environment

Since the Mininet and Ryu packages are only available on Linux, Ubuntu virtual machines and Oracle VM VirtualBox are used in the development process. The detailed development environment is listed in the table below.

TABLE I
DEVELOPMENT ENVIRONMENT

Field	Specification
Virtual Machine OS	Ubuntu 20.04.4
CPU	AMD Ryzen7 5800x
RAM	16GB@3600 MHz
IDE	PyCharm 2024.3
Python	Python 3.8
SDN Controller	Ryu 4.34

• **Used Python Modules**: mininet, ryu.

B. Programming Skills

• **Object Oriented Programming**: The Ryu framework is implemented in a object-oriented way, each SDN controller in Ryu framework is an instance of Ryu application, with custom event handler to act as a SDN switch.

• **Event System**: In the Ryu framework, messages and actions are abstracted into events, a Ryu application is supposed to handle them by using '@set_ev_cls' decorator to appropriate methods. These methods are used to process events like the "OpenFlow Packet In" event.

C. Steps of Implementation

1) **Analysis specification**: The task requires to build a simple SDN network with three hosts and one SDN controller. The SDN controller can forward or redirect the network traffic to different destinations.
2) **Create network topology**: The network topology is built using Mininet library according to the figure provided in the task sheet.
3) **Understand Ryu framework**: In this step, the "simple_switch_13.py" is used an example, to learn how Ryu application works and how it acts SDN switch.
4) **Program the SDN controller**: After learning how it works, attempts are made to wirte code to manipulate the flow table, and implement a functional SDN switch to forward or redirect the network traffic.
5) **Debugging and testing**: Test the finished SDN switch in the virtual machine, find bugs and fix them.

D. Actual Implementation

1) Forwarding SDN controller: The forwarding SDN controller switch is implemented as a self-learning switch, which means it can learn the host port and MAC address relation automatically. This is achieved by maintaining a MAC address and port relation table. When the switch receives an incoming packet, it will update the table with the source MAC address and the corresponding port. If the destination MAC address is not in the MAC address table, the switch floods all ports except the source port, to make sure the packet can get to the target destination. If the destination MAC address is found in the table, then the packet is forwarded to the corresponding port directly. Then the controller installs the flow entry to the flow table with an idle timeout of 5 seconds.

2) Redirecting SDN controller: The base redirecting SDN controller is implemented similarly to the forwarding controller, except the TCP packet sent from client to server1 is redirected to server2. This is done by using the Ryu framework's packet parser. When the redirecting SDN controller receives a TCP packet from the client, and the packet destination is server1, it will then modify the packet destination MAC and IP to server2's MAC and IP. After the packet modification, the packet is intercepted and sent to the server2's port, then the client-to-server redirection is complete.

When server2 responds the client's incoming packet, the SDN controller will modify the packet destination MAC and IP to server1's MAC and IP, and then forward it to the client. This modification is needed because the SDN controller needs to trick the client to think the packet is still being sent from server1 rather than server2.

E. Difficulties and Solutions

During the development of programming the actual SDN controller application, we found that the Ryu library in Python is too abstracted and hard to understand, but the official documentation is very limited. To overcome this, we referred to the source code of the Ryu framework and read other documentation such as the Ryubook. This helped us understand how the Ryu framework and Ryu SDN controller application works.

The other difficulty is that when testing the redirect functionality, the network traffic was not being redirected as expected. We found that the TCP three-way handshake cannot be established properly, the client does not process the SYN-ACK segment from server2. This is because the SDN controller needs to fake the responding packet from server2 as server1. After this, the traffic is successfully redirected to server2.

V. Testing and Results

A. Testing Environment

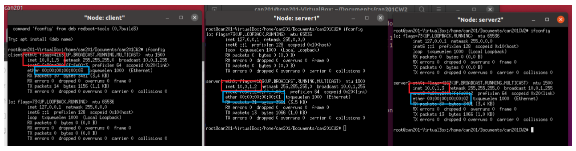
The testing environment is shown in the table II:

TABLE II
Testing Environment

Field	Description
CPU	Intel(R) Core(TM) i9-12950HX @1C
RAM	4GB @4800MHz
Virtual Machine OS	Ubuntu 20.04.4 LTS
Python Version	Python 3.8.10
SDN Controller	Ryu 4.34

B. Testing Steps

1) Task 1: Open a terminal in the program directory and run `sudo python3 networkTopo.py` to establish the SDN through Mininet. This will open six XTerm windows corresponding to the client, two servers, the switch, and the controller. As shown in Fig.2, you can confirm that the client, Server1, and Server2 are using the correct IP and MAC addresses by running `ifconfig` in the XTerm windows.



```
 command 'fconfig' from deb redboot-tools (0.7build3)

Try: apt install <deb name>

root@can201-VirtualBox:/home/can201/Documents/can201CW2# 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 fe80::200:ff:fe00:3  prefixlen 64  scopeid 0x20<link>
        ether 00:00:00:00:00:03  txqueuelen 1000  (Ethernet)
        RX packets 30  bytes 3491 (3.4 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 14  bytes 1156 (1.1 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/Documents/can201CW2# ifconfig
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

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 fe80::200:ff:fe00:1  prefixlen 64  scopeid 0x20<link>
        ether 00:00:00:00:00:01  txqueuelen 1000  (Ethernet)
        RX packets 31  bytes 3581 (3.5 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 13  bytes 1066 (1.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@can201-VirtualBox:/home/can201/Documents/can201CW2# 
```

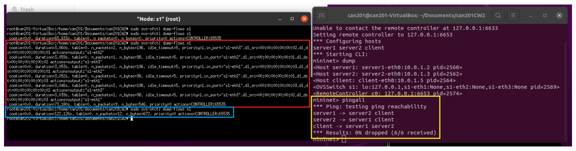
```
root@can201-VirtualBox:/home/can201/Documents/can201CW2# ifconfig
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        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

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 fe80::200:ff:fe00:2  prefixlen 64  scopeid 0x20<link>
        ether 00:00:00:00:00:02  txqueuelen 1000  (Ethernet)
        RX packets 30  bytes 3401 (3.4 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 13  bytes 1066 (1.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@can201-VirtualBox:/home/can201/Documents/can201CW2# 
```

Fig. 2. Task 1 Screenshot

2) Task 2: Run the SDN controller application based on the Ryu framework (e.g., `sudo ryu-manager ryu_forward.py`) to control the switch's behavior. Then, execute `pingall` in the Mininet terminal. As shown in Fig.3, all nodes successfully connect. Additionally, running `sudo ovs-ofctl -O openflow13 dump-flows s1` will display the corresponding flow table in the switch. After 5 seconds, check the flow table again; as `idle_timeout=5` is set, all flow entries will have expired and been removed.



```
root@can201-VirtualBox:/home/can201/Documents/can201CW2# sudo ovs-ofctl dump-flows s1
root@can201-VirtualBox:/home/can201/Documents/can201CW2# sudo ovs-ofctl dump-flows s1
 cookie=0x0, duration=65.833s, table=0, n_packets=0, n_bytes=0, priority=0 actions=CONTROLLER:65535
root@can201-VirtualBox:/home/can201/Documents/can201CW2# sudo ovs-ofctl dump-flows s1
 cookie=0x0, duration=3.060s, table=0, n_packets=2, n_bytes=196, idle_timeout=5, priority=1,in_port="s1-eth3",dl_src=00:00:00:00:00:02,dl_d
st=00:00:00:00:00:01 actions=output:"s1-eth2"
 cookie=0x0, duration=3.059s, table=0, n_packets=1, n_bytes=98, idle_timeout=5, priority=1,in_port="s1-eth2",dl_src=00:00:00:00:00:01,dl_ds
t=00:00:00:00:00:02 actions=output:"s1-eth3"
 cookie=0x0, duration=3.053s, table=0, n_packets=2, n_bytes=196, idle_timeout=5, priority=1,in_port="s1-eth1",dl_src=00:00:00:00:00:03,dl_d
st=00:00:00:00:00:01 actions=output:"s1-eth2"
 cookie=0x0, duration=3.052s, table=0, n_packets=1, n_bytes=98, idle_timeout=5, priority=1,in_port="s1-eth2",dl_src=00:00:00:00:00:01,dl_ds
t=00:00:00:00:00:03 actions=output:"s1-eth1"
 cookie=0x0, duration=3.046s, table=0, n_packets=2, n_bytes=196, idle_timeout=5, priority=1,in_port="s1-eth1",dl_src=00:00:00:00:00:03,dl_d
st=00:00:00:00:00:02 actions=output:"s1-eth3"
 cookie=0x0, duration=3.039s, table=0, n_packets=1, n_bytes=98, idle_timeout=5, priority=1,in_port="s1-eth3",dl_src=00:00:00:00:00:02,dl_ds
t=00:00:00:00:00:03 actions=output:"s1-eth1"
 cookie=0x0, duration=73.180s, table=0, n_packets=9, n_bytes=546, priority=0 actions=CONTROLLER:65535
root@can201-VirtualBox:/home/can201/Documents/can201CW2# sudo ovs-ofctl dump-flows s1
 cookie=0x0, duration=122.126s, table=0, n_packets=12, n_bytes=672, priority=0 actions=CONTROLLER:65535
root@can201-VirtualBox:/home/can201/Documents/can201CW2# 
```

```
Unable to contact the remote controller at 127.0.0.1:6633
Setting remote controller to 127.0.0.1:6653
*** Configuring hosts
server1 server2 client
*** Starting CLI:
mininet> dump
<Host server1: server1-eth0:10.0.1.2 pid=2560>
<Host server2: server2-eth0:10.0.1.3 pid=2562>
<Host client: client-eth0:10.0.1.5 pid=2564>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None,s1-eth3:None pid=2569>
<RemoteController c0: 127.0.0.1:6653 pid=2574>
mininet> pingall
*** Ping: testing ping reachability
server1 -> server2 client
server2 -> server1 client
client -> server1 server2
*** Results: 0% dropped (6/6 received)
mininet> 
```

Fig. 3. Task 2 Screenshot

3) Task 3: The completion of Task 3 will be demonstrated in Task 4.

4) Task 4: Run `ryu_forward.py` in the controller to ensure that when the switch receives the first Packet_In SDN packet, it adds a flow entry to the SDN switch and forwards the packet. Then, run `python3 server.py` on both servers. After waiting for 5 seconds (to ensure the flow table is refreshed), run `python3 client.py` on the client. As shown in Fig.4, all subsequent traffic sent from the client to Server1 is correctly forwarded to Server1. By examining the switch's flow table, you can confirm that the forwarding rules have been added.

Before establishing the client-server connection, if you use root privileges to open Wireshark to monitor the client's port `s1-eth1` and then run the client, you can capture packets and analyze the latency of the TCP three-way handshake. As shown in Fig.5, the first SYN segment can be set as the time reference, allowing direct calculation of the ACK segment's latency.

5) Task 5: Run `ryu_redirect.py` in the controller to ensure that when the switch receives the first Packet_In SDN packet, it adds a flow entry for redirection to the SDN switch. Then, run `python3 server.py` on both servers.

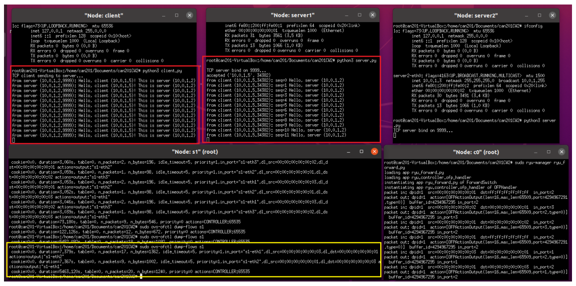
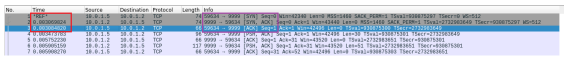


Fig. 4. Task 4.1 Screenshot



```
No.  Time         Source    Destination Protocol Length Info
   1 *REF*        10.0.1.5  10.0.1.2    TCP          74 59634 → 9999 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM=1 TSval=930875297 TSecr=0 WS=512
   2 0.003069824  10.0.1.2  10.0.1.5    TCP          74 9999 → 59634 [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460 SACK_PERM=1 TSval=2732983649 TSecr=930875297 WS=512
   3 0.003084820  10.0.1.5  10.0.1.2    TCP          66 59634 → 9999 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=930875300 TSecr=2732983649
   4 0.003473783  10.0.1.5  10.0.1.2    TCP          96 59634 → 9999 [PSH, ACK] Seq=1 Ack=1 Win=42496 Len=30 TSval=930875301 TSecr=2732983649
   5 0.005752230  10.0.1.2  10.0.1.5    TCP          66 9999 → 59634 [ACK] Seq=1 Ack=31 Win=43520 Len=0 TSval=2732983651 TSecr=930875301
   6 0.005905159  10.0.1.2  10.0.1.5    TCP         117 9999 → 59634 [PSH, ACK] Seq=1 Ack=31 Win=43520 Len=51 TSval=2732983651 TSecr=930875301
   7 0.005908270  10.0.1.5  10.0.1.2    TCP          66 59634 → 9999 [ACK] Seq=31 Ack=52 Win=42496 Len=0 TSval=930875303 TSecr=2732983651
```

Fig. 5. Task 4.2 Screenshot

After waiting for 5 seconds (to ensure the flow table is refreshed), run `python3 client.py` on the client. As shown in Fig.6, all subsequent traffic sent from the client to Server1 is redirected to Server2. By examining the switch's flow table, you can confirm that the redirection rules have been added.

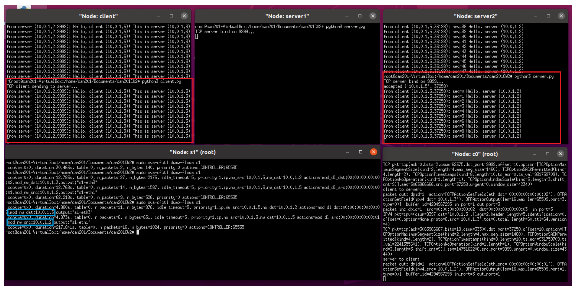
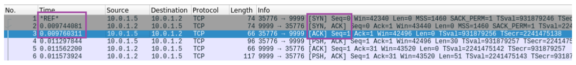


Fig. 6. Task 5.1 Screenshot

Similarly, as shown in Fig.7, packet capturing with Wireshark can measure the network latency under redirection.



```
No.  Time         Source    Destination Protocol Length Info
   1 *REF*        10.0.1.5  10.0.1.2    TCP          74 35776 → 9999 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM=1 TSval=931879246 TSec
   2 0.009744081  10.0.1.2  10.0.1.5    TCP          74 9999 → 35776 [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460 SACK_PERM=1 TSval=224
   3 0.009760311  10.0.1.5  10.0.1.2    TCP          66 35776 → 9999 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=931879256 TSecr=2241475138
   4 0.011297844  10.0.1.5  10.0.1.2    TCP          96 35776 → 9999 [PSH, ACK] Seq=1 Ack=1 Win=42496 Len=30 TSval=931879257 TSecr=2241475
   5 0.011562200  10.0.1.2  10.0.1.5    TCP          66 9999 → 35776 [ACK] Seq=1 Ack=31 Win=43520 Len=0 TSval=2241475142 TSecr=931879257
   6 0.011573924  10.0.1.2  10.0.1.5    TCP         117 9999 → 35776 [PSH, ACK] Seq=1 Ack=31 Win=43520 Len=51 TSval=2241475143 TSecr=93187
```

Fig. 7. Task 5.2 Screenshot

C. Test Result

In conclusion, we successfully implemented the correct network topology, node inter-connectivity, and accurate forwarding or redirection behavior in the switch. By analyzing timestamps of the TCP triple handshake, we found that 99.83% of the time is consumed during the first two handshakes. This is because the initial connection requires the switch to send information to the controller, wait for decisions, and install rules. The third handshake, taking only 0.17% of the total time, is much faster because the correct flow table is already installed, accelerating forwarding by approximately two orders of magnitude.

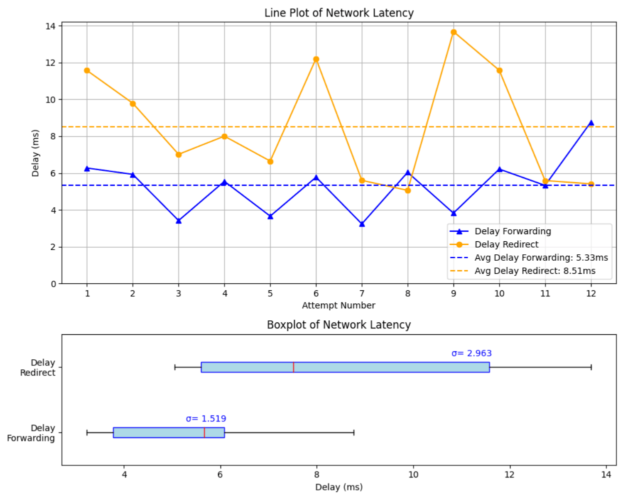


Fig. 8. Network Latency Test Statistics

As shown in Fig.8, we conducted 12 tests for both forwarding and redirection to measure average latency and standard deviation. The results indicate that redirection's average latency is about 2ms higher than forwarding, and its standard deviation is also larger. This is likely because modifying the packet's destination address requires more processing time, leading to variability in forwarding time. Further automated testing is needed to confirm this hypothesis and eliminate the impact of network fluctuations and other factors.

VI. Conclusion

Through the process of design, implementation, and testing processes, our team successfully developed an SDN architecture with a controller for efficient flow control, implementing direct transmission and client message redirection as specified. For performance evaluation, each task was tested multiple times, and the average results and standard deviations were calculated to ensure accuracy.

Future work includes enhancing system robustness by optimizing flow table constraints to prevent flooding and improve resilience against DoS attacks. Second, the forwarding logic can be further refined by implementing load-balancing algorithms such as Weighted Round Robin or Least Connections to better apply to reality. These improvements will ensure reliable SDN performance across diverse environments.

VII. Acknowledgment

We collectively agree that each team member has contributed equally to the project.

References

[1] M. Alsaeedi, M. M. Mohamad, and A. A. Al-Roubaiey, "Toward Adaptive and Scalable OpenFlow-SDN Flow Control: A Survey," *IEEE Access*, vol. 7, pp. 107 346–107 379, 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8784036/
[2] "nginx." [Online]. Available: https://nginx.org/en/
[3] H. Yang, H. Pan, and L. Ma, "A Review on Software Defined Content Delivery Network: A Novel Combination of CDN and SDN," *IEEE Access*, vol. 11, pp. 43 822–43 843, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10103706/