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

Jingyao Zhang
CST XJTLU
Student ID: 2363985
Suzhou, China
Jingyao.Zhang23@student.xjtlu.edu.cn

Zixi Chen
ICS XJTLU
Student ID: 2362352
Suzhou, China
Zixi.Chen23@student.xjtlu.edu.cn

Kaiming Liu
CST XJTLU
Student ID: 2362259
Suzhou, China
Kaiming.Liu23@student.xjtlu.edu.cn

***Abstract*—This project employs the Mininet and Ryu frameworks to design a simplified Software-Defined Network (SDN) aimed at mitigating limitations inherent in traditional network architectures. The implementation encompasses the creation of an SDN topology, simulation of traffic control mechanisms, and development of a custom controller for directed packet forwarding and service redirection. Core activities involve flow rule insertion, traffic analysis, and latency assessment to ensure uninterrupted client-server redirection. The design and implementation are delineated through topological diagrams, pseudo-code, and flowcharts, with subsequent testing validating the system's operational efficacy. Concluding remarks propose potential avenues for future enhancement.**

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

I. Introduction

A. Background

Software-Defined Networking (SDN) represents an emerging network architecture paradigm characterized by the decoupling of the control plane from the data plane. This separation fundamentally alters the paradigm for network design and management. In contrast to traditional network devices, which integrate control logic and data forwarding within each individual unit, SDN introduces a logically centralized controller. This controller enables network administrators to dynamically manage traffic flows and adjust policies in real time through software-based interfaces.

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. Contribution

To successfully complete the project, a thorough understanding and command of SDN's core principles are required. This necessitates familiarity with the Ryu controller programming framework, specifically mastering techniques for event handling and the dynamic deployment of flow table entries. In essence, the project synthesizes SDN theory, network simulation, controller programming, algorithm design, and network performance analysis, thereby comprehensively enhancing both theoretical understanding and practical proficiency in SDN technologies.

II. Related Work

Related work indicates that Software-Defined Networking (SDN) facilitates logically centralized control and fine-grained traffic management by decoupling the control plane from forwarding devices. OpenFlow provides a standardized protocol for controllers to install and update flow-table entries in switches, enabling event-driven behaviors (e.g., reacting to new flows) and making mechanisms such as policy-based forwarding and header rewriting practically implementable [1]. Broader surveys on SDN further categorize this ecosystem (e.g., southbound APIs, controller platforms, network applications) and discuss associated challenges, including scalability, consistency, debugging, and security—issues that directly influence controller logic and the design of rule timeouts [2].

For evaluation and rapid prototyping, Mininet is widely adopted as it emulates networks on a single machine while executing real protocol stacks, thereby offering a convenient environment for validating OpenFlow rule installation and measuring TCP-level effects, such as handshake latency [3]. Within this experimental paradigm, transparent redirection is commonly implemented through paired forward and reverse flow rules that rewrite IP/MAC headers (in a NAT-like manner) to maintain session transparency while steering traffic to an alternate server—consistent with the controller-driven redirection approach described in the current project context [2], [3].

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

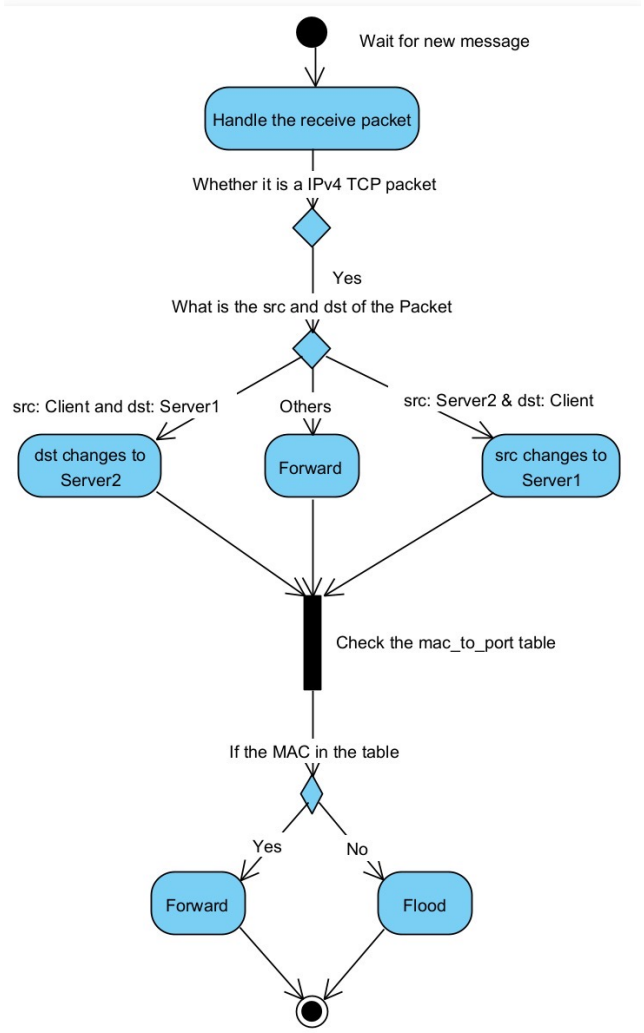

Fig. 1: Redirecting Activity Diagram

The SDN architecture implemented in this project follows a centralized control plane model, which fundamentally separates network control logic from the data forwarding functions. This architectural paradigm enables dynamic network management and programmatic control over traffic flow. The network structure is shown in Fig.1. The following is about the explanation of main Fig. 1. Redirecting Activity Diagram elements:

1) SDN Controller: The SDN controller functions as the centralized control core of the network, implemented utilizing the open-source Ryu framework. It adopts a three-layer architecture that delivers essential functionalities through its northbound API to enable dynamic traffic flow management. The controller communicates with switches via the OpenFlow protocol to process critical events such as `Packet_In` and `Packet_Out`. This interaction facilitates the controller in installing flow table entries within switches to administrate, forward, or redirect traffic.

2) SDN Switch: The SDN switch maintains flow tables specified by the OpenFlow standard. Each flow table entry incorporates essential components including match fields, priority, and actions. Match fields are configurable to filter traffic according to IPv4 addresses or TCP ports, whereas the corresponding actions dictate packet processing. When a packet fails to match any existing rule, the switch transmits a `Packet_In` message to the controller to solicit further directives. Following the reception of a `Packet_Out` response, the switch proceeds to update its flow table entries.

3) Hosts: The network hosts comprise one client and two servers, each configured with unique IP and MAC addresses for their interfaces. The client initiates traffic towards the servers, with bidirectional communication realized through socket programming. These hosts act as terminals for evaluating traffic forwarding and redirection mechanisms, and are consistently managed by the SDN controller.

B. The operation flow

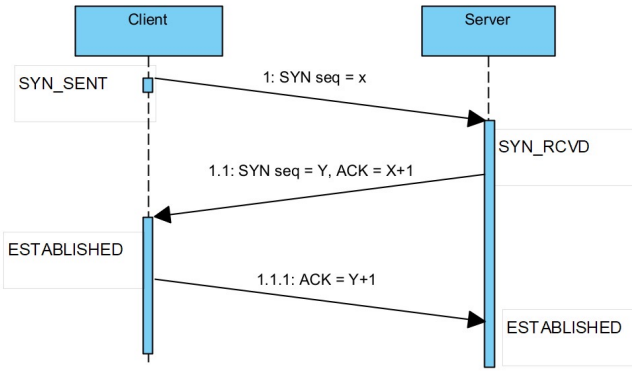

Fig. 2: Flowchart for SDN

The system's operational workflow involves constructing a network with one client, two servers, an OpenFlow switch, and a central SDN controller built upon the Ryu framework. All entities are deployed within a Mininet emulation environment. Host interfaces are pre-configured with unique IP and MAC addresses. The controller communicates with the switch using the OpenFlow 1.3 protocol to manage flow tables dynamically. The core functionality addresses two objectives: enabling standard packet forwarding and implementing transparent packet redirection. Initialization requires starting the servers to listen for TCP connections and the client to initiate a socket. The controller and switch are then launched. A default table-miss flow entry is configured to send unmatched packets to the controller via `Packet_In` messages. For basic forwarding, the controller identifies the egress port to the target server (Server1) and installs a corresponding flow entry, directing subsequent matching traffic without further intervention. The redirection mechanism is more complex, aiming to transparently reroute client traffic destined for Server1 to Server2. This requires the controller to install flow entries that perform packet header modification. For the client-to-server path, the

destination address is rewritten from Server1 to Server2. For the return path, the source address in packets from Server2 is rewritten to appear as Server1, thereby maintaining session transparency for the client. This programmatic control demonstrates the key SDN capability of dynamically managing traffic paths.

C. Algorithm

Algorithm 1: Redirecting Algorithm

Input: packet_in_event, topology parameters (CLIENT_IP, SERVER1_IP, etc.)
Output: flow rules installed, packet forwarded
$pkt \leftarrow$ extract_packet($packet\_in\_event$)
$eth \leftarrow pkt$.get_ethernet_header()
$ipv4 \leftarrow pkt$.get_ipv4_header()
$tcp \leftarrow pkt$.get_tcp_header()
if $tcp \neq$ *null* ***and*** $tcp.is\_syn()$ **then**
  if $ipv4.src =$ *CLIENT_IP* ***and*** $ipv4.dst =$ *SERVER1_IP* **then**
    $flow\_key \leftarrow (ipv4.\text{src}, ipv4.\text{dst}, tcp.\text{src\_port}, tcp.\text{dst\_port})$
    if $flow\_key \notin installed\_flows$ **then**
      $match\_fwd \leftarrow$ {ipv4_src : CLIENT_IP, ipv4_dst : SERVER1_IP, tcp_src : tcp.src_port, tcp_dst : tcp.dst_port}
      $actions\_fwd \leftarrow$ [set_field(ipv4_dst = SERVER2_IP), set_field(eth_dst = SERVER2_MAC), output(PORT_SRV2)]
      add_flow($datapath, match\_fwd, actions\_fwd$,
      priority $= 1$, idle_timeout $= 5$)
      $match\_rev \leftarrow$ {ipv4_src : SERVER2_IP, ipv4_dst : CLIENT_IP, tcp_src : tcp.dst_port, tcp_dst : tcp.src_port}
      $actions\_rev \leftarrow$ [set_field(ipv4_src = SERVER1_IP), set_field(eth_src = SERVER1_MAC), output(PORT_CLIENT)]
      add_flow($datapath, match\_rev, actions\_rev$,
      priority $= 1$, idle_timeout $= 5$)
      installed_flows.add($flow\_key$)
    else
      print("$Flowalreadyexists$: ", $flow\_key$)
    end
    packet_out($datapath, msg, actions\_fwd$)
    return *SUCCESS*
  else
    flood_packet($datapath, msg$)
  end
else
  flood_packet($datapath, msg$)
end

IV. Implementation

A. Implementation Environment

We implemented the program with a laptop, the host environment is shown as followed:

TABLE I: Development Environment Configuration

Field	Description
Operating System	Microsoft Windows 11
CPU	12th Gen Intel(R) Core(TM) i7-12700H
Memory	16.0 GB
IDE	PyCharm 2023.3
Python Version	3.8

TABLE II: Experimental Environment and Tools

Field	Description
Simulation Platform	Mininet (Python API)
Switch	Open vSwitch (OVS)
Controller Framework	Ryu (OpenFlow 1.3)
Transport Protocol	TCP
Packet Capture Tool	tcpdump

B. Step of Implementation

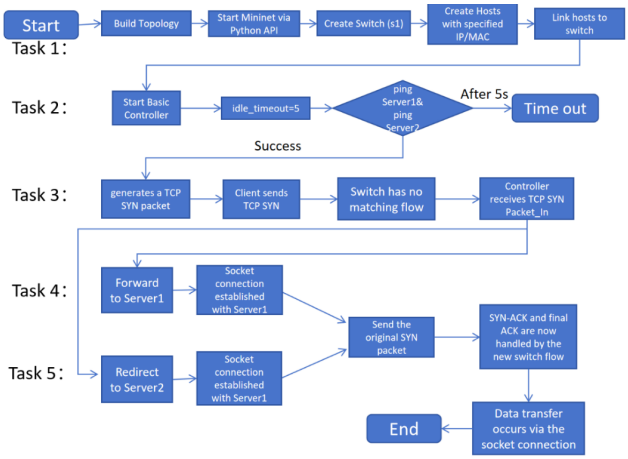

Fig. 3: Program Flow Chart

In Task 1, the initial step is to create a Python program using the Mininet library to construct a specified SDN network topology. The program must configure the Client, Server1, and Server2 with the designated IP and MAC addresses as required. In Task 2, program and launch an SDN controller application using the Ryu framework. The core function of this application is to ensure all hosts are reachable to each other via 'ping'. Every installed flow entry, excluding the default table-miss entry, must be configured with an idle timeout of 5 seconds to allow automatic removal after inactivity. For Task 3, deploy the given socket programs by running server.pyon both Server1 and Server2, and client.pyon the Client. It is critical to initiate the client program 5 seconds after the initial ping tests to ensure the flow entries generated by ICMP traffic

have timed out and been removed, forcing subsequent TCP traffic to be handled by the controller. In Task 4, the SDN controller application must be programmed with specific logic for handling new TCP connections. Upon receiving the first Packet_In message from the switch, which is triggered by a TCP SYN segment from the Client to Server1, the controller should create a corresponding flow entry. This entry is then installed into the switch to ensure all subsequent traffic from the Client to Server1 is correctly forwarded to Server1. Finally, the controller sends the original SYN segment back to the switch encapsulated in a Packet_Out message. Following this, use a packet capture tool on the Client to measure the network latency of the TCP three-way handshake for this connection. Task 5 modifies the controller logic programmed in Task 4.1. When the first TCP SYN `Packet_In`(from Client to Server1) is received, the controller should create and install a flow entry that redirects all following traffic from the Client intended for Server1 to Server2 instead. The SYN segment is then sent out via `Packet_Out` towards Server2. Similarly, the networking latency for this redirected TCP handshake must be calculated using packet capture.

C. Programming Skills

- **Object-Oriented Programming (OOP)** The SDN project demonstrates strong object-oriented design through the implementation of Ryu controller applications. The RedirectControllerclass extends RyuApp, utilizing inheritance to leverage the Ryu framework's OpenFlow functionality while implementing custom traffic engineering logic. Encapsulation is evident in the class structure, where topology parameters (IP/MAC addresses, port mappings) are stored as class attributes, and flow state management is handled through private collections like installed_flows. Polymorphism is achieved through decorator-based event handlers (@set_ev_cls) that dynamically route different packet types to appropriate processing methods. The abstraction layer converts raw network packets into Python objects (Ethernet, IPv4, TCP classes), hiding protocol complexity and providing a clean API for network programming. This OOP approach enables modular, maintainable code where new features can be added through class extension rather than modification.
- **Parallel** Parallel Processing Analysis: The SDN controller employs an event-driven architecture that enables implicit parallelism. Its async event loop handles multiple switch connections concurrently, while the decoupled control/data planes allow simultaneous processing—the controller installs flows while switches forward packets independently. Each TCP connection is handled as an isolated unit, enabling parallel flow management. The OOP design with encapsulated state supports potential distributed scaling across controller clusters, where different instances can process events in parallel while maintaining consistent network policies through shared state management. This foundation enables high-throughput packet processing without blocking operations.

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 frame work'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

In implementing large file transfer functionality, the most critical challenge lies in ensuring reliability while preventing memory overflow. This challenge is addressed through the design of a chunked transfer mechanism, where large files are segmented into multiple data blocks of a fixed size (e.g., 20 KB) for incremental transmission. Each block is accompanied by precise indexing information to guarantee correct sequencing. On the server side, a two-phase commit strategy is employed: data blocks are initially stored in a temporary directory with transfer logs recorded, and upon reception of all blocks, a comprehensive MD5 checksum verification is performed before the data is moved to permanent storage.

V. Testing and Results

A. Testing Environment

The testing environment is shown in the table II:

B. Testing Steps

1) Task 1: Build the SDN Topology (Mininet): Navigate to the program's directory and open a terminal there. Execute the command

TABLE III: 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

`sudo python3 networkTopo.py`

to set up the SDN (Software-Defined Network) via Mininet. This operation will launch five XTerm windows, which correspond to the client, two servers, the switch, and the controller respectively. As illustrated in Fig.2, you can verify that the client, Server1, and Server2 are assigned the correct IP and MAC addresses by running the ifconfig command in each of their respective XTerm windows.t

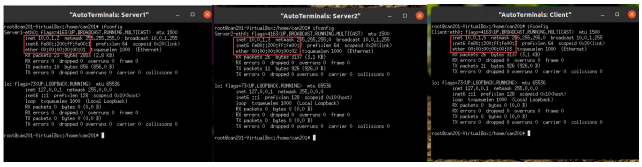

Fig. 4: Task 1 Screenshot

2) Task 2: Forwarding Controller and Flow Table Timeout: 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. 5, all nodes successfully connect.

```
can201@can201-VirtualBox:~$ sudo python3 networkTopo.py
[sudo] password for can201:
Sorry, try again.
[sudo] password for can201:
*** Creating network
*** Adding hosts:
Client Server1 Server2
*** Adding switches:
s1
*** Adding links:
(Client, s1) (Server1, s1) (Server2, s1)
*** Configuring hosts
Client Server1 Server2
*** Starting controller
c0
*** Starting 1 switches
s1 ...

*** Host IP/MAC 信息:
Client: IP=10.0.1.5, MAC=00:00:00:00:00:03
Server1: IP=10.0.1.2, MAC=00:00:00:00:00:01
Server2: IP=10.0.1.3, MAC=00:00:00:00:00:02
*** 已打开 XTerm 终端：Client / Server1 / Server2 / s1 / Controller
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
Client -> Server1 Server2
Server1 -> Client Server2
Server2 -> Client Server1
*** Results: 0% dropped (6/6 received)
```

Fig. 5: Task 2 Screenshot

C. Task 3: TCP Client/Server Programs

The completion of Task 3 is demonstrated in Task 4.

1) Task 4: Forwarding Verification and Handshake Latency (Forwarding Case): Run `ryu_forward.py` on the controller to ensure that when the switch receives the first Packet-In SDN packet, it installs a flow entry into the 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. 6, all subsequent traffic sent from the client to Server1 is correctly forwarded to Server1. By examining the switch's flow table, the forwarding rules can be confirmed as installed.

Before establishing the client-server connection, Wireshark can be started with root privileges to monitor the client-facing interface (e.g., the client host interface `Client-eth0` or the switch port `s1-eth1`). Then run the client to capture packets and analyze the TCP three-way handshake latency. As shown in Fig. 7, using Wireshark *Set/Unset Time Reference* on the first SYN segment allows direct latency calculation from the subsequent handshake packets.

From the Wireshark timestamps in Fig. 7, $t_{\text{SYN}} = 8.793533933$ s and $t_{\text{SYN/ACK}} = 8.795910042$ s.

$$\text{Delay}_{\text{forward}} = t_{\text{SYN/ACK}} - t_{\text{SYN}} \approx 2.376 \text{ ms}. \quad (1)$$

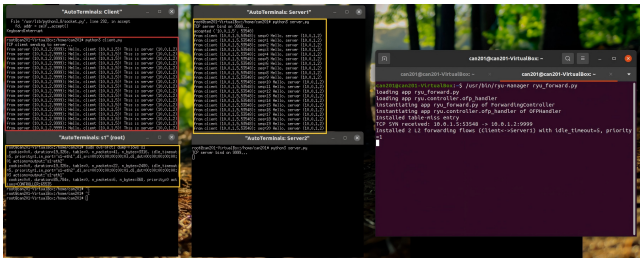

Fig. 6: Task 4.1 Screenshot

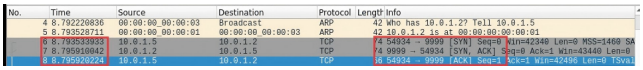

Fig. 7: Task 4.2 Screenshot (Forwarding case TCP handshake)

2) Task 5: Redirection Verification and Handshake Latency (Redirection Case): Run `ryu_redirect.py` on the controller to ensure that when the switch receives the first Packet-In SDN packet, it installs redirection flow entries into the switch.

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. 8, all subsequent traffic sent from the client to Server1 is redirected to Server2. By examining the switch's flow table, the redirection rules can be confirmed as installed. In addition, Server2 receives the client's TCP payload, while the client still observes the peer as Server1 (IP spoofing / NAT-style rewriting).

Similarly, as shown in Fig. 9, packet capturing with Wireshark can be used to measure the TCP three-way handshake latency under the redirection behavior.

From the Wireshark timestamps in Fig. 9, $t_{\text{SYN}} = 13.12198752$ s and $t_{\text{ACK}} = 13.124433497$ s.

$$\text{Delay}_{\text{redirect}} = t_{\text{ACK}} - t_{\text{SYN}} \approx 2.446 \text{ ms}. \quad (2)$$

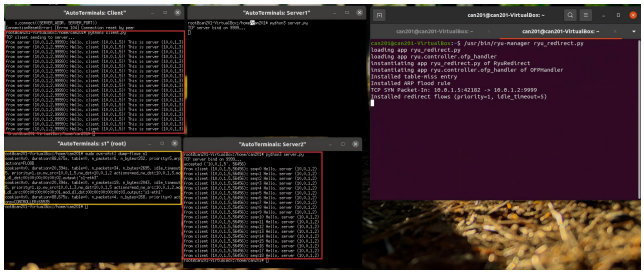

Fig. 8: Task 5.1 Screenshot (Redirect verification)

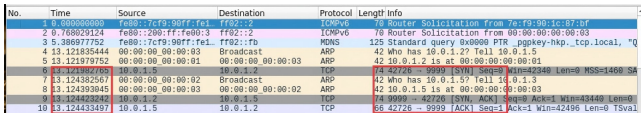

Fig. 9: Task 5.2 Screenshot (Redirection case TCP handshake)

D. Test Result

In conclusion, we successfully implemented the required SDN topology and verified both the forwarding and NAT-style redirection behaviors on the OpenFlow switch. Using Wireshark/tcpdump timestamps of the TCP three-way handshake, we measured the handshake latency under two modes. For the forwarding case, the average handshake delay over 12 trials is $1.498\,\text{ms}$, while for the redirection case it increases to $4.798\,\text{ms}$. As shown in our latency plot with mean reference lines, redirection consistently exhibits higher delay and larger variance, which is expected because packet header rewriting (IP/MAC modification) and the additional forwarding path introduce extra processing and/or path overhead compared with direct forwarding. Overall, the experimental results confirm that the switch rules are correctly installed and that redirection incurs a measurable latency penalty relative to basic forwarding.

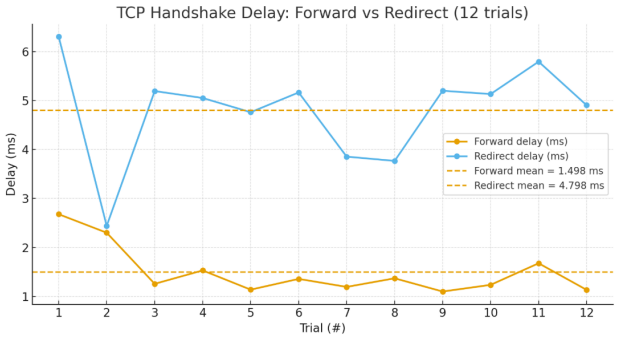

Fig. 10: Network Latency Test Statistics

VI. Conclusion

Through the design, implementation, and rigorous testing phases, we have successfully designed and implemented an SDN architecture featuring a dedicated controller for efficient flow management. This system fulfills the specified requirements for direct data transmission and client message redirection.

To ensure a reliable performance evaluation, each experimental task was executed repeatedly. The average results, accompanied by their standard deviations, were calculated to validate measurement accuracy and consistency.

Future directions for this work include enhancing system robustness by refining flow-table constraints to mitigate table overflow and improve resilience against Denial-of-Service (DoS) attacks. Furthermore, the forwarding logic can be advanced by integrating load-balancing algorithms, such as Weighted Round Robin or Least Connections, to increase the practicality and adaptability of the system in real-world scenarios. These proposed improvements aim to ensure reliable and scalable SDN performance across diverse network environments.

Acknowledgment

In this project, although there were only three of us, we had a clear division of labor and each member fulfilled their duties. Therefore, each of us contributed 33.3%.

References

[1] McKeown, N. et al. “OpenFlow: Enabling Innovation in Campus Networks.” *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, 2008, pp. 69–74. https://ccr.sigcomm.org/online/files/p69-v38n2n-mckeown.pdf.

[2] Kreutz, D. et al. “Software-Defined Networking: A Comprehensive Survey.” *Proceedings of the IEEE*, 2014, arXiv:1406.0440. https://arxiv.org/abs/1406.0440.

[3] Lantz, B., Heller, B., McKeown, N. “A Network in a Laptop: Rapid Prototyping for Software-Defined Networks.” *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets-IX)*, 2010. https://conferences.sigcomm.org/hotnets/2010/papers/a19-lantz.pdf.