Implementation and Evaluation of Traffic Forwarding and Redirection Using Ryu Framework

Chenrui Zhu 2361387 Qi Chen 2363203 Shengyang Yin 2361841 Tianchen Niu 2361034 Youming Yang 2361217

***Abstract*—This coursework implements a simple Software-Defined Networking (SDN) system using Mininet and the Ryu controller to emulate a programmable traffic control mechanism. A three-host topology consisting of a client, Server1, and Server2 is constructed, where the client is only aware of the service on Server1. Two controller applications are developed: the first performs reactive forwarding based on TCP SYN packet detection so that subsequent traffic is forwarded to Server1, while the second redirects the same TCP connection to Server2 by rewriting packet headers and installing bidirectional flow entries. All flow entries, except the table-miss rule, follow the required idle timeout of five seconds. The results demonstrate that both forwarding and redirection behaviours can be correctly triggered by the first SYN packet.**

I. Introduction

A. Project Task Specification

Software-Defined Networking (SDN) makes a fundamental change in network architecture by separating the control plane from the data plane [1]. To complete the target course assignment, the implementation process consists of three main tasks:

1) **Build SDN Topology.** We built a topology in Mininet containing 3 hosts: a client, Server1, and Server2, all connected via a single OpenFlow switch. IP and MAC addresses were manually configured as required by the assignment, and their basic network reachability was verified.
2) **Develop SDN Controller Application.** This project involves developing two Ryu-based controllers. The first controller handles forwarding triggered by TCP SYN packets, ensuring subsequent traffic reaches Server1. The second controller redirects traffic that would otherwise be sent from the client to Server1 to Server2 by rewriting IP and MAC headers and installing symmetric flow table entries. All non-default rules include a mandatory five-second idle timeout.
3) **Simulate Traffic and Measure Latency.** We deployed the provided socket client and server applications to the Mininet hosts. We use Wireshark or Tcpdump on the client side to capture packets to measure the latency of the TCP three-way handshake in forwarding and redirection scenarios.

B. Project Challenges

This assignment encountered a series of technical challenges:

1) **Accurate detection of TCP SYN packets.** Since forwarding and redirection are triggered by TCP SYN packets, the controller must accurately parse the TCP flags to establish the correct request across different data traffic.
2) **Maintaining accurate flow redirection.** When TCP traffic destined for Server1 is captured, the TCP traffic originally destined for Server1 is redirected to Server2 by installing the IP address and MAC address of Server2 in the controller.
3) **Timing constraints introduced by idle timeouts.** The five-second idle timeout requires precise execution during program execution. We must ensure that flow entries generated during traffic forwarding are deleted after 5 seconds, and new flow tables are installed only when new connections are established.

C. Practice Relevance

The traffic redirection mechanism implemented in this project has significant practical value in modern network management. For instance, it can be applied to Server Load Balancing, where traffic is dynamically distributed to different servers to prevent overload. Additionally, it is relevant to Network Security, where suspicious traffic can be redirected to a honeypot or a deep packet inspection (DPI) system for further analysis without alerting the client.

D. Contributions

This work completed a fully functional SDN system, including forwarding and redirection capabilities. Key contributions include: A manually configured Mininet topology with pre-configured IP/MAC addresses was created, and a Ryu-based reactive forwarding controller was implemented. This controller, triggered by TCP SYN packets, successfully performs forwarding and redirection functions. Furthermore, packet capture latency analysis and experimental verification were conducted to reflect the system's performance in forwarding and redirection.

II. Related Work

To achieve connection management and dynamic redirection of network traffic, this experiment adopts the RYU controller framework based on SDN. According to the research of Asadollahi et al. [2], RYU establishes a control channel with the switch through a southbound interface (such as OpenFlow), and is capable of creating, parsing and processing OpenFlow

messages. At the same time, it can also centrally manage message events from the switching device. Moreover, RYU also closely collaborates with various protocols to make it suitable for building custom flow table rules and implementing flexible network control logic [2]. Therefore, IN the experiment, RYU dynamically issues the flow table based on the Packet IN/OUT events reported by the switch, thereby achieving policy control such as data forwarding and path redirection. Furthermore, research shows that the RYU controller can also support TCP connection migration and switching mechanisms by modifying IP and MAC addresses, making it more stable and reliable in network traffic engineering, link switching, and fault-tolerant scenarios [3] [4].

III. Design

A. Network System Design Diagram

This diagram (Figure 1) details the sequential decision logic within the SDN controller's packet handler. It starts with learning the source MAC address, and then proceeds through a series of binary classifications. The first branch handles ARP packets via flooding. For other packets, the logic checks if the destination MAC is known; if not, the packet is flooded. If the MAC is known, the process further filters for IPv4 traffic. For qualifying IPv4 packets, a final distinction is made between TCP SYN segments—which trigger the installation of a high-priority, precise flow entry—and other IPv4 packets, which result in a standard flow entry. Generally speaking, this structure enables the controller to efficiently make context-aware forwarding decisions, from basic Layer 2 learning to application-aware flow steering.

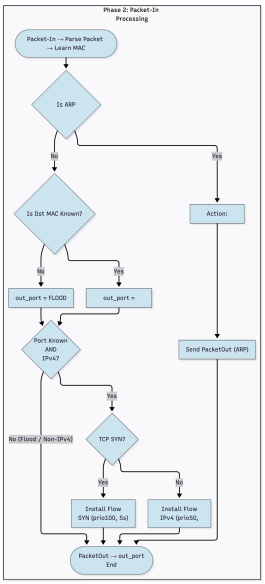


Fig. 1: SDN System Architecture Design

B. Workflow of the Solution

The solution our team provides implements an SDN controller that dynamically manages network flows through a three-phase operational workflow. In the first phase, Mininet establishes the virtual network topology according to the specified IP and MAC addresses. Then the OpenFlow switch establishes a secure connection with the Ryu SDN controller, which installs a default table-miss flow entry to direct unmatched packets to the controller for processing. When a packet arrives at the switch, the packet processing workflow begins and searches for a matching flow entry. If the switch cannot find a match, it will trigger a table-miss condition and send a Packet-In message to the controller. The controller then parses the packet headers, including Ethernet, IP, and TCP layers, updates its MAC-to-port mapping table through MAC learning, and determines the appropriate forwarding strategy based on packet type. Following this decision, the controller installs the corresponding flow entry with a 5-second idle timeout. It enables the switch to forward subsequent packets using the installed flow rules, instead of being intervened by the controller.

In our design, two distinct traffic control modes decide the forwarding behavior. In the forwarding mode, TCP SYN packets whose destination is Server 1 trigger the installation of precise flow entries that match the exact TCP five-tuple (source and destination IP addresses, source and destination ports, and protocol). As a result, subsequent packets follow the established path autonomously. In contrast, the redirection mode transparently redirects TCP SYN packets intended for Server 1 to Server 2. We achieve this process cleverly through bidirectional flow modification, where the forward path alters the destination IP and MAC addresses to those of Server 2, while the reverse path modifies the source IP and MAC addresses to disguise as Server 1. This approach allow the client to remain unaware of the redirection. To show this, we used Wireshark to capture packets, and as we expected, it was consistently the IP addresses 10.0.1.5 and 10.0.1.2 communicating with each other.

C. Algorithm for SDN Controller

The core algorithm for network traffic redirection function is implemented through the following pseudo-code:

Algorithm 1 Flow Installation Algorithm

Require: Datapath $datapath$, priority $priority$, match fields $match$, actions $actions$

Ensure: Flow entry installed in switch with specified parameters

$ofproto \leftarrow datapath.ofproto$
$parser \leftarrow datapath.ofproto\_parser$
$inst \leftarrow [parser.OFPInstructionActions$
    (ofproto.OFPIT_APPLY_ACTIONS, actions)]
        $mod \leftarrow parser.OFPFlowMod(datapath = datapath, priority = priority,$
                $match = match, instructions = inst, idle\_timeout = 5)$
        $datapath.send\_msg(mod)$
        Log flow installation for monitoring purposes
        return SUCCESS

Algorithm 2 SDN Traffic Redirection Algorithm (condensed)

Require: Packet-In msg, datapath $datapath$, in_port in_port
Ensure: Install flows and forward packets (with redirection)

Parse Ethernet from $msg.data$; get eth_src, eth_dst; $mac\_to\_port[datapath.id][eth\_src] \leftarrow in\_port$
if ARP **then**
    Flood; **return**
end if
if IPv4 **then**
    Parse IP; let ip_src, ip_dst; $ip\_to\_mac[ip\_src] \leftarrow eth\_src$
    if TCP and SYN and $ip\_dst = SERVER1\_IP$ **then**
        $server2\_port \leftarrow mac\_to\_port[datapath.id][SERVER2\_MAC]$
        if $server2\_port$ known **then**
            Install forward flow: match $ip\_src \rightarrow SERVER1\_IP \rightarrow$ rewrite to $SERVER2\_IP$ (idle=5s)
            Install reverse flow: match $SERVER2\_IP \rightarrow ip\_src \rightarrow$ rewrite to $SERVER1\_IP$
            Rewrite packet dst to $SERVER2\_IP/SERVER2\_MAC$; send to $server2\_port$; **return**
        end if
    end if
    $out\_port \leftarrow$ lookup(mac_to_port, eth_dst)
    if out_port known **then**
        Install flow (idle=5s) and forward to out_port
    else
        Flood
    end if
end if

IV. Implementation

A. Development Environment

1) Host Environment

Specification	Windows	macOS
OS	Windows 11	macOS Sequoia 15.6
CPU	Intel(R)Core(TM) i9-14900HX	Apple M3
RAM	32GB	8GB @ 6400MHz
IDE	PyCharm 2025.1.3	PyCharm 2024.3.3

TABLE I: DEV ENVIRONMENT

2) SDN Development Setup

For the SDN development environment, we standardized the IDE tool and version (PyCharm 3.8.10) and used Mininet as the virtual network topology as required. We implemented basic logical operations in the controller using the Ryu architecture.

B. Implementation Steps

The implementation process followed a structured approach:

1) **Establish network topology:** Use Mininet to create a topology consisting of one client, two servers, and SDN core components (one switch and one controller).
2) **Initialize the Ryu controller:** Run a custom controller program using Ryu to enable it to handle OpenFlow messages from the switch.
3) **Packet-In processing:** Implement the handling of Packet-In events triggered by table-miss flow entries in the controller, ensuring the ability to correctly receive and parse data packets when no matching flow entries exist.
4) **Parse packets and install flow entries:** Distinguish packets of different protocols such as ARP, ICMP, and TCP SYN. Learn port information based on source MAC addresses and install corresponding flow entries for subsequent similar traffic to reduce controller load.
5) **Redirection:** Upon detecting a TCP SYN packet sent by the client to Server1, modify the necessary header fields and redirect it to Server2 to achieve SYN redirection.
6) **Test and verify:** Test communication using ping and tcpdump, and verify the correctness of redirection logic by capturing packets and querying the flow table via Wireshark.

C. Programming Paradigms

1) Object-Oriented Programming (OOP)

One of the major features of our code is the application of OOP. We defined a class named `ReactiveController` to encapsulate the methods and attributes of the SDN controller. This class inherits from `app_manager.RyuApp`, allowing it to directly reuse methods from its parent class.

2) Event-Driven Programming

The SDN controller application employs an event-driven mechanism. OpenFlow messages sent by the switch are encapsulated as events. One of the core methods, `packet_in_handler`, responds to these events. When an event occurs, this method executes the control plane logic.

3) Modular Design

Our code is divided into several core modules: the **Topology module** loads host topology information; the **Forward module** is responsible for parsing message types, forwarding packets, and implementing learning functions; and the **Redirection module** adjusts forwarding paths based on preset policies and algorithms.

D. Actual Implementation

1) SDN Forwarding Function

Our forward controller first provides each switch with a learning table mapping MAC addresses to ports. If the learned data is not found, we install the table-miss flow entry via the `switch_features_handler` function to send unknown traffic to the controller. The `_packet_in_handler` function learns the source MAC, finds the target port, and installs the flow entry using the `add_flow` function. In `add_flow`, we added a `buffer_id` and set `idle_timeout=5`. The buffer ID prevents duplicate forwarding, and the idle timeout

mechanism ensures that flow entries are deleted if idle for more than 5 seconds, guaranteeing flow table security.

Inside `_packet_in_handler`, when the switch recognizes an unlearned packet, it identifies the target port and initiates ARP flooding. After receiving the response, the Client learns the MAC address and begins sending data. The controller determines the forwarding port and installs the flow table, completing the learning function. Protocols are prioritized: TCP SYN packets have the highest priority, followed by ordinary IPv4 forwarding (ICMP ping) with priority 50, and finally ordinary packets.

2) SDN Redirection Function

Redirection enables reactive forwarding and TCP SYN redirection. Upon connection, `switch_features_handler` installs a table-miss entry to forward mismatched packets to the controller. `packet_in_handler` updates the `mac_to_port` table for every Packet-In event.

For TCP traffic, the packet parser triggers redirection when it detects a SYN destined for Server1. This process installs two high-priority flow rules via `add_flow`:

1) A rule that overwrites the target IP and MAC, redirecting the SYN to Server2.
2) A reverse rule that overwrites Server2's return packets, making them appear to come from Server1.

These rules use OpenFlow actions such as `OFPActionSetField` and `OFPActionOutput`. After installing the flow, the controller immediately forwards the current message via `OFPPacketOut` to prevent packet loss. All other traffic falls back to default L2 learning.

E. Challenges and Solutions

1) Understanding the Ryu Framework

Difficulty: A significant challenge lay in understanding the Ryu framework. Ryu does not execute code sequentially but relies on OpenFlow messages to trigger event handling functions (e.g., `EventOFPSwitchFeatures`).

Solution: We studied the official documentation to understand flow table priorities, the table-miss mechanism, and the interaction between PacketOut and PacketIn messages, mastering the asynchronous structure.

2) TCP Three-Way Handshake Failure in Redirection

Difficulty: In initial tests, the SYN was redirected to Server2, but the TCP connection failed. Packet capture analysis revealed that while the client sent a SYN to Server1, it received a SYN/ACK from Server2. Since the source IP did not match the expected endpoint (Server1), the client sent an RST packet, interrupting the handshake.

Solution: We implemented a reverse flow table entry. When Server2 replies, the switch modifies the packet header to disguise it as originating from Server1. This allows the client to receive the SYN/ACK from the expected endpoint, successfully establishing the three-way handshake.

V. TESTING AND RESULTS

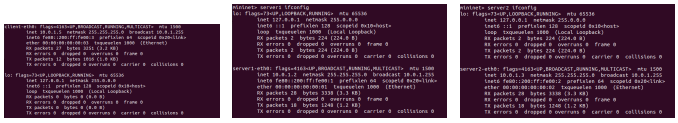
A. Testing environment

	ENV
CPU	4 vCPUs
RAM	4GB
OS	Ubuntu 22.04 LTS
IDE	Terminal
Python Version	Python 3.8.10
Connection	NAT

TABLE II: Test environment configuration

B. Testing steps

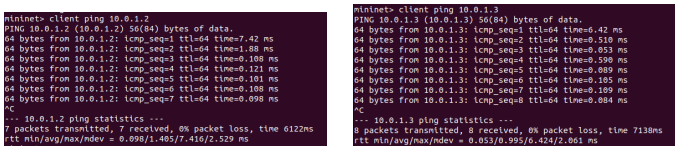
1) Task1 In the program directory, run `sudo python3 networkTopo.py` to establish the SDN topology through Mininet. This will open 6 XTerm Windows, which respectively correspond to the client (client), two servers (server1,server2), switch (s1), and controller (c1). As shown in Fig. 2a, 2b, and 2c, we run `ifconfig` in the XTerm window. It can be confirmed that the hosts use the correct IP and MAC.



(a) Client (b) Server1 (c) Server2

Fig. 2: Network interface configuration of all hosts.

2) Task2 We run `ryu-manager ryu_forward.py` and then use Client to ping the IP addresses of Server1 and Server2 respectively. Fig. 3a and 3b shows that the client successfully sent ICMP Ping requests to Server1 and Server2. The ping results indicate that both servers responded, suggesting that the controller has installed the rules required by the `ryu_forward.py` application and also verifying that the forwarding logic on the switch is working properly. Further, We verify the link topology through `pingall`.



(a) Client ping Server1 (b) Client ping Server2

Fig. 3: Client ICMP reachability test to Server1 and Server2.

3) Task3 We run `server.py` on the xterm terminals of server1 and server2 and then wait for the idle flow entry generated by the ICMP ping datagram in task 2 to expire (5 seconds) to ensure that the previously installed traffic has cleared the switch rules. Then run `client.py` on the xterm terminal of the client. In Fig. 4, a screenshot of the flow entry (flow entry) and traffic logic of Server1 and the client are shown. This confirms the logic of the 5-second timeout of the flow entry: once the idle timeout occurs after the data packet arrives, the forwarding logic of the controller is triggered again. Generate new flow items. As shown in Fig. 5, the traffic from client was successfully forwarded to the server.

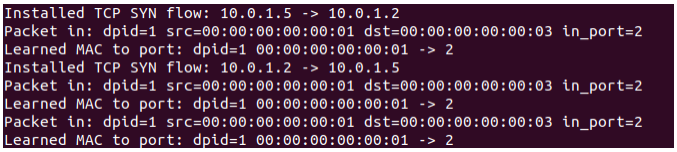
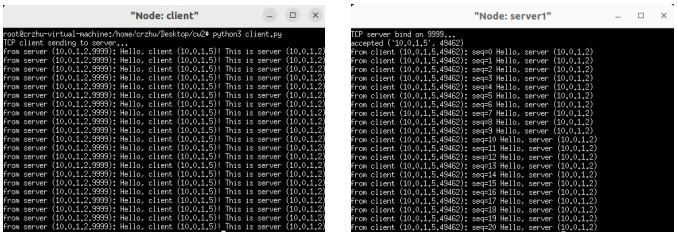


Fig. 4: flow entry



(a) Client reply (b) Server1 reply

Fig. 5: Traffic behavior during forwarding testing.

4) Task4

Task 4.1: According to Fig. 4 (the test results completed in Task 3), we successfully verified that the first TCP SYN successfully triggered `Packet_In`, and the SDN controller could correctly install the bidirectional flow tables of Client→Server1 and Server1→Client. The controller correctly forwards the initial SYN using `Packet_Out`. All subsequent TCP traffic is directly forwarded through the switch flow table without triggering `Packet_In`.

Task 4.2: In the xterm terminal of the client, execute `tcpdump -i client-eth0 -w forward.pcap`, specify the packet capture at the `eth0` network interface of the listening client, and directly write the captured packets into `forward.pcap`. It is convenient to directly analyze the data using wireshark. Fig. 6 shows that 67 packets were successfully received in our verification without any packet loss. The analysis of the packet capture results(Fig. 7) will be reflected in the testing result.

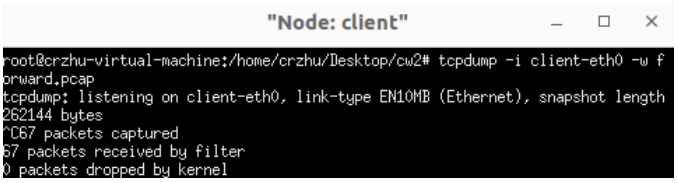


Fig. 6: Packet Capture Results for Forwarding Case(tcpdump)

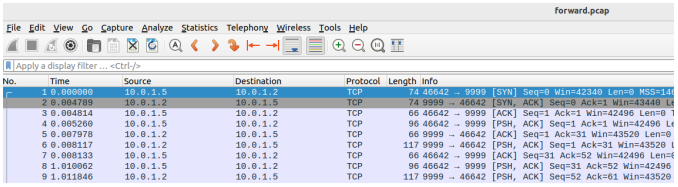
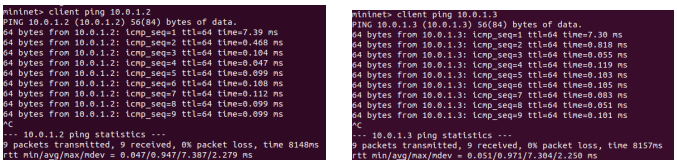


Fig. 7: Packet Capture Results for Forwarding Case(wireshark)

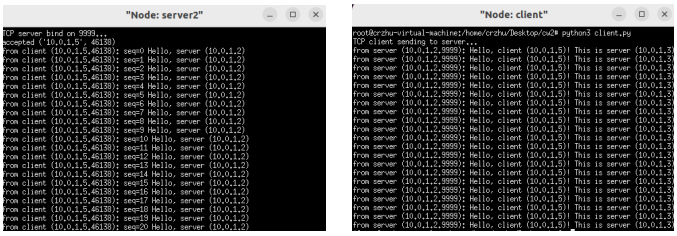
5) Task5 Task 5.1: We run `ryu_redirect.py` on the Controller and use the Client ping the IP address of Server1 and the IP address of Server2. Fig. 8a and 8b shows that the client successfully pings Server1 and Server2 under the `ryu_redirect.py` application of the controller.



(a) Client ping Server1 (b) Client ping Server2

Fig. 8: Client ICMP reachability test to Server1 and Server2.

As before, we run `server.py` on both Server1 and Server2, and wait for the idle timeout (i.e., 5 seconds) of the flow entry generated by the previous ICMP ping. Then we run `client.py` on the Client and initiate a new traffic redirection mode. We observed that the client successfully passed traffic to the specified server2 (Fig. 9a and 9b) and no flow to server1, which explains the successful installation of the redirection rule by the controller.



(a) Client reply (b) Server2 reply

Fig. 9: Traffic behavior during redirection testing.

Furthermore, by observing the flow table output of the controller, we verified that the controller can correctly detect TCP connection initiation and execute server redirection. As shown in the runtime log (Fig. 10), when a client with IP of 10.0.1.5 sends a TCP SYN packet to Server1 10.0.1.2, the controller recognizes it as a new stream and redirects the SYN to server2 10.0.1.3 according to the load balancing strategy. The log output clearly shows that the controller has rewritten the destination IP and destination MAC fields and forwarded the modified message to the corresponding output ports. After detecting SYN, the controller installs two flow items with a 5-second idle timeout: one for the client →server2 (redirect) direction and the other for the server2→ client. The source field is rewritten to maintain the illusion of server1. This confirms that bidirectional stateful redirection works as expected.

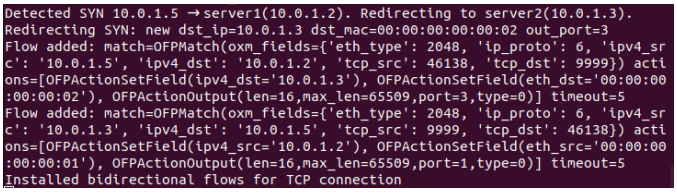


Fig. 10: TCP SYN redirection and flow table installation log

Task 5.2: In the xterm terminal of the client, execute `tcpdump -i client-eth0 -w redirect.pcap` to

specify packet capture at the `eth0` network port of the listening client, and directly write the captured packets to `redirect.pcap`. Wireshark can be used to conveniently analyze data directly. Figure 11 shows that in our verification; 70 data packets were successfully received without packet loss. The analysis of the packet capture results(Fig. 12) will also be reflected in the testing result.

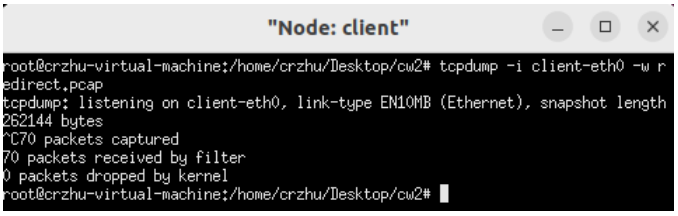


Fig. 11: Packet Capture Results for Redirect Case(tcpdump)

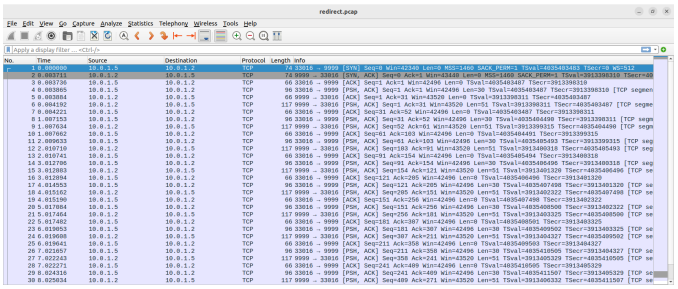
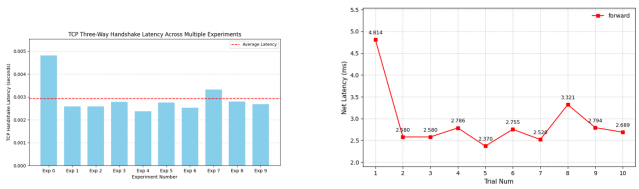


Fig. 12: Packet Capture Results for Redirect Case(wireshark)

C. Testing results

1) Performance analysis of forwarded cases

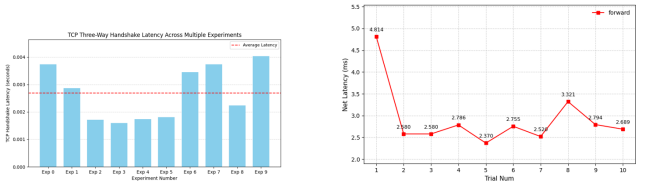
In the forwarding scenario, the experimental results show that the delay of the TCP three-way handshake is relatively consistent, and the calculated average delay is approximately 2.9 ms. As shown in the line graph (Fig. 13a and 14b), the initial test reveals a significant outlier with a peak delay of 4.814 ms, which may be attributed to cold start overhead, such as the initial ARP resolution or rule installation delay. After this initialization stage, the system stabilized significantly, and the subsequent delay (Experiments 2-10) oscillated within a narrow range between 2.370 ms and 3.321 ms. The relatively low variance observed in the steady-state stage indicates that the forwarding mechanism maintains a deterministic processing path.



(a) Forward latency (Bar chart) (b) Forward latency (line chart)

Fig. 13: Comparison of forwarding latency measurements.

Performance analysis of Redirection cases On the contrary, the redirection scenario shows a lower overall average latency (approximately 1). It is 2.7ms, but its characteristic is significantly higher volatility and unpredictability. As shown in Fig. 14a and 14, the delay distribution is non-uniform; Although this mechanism achieved excellent performance in the middle of the experiment, reaching a minimum of 1.596 ms (Experiment 4), it failed to maintain this efficiency. It is worth noting that the subsequent tests (Tests 7-10) demonstrated a significant performance degradation, with the peak delay reaching up to 4.033 ms. This obvious fluctuation indicates that although the redirection method can theoretically outperform forwarding under ideal conditions, in different operational states, it may be vulnerable to instability caused by control plane overhead, resource contention, or delay in flow table updates.



(a) redirect latency (Bar chart) (b) redirect latency (line chart)

Fig. 14: Comparison of redirect latency measurements.

VI. Conclusion

This project successfully implemented and evaluated SDN-based forwarding and redirection using Ryu controllers and the `mininet` topology. By utilizing responsive Packet-In processing and tcp-syn triggered stream installation, correct forwarding to Server1 and redirection to Server2 have been achieved. The packet capture during the test verified that both mechanisms met expectations and achieved a 5-second idle timeout. The delay measurement further indicates that the performance is stable in the forwarding case, while the float is higher in the redirection case due to the additional control plane overhead. Future work can explore optimizing the redirection logic to reduce controller overhead and further stabilize latency under dynamic traffic conditions.

Acknowledgment

All group members contribute the same percentage.

References

[1] J. C. Correa-Chica, J. Cuatindioy-Imbachi and J. F. Botero-Vega, "Security in SDN: A comprehensive survey," *Journal of Network and Computer Applications*, vol. 159, p. 102595, 2020, doi:10.1016/j.jnca.2020.102595.

[2] 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)*, Bangalore, India, 2018, pp. 1-5, doi:10.1109/ICCTAC.2018.8370397.

[3] W. Fan and D. Fernandez, "A novel SDN based stealthy TCP connection handover mechanism for hybrid honeypot systems," *2017 IEEE Conference on Network Softwarization (NetSoft)*, Bologna, Italy, 2017, pp. 1-9, doi:10.1109/NETSOFT.2017.8004194.

[4] T. Boros and I. Kotuliak, "SDN-Based Transparent Redirection for Content Delivery Networks," *2019 42nd International Conference on Telecommunications and Signal Processing (TSP)*, Budapest, Hungary, 2019, pp. 373-377, doi:10.1109/TSP.2019.8769071.