Implementation and Evaluation of Traffic Forwarding and Redirection in an SDN Environment

1st Yize Liu
2254472

2nd Shengtian Huang
2254461

3rd Qing Qin
2254084

4th Xu Chen
2257453

5th Zichen Qiu
2252705

December 13, 2024

***Abstract*—The internet has given rise to the digital society, where nearly everything is interconnected and accessible from anywhere. Despite the widespread adoption of traditional IP networks, their complexity and challenges in management remain significant issues. Software-Defined Networking (SDN), as an emerging paradigm, decouples network control logic from underlying routers and switches, enabling logical centralization of network control and introducing programmability. However, in SDN, when network traffic load increases, node buffers can overflow, leading to packet loss and retransmissions, which affect transmission performance. This remains a major challenge in wireless sensor networks. This paper proposes an SDN-based system for efficiently managing traffic forwarding and redirection between multiple hosts. We conducted network simulations using Mininet and employed the Ryu controller to install dynamic flow table rules. A traffic redirection mechanism was introduced, enabling seamless rerouting of traffic initially destined for Server1 to Server2 when bottlenecks occur. Through packet capture, ICMP ping tests, and TCP flow experiments, we validated the correctness of IP/MAC configurations, the effectiveness of forwarding rules, and the success of traffic redirection. Experimental results demonstrate that, using predefined rules, the SDN controller can achieve effective traffic redirection, ensuring connectivity and stability of communication paths, further verifying the flexibility and operability of SDN-based traffic management.**

I. INTRODUCTION

As Internet usage grows and application scenarios diversify, traditional network architectures struggle to adapt to dynamic resource demands, optimize traffic, and recover quickly from faults. In conventional networks, control logic resides directly in hardware, limiting flexibility and requiring labor-intensive manual configuration [1]. This rigidity is especially problematic during sudden traffic surges or urgent adjustments. Software-Defined Networking (SDN) addresses these challenges by decoupling the control plane from the data plane and enabling centralized, programmable network management [1]. Building on this concept, our project implements a simple SDN topology in Mininet and uses a Ryu controller to achieve dynamic flow management and flexible traffic forwarding. By validating connectivity, dynamically installing and removing flow entries, and accurately redirecting and analyzing traffic, the project demonstrates SDN's potential to optimize resources and adapt policies more efficiently.

There exist numerous challenges in practice. First, the controller may introduce latency when handling new traffic. For example, when a TCP SYN packet initiates a connection, delays in installing flow rules and returning the packet can result in connection interruptions or lag. Additionally, managing the lifecycle of flows presents its own issues. With idle timeouts, the controller may prematurely delete active flow rules due to inaccurate activity tracking or retain inactive rules, leading to resource wastage. Transparent traffic redirection is also prone to errors, especially when adjusting MAC and IP headers; any incorrect modification can disrupt client-server communication or cause packet loss. Finally, accurately measuring performance metrics, such as TCP handshake latency, poses difficulties. Fluctuations in the test environment and the complexity of synchronizing timestamps with tools like Wireshark or Tcpdump make it challenging to ensure stable and precise measurement results.

Beyond validating the feasibility of SDN for dynamic traffic management, this project provides insights into real-world applications. With centralized, programmable control, administrators can rapidly implement load balancing strategies, incorporating multiple servers to enhance resource utilization [1]. They can also swiftly deploy security policies to isolate malicious flows or contain anomalies, thereby improving the overall robustness and security of the network [1]. Dynamic creation, updating, and removal of flow rules serve as a blueprint for more advanced scenarios in data centers and cloud computing environments, where automated operations and resource orchestration are critical.

Our Contributions:

- **Utilized functional programming paradigms to abstract flow management processes in SDN, improving scalability and reducing implementation complexity**
- **Implementing transparent traffic redirection without altering client perceptions**
- **Employing idle timeout mechanisms for precise flow table lifecycle management**

II. RELATED WORKS

In the area of congestion control, Wireless Sensor Networks (WSNs) and Software-Defined Networks (SDNs) are widely used in applications such as environmental monitoring, healthcare, and industrial automation. Addressing network bottlenecks is a key focus, with the Traffic-Redirection Congestion Control and Traffic Protocol (TRCCTP) proposed to dynamically redirect traffic to less congested paths [2]. By leveraging real-time traffic analysis, this protocol reduces packet loss and delays while maintaining data flow integrity, aligning closely with SDN's centralized and dynamic traffic management capabilities [2].

For security in routing protocols, protecting against traffic redirection attacks is critical to maintaining network reliability. Hans et al. proposed techniques such as anomaly detection and traffic analysis to detect and mitigate these threats [3]. Their research highlights the importance of integrating secure routing protocols with dynamic traffic control mechanisms, particularly in SDN environments, where centralized control enhances security and efficiency [3].

III. DESIGN

A. The network system design diagram

The Fig.1 shows a simple Software-Defined Networking (SDN) scenario comprising an SDN Controller, an SDN Switch, a Client, and two Servers (Server1 and Server2). The setup is as follows:

SDN Controller provides centralized decision-making and control over the network. It communicates with the SDN Switch using a southbound interface protocol such as OpenFlow. The controller can dynamically install, modify, or remove flow entries in the switch to determine how packets are routed, thereby enabling features like traffic redirection.

SDN Switch operates in the data plane and forwards packets based on flow rules provided by the controller. It does not make complex routing decisions on its own. When it encounters unknown traffic, it queries the controller for instructions and then applies the received flow entries to guide packet forwarding.

The client, with IP 10.0.1.5/24 and MAC 00:00:00:00:00:03, believes it is sending requests directly to Server1. From the client's perspective, it targets Server1's IP and MAC addresses for any service request. The client is unaware that its traffic might be redirected elsewhere.

Server1 has the IP 10.0.1.2/24 and MAC 00:00:00:00:00:01. Under normal circumstances, the client would directly communicate with Server1 to obtain the desired services.

Server2's IP is 10.0.1.3/24, and its MAC address is 00:00:00:00:00:02. In this redirection scenario, even though the client thinks it is interacting with Server1, the SDN controller and switch can reroute the traffic to Server2 seamlessly. This transparent redirection ensures the client remains unaware of the change while the network administrators can optimize traffic flow, balance loads, or test services by redirecting traffic from the intended Server1 to Server2.

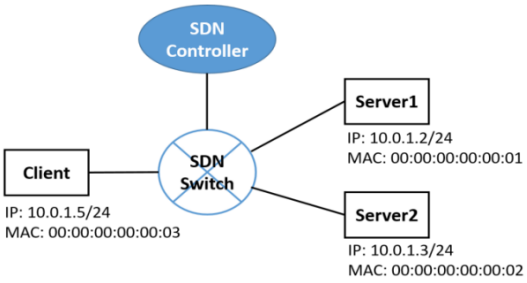


Fig. 1: The network topology

B. Workflow of Program

In the case of forwarding, the system starts with the Controller running and ready, Server1 listening on port 9999, and the Switch connected to the Controller. When the Client sends a TCP SYN packet to Server1, the Switch, having no matching flow entry, sends a PacketIn message to the Controller. The Controller processes the PacketIn, learns the Client's MAC-to-port mapping, and installs a flow entry with a 5-second idle timeout to forward traffic from the Client to Server1. This entry allows the Switch to forward all subsequent packets directly, avoiding further involvement of the Controller. The TCP three-way handshake between the Client and Server1 is completed, and the data exchange proceeds seamlessly. If no traffic occurs within the 5-second timeout period, the flow entry expires, requiring the PacketIn process to restart for future traffic.

In the case of redirecting, when a Client initiates a connection to Server1, the Switch forwards the PacketIn message to the controller because no matching flow entry exists. The controller identifies the client and Server 1 based on their

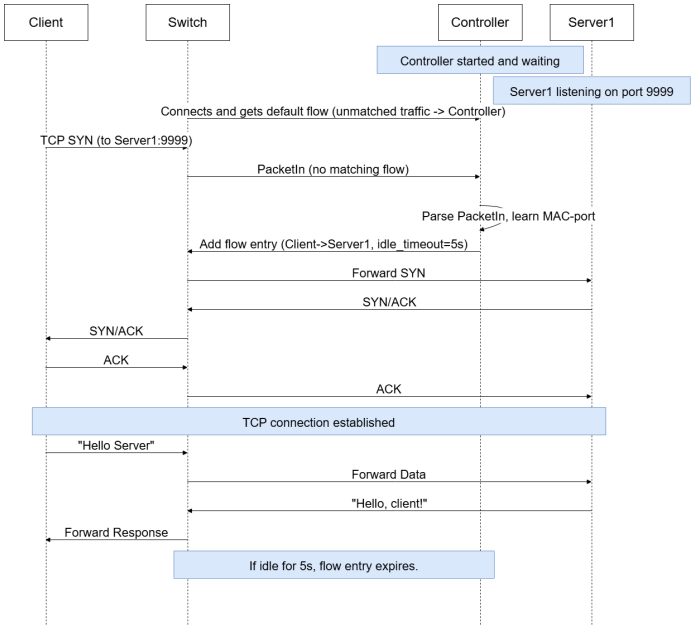


Fig. 2: The sequence diagram of forwarding

MAC/IP addresses, determines that redirection is needed, and modifies the flow entry to forward the packet to Server 2. It updates the destination MAC/IP field to Server 2 while preserving the original server identity for the client. For traffic returning from Server 2 to the client, the controller modifies the source MAC/IP field to impersonate Server 1, maintaining transparency. All subsequent packets follow these rules unless the flow entry expires after 5 seconds of inactivity.

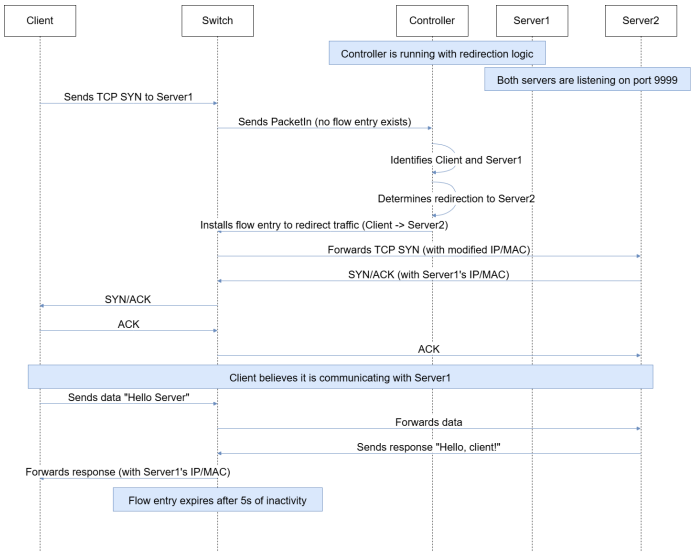


Fig. 3: The sequence diagram of redirecting

C. Algorithms

To provide a clear understanding of the proposed approach, the following section presents pseudocode illustrating the key steps and logic underlying our traffic redirection mechanism.

Algorithm 1: Forward Algorithm

Input: PacketIn event
Output: PacketOut
Extract src_mac, dst_mac ;
if *dst_mac known* **then**
  $out\_port \leftarrow mac\_to\_port[dst\_mac]$
else
  $out\_port \leftarrow FLOOD$
end
if *IP packet* **then**
  $match \leftarrow \{src\_ip, dst\_ip, protocol\}$;
  $actions \leftarrow Output(out\_port)$;
  add_flow($match, actions, idle\_timeout = 5s$);
  send_packet_out($actions$);
else
  if *ARP packet* **then**
    $match \leftarrow \{src\_mac, dst\_mac\}$;
    $actions \leftarrow Output(out\_port)$;
    add_flow($match, actions, idle\_timeout = 5s$);
    send_packet_out($actions$);
  else
    send_packet_out($Output(out\_port)$);
  end
end

Algorithm 2: Redirect Algorithm

Input: PacketIn event
Output: PacketOut
Extract $src\_ip, dst\_ip, src\_mac, dst\_mac$;
if *TCP & Client→Server1* **then**
  $out\_port \leftarrow port(server2)$;
  $actions \leftarrow \{Set(dst = server2), Output(out\_port)\}$;
  $match \leftarrow \{src\_ip, dst\_ip, tcp\_dst\}$;
  add_flow($match, actions, idle\_timeout = 5s$);
  send_packet_out($actions$);
end
else if *TCP & Server2→Client* **then**
  $out\_port \leftarrow port(client)$;
  $actions \leftarrow \{Set(src = server1), Output(out\_port)\}$;
  $match \leftarrow \{src\_ip, dst\_ip, tcp\_dst\}$;
  add_flow($match, actions, idle\_timeout = 5s$);
  send_packet_out($actions$);
end
else
  $match \leftarrow \{src\_ip, dst\_ip, ...\}$;
  $actions \leftarrow Output(out\_port)$;
  add_flow($match, actions, idle\_timeout = 5s$);
  send_packet_out($actions$);
end

IV. IMPLEMENTATION

A. The host environment and the development tools

The implementation was developed and tested in the following host environment:

- **CPU**: Intel Core i9-12900H
- **Memory**: 16 GB
- **Operating System**: Ubuntu 20.04
- **Development Tools**: PyCharm
- **Programming Language**: Python 3.8
- **Python Libraries Used**:
  - `socket`: For network socket communication.
  - `time`: For time-related operations.
  - `mininet`: For creating and simulating a virtual SDN network topology.
  - `ryu`: As the SDN controller framework for handling OpenFlow events.

B. Steps of Implementation

The following steps outline the core phases of the implementation process:

1) **Topology Setup:** Using Mininet to create a simple SDN topology with a single switch, a client host, and two servers.
2) **Controller Initialization:** Launching the Ryu controller application. The controller installs default flow entries to direct unknown flows to the controller.
3) **PacketIn Handling:** Upon receiving a `PacketIn`, the controller parses the packet, learns MAC-to-port mappings, and decides whether to forward traffic directly or apply redirection rules.
4) **Flow Rule Installation:** Based on packet attributes (e.g., IP, TCP), the controller installs flow entries with an idle timeout to avoid repeated PacketIn events.
5) **Redirection:** Modify the destination MAC/IP from Server1 to Server2. For return traffic, modify the source fields to preserve the illusion that Server1 is responding.
6) **Testing and Verification:** Validate the redirection by initiating TCP connections from the client and ensuring data packets are seamlessly rerouted.

C. Programming Skills Used

The implementation leverages the following programming techniques:

- **Object-Oriented Programming (OOP):** Organized the controller logic into classes, encapsulating methods for adding flows, handling events, and parsing packets.
- **Event-Driven Programming:** Ryu uses asynchronous callbacks, triggered by events such as `PacketIn` or switch feature advertisements.
- **Networking Protocol Handling:** Detailed parsing of Ethernet, ARP, IP, and TCP protocols to match fields and set actions.
- **Modular Design:** Separate functions for flow addition (`add_flow`), packet handling, and redirection logic to maintain readability and scalability.

D. Actual Implementation of the Traffic Redirection Function

The redirection function resides within the `PacketIn` handler in the Ryu controller. When a TCP packet from the client to Server1 is detected:

1) Extract `src_ip`, `dst_ip`, and protocol fields.
2) Install a flow entry with these modified fields, directing future packets to Server2 without controller intervention.
3) For return traffic from Server2, similarly modify the source MAC/IP fields to appear as if Server1 is responding.

This ensures the redirection process is seamless and transparent to the client, while minimizing controller overhead through proactive flow rule installation.

E. Difficulties and Solutions

During implementation, several challenges arose:

- **Inconsistent MAC-to-Port Learning**
  Difficulty: Initially, the controller did not maintain consistent MAC-to-port mappings due to rapidly changing network conditions.
  Solution: Introduced a robust MAC learning mechanism and ensured that flow entries with idle timeouts were installed promptly, reducing PacketIn occurrences.
- **Correctly Modifying Packet Fields for Redirection**
  Difficulty: Ensuring that the rewritten MAC and IP fields matched the expected packet structure required careful attention to protocol details.
  Solution: Used Ryu's protocol libraries and verified packet fields with `tcpdump` and logging. This iterative testing confirmed that packets were correctly modified and forwarded.
- **Timeout Management**
  Difficulty: Choosing an appropriate `idle_timeout` for flow entries to minimize controller overhead without allowing stale rules to persist.
  Solution: Experimented with different timeout values and settled on 5 seconds for a balance between dynamic adaptation and efficiency.

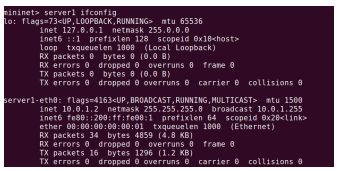
V. Testing and Results

A. Testing Environment

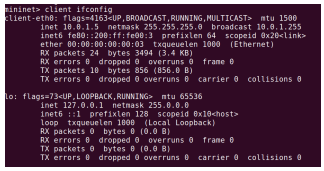
The same test environment as the implementation is used to test our implementation and display the results.

B. Testing Steps

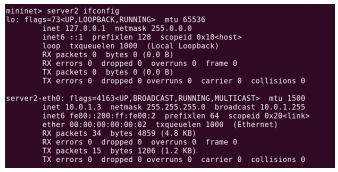
1) Forwarding case: Task 1.1 : We run the networkTopo.py to create the SDN network topology. In the Fig. 4, we have three images each showing the ifconfig outputs of Server1, Client, and Server2. These outputs confirm that each node has been assigned the intended IP and MAC addresses. The figures demonstrate that all three nodes are properly configured within the SDN environment created by networkTopo.py. Thus, we can be confident that the initial network setup is correct and that each machine (Server1, Client, Server2) can potentially communicate as intended.



(a) server 1



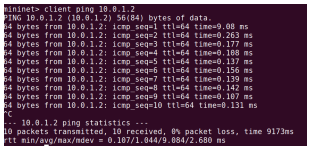
(b) client



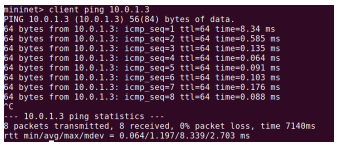
(c) server 2

Fig. 4: IP and MAC Address Configuration

Task 1.2: We run the ryu_forward.py on Controller, and use Client to ping Server1's IP address and Server2's IP address. Fig. 5 illustrates the Client successfully sending ICMP Ping requests to both Server1 and Server2. The ping results show that both servers respond, indicating that the controller's ryu_forward.py application has installed the necessary forwarding rules on the switch. This ensures that the traffic can flow directly between Client and both servers without any packet loss. By confirming connectivity via ping tests, we verify that the forwarding logic on the switch is working correctly and the initial flow rules installed by the controller are effective.



(a) server 1

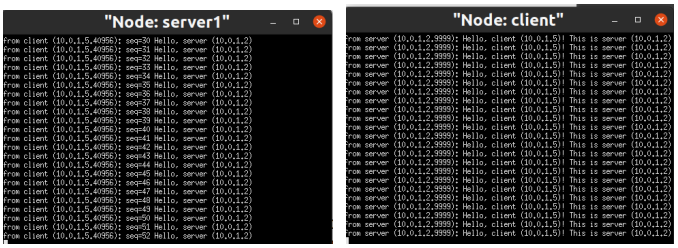


(b) server 2

Fig. 5: Ping Test from Client to Servers

Task 1.3: We run server.py on both Server1 and Server2, and also run client.py on Client after the previous ICMP ping incurred flow entry's idle timeout (i.e., 5 seconds). After the initial ICMP tests, we wait for the flow entry's idle timeout (5 seconds) to expire, ensuring that previously installed flow rules in the switch are cleared. Then we run server.py on both servers to establish TCP listening sockets and client.py on the Client to initiate a new TCP connection. In Fig.6, the flow table (flow entry) screenshot is presented alongside the configurations of Server1 and the Client. The flow table image demonstrates how the controller installs new flow entries when the Client starts sending TCP traffic. This confirms the controller's dynamic behavior: once packets arrive after the idle timeout, the controller's forwarding logic is triggered again, resulting in fresh flow entries that enable transparent end-to-end communication. With server.py running on Server1, the final image (or the verification step) confirms that Server1 is indeed receiving traffic from Client. Since the TCP connection

is established after idle timeout, the presence of these flow entries and the server's correct reception of data illustrate that the forwarding mechanism is operational, even after entries expire and are re-installed.

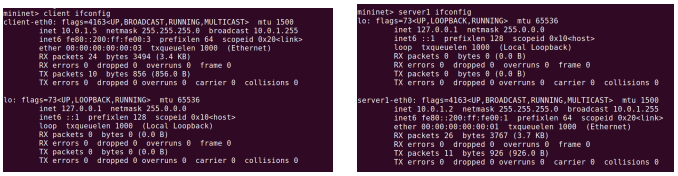


(a) server 1 (b) client

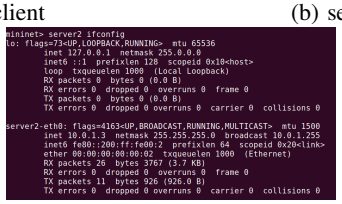
(c) flow entry

Fig. 6: Network Configuration and Flow Table Entry

2) Redirecting case: Task 2.1: We run the networkTopo.py to create the SDN network topology. Check Client, Server1, and Server2 use the correct IP addresses and MAC addresses. In Fig.7, we again see screenshots from ifconfig outputs on Client, Server1, and Server2. These ensure that each node is properly configured with its intended IP and MAC addresses in the newly created SDN topology.



(a) client (b) server 1

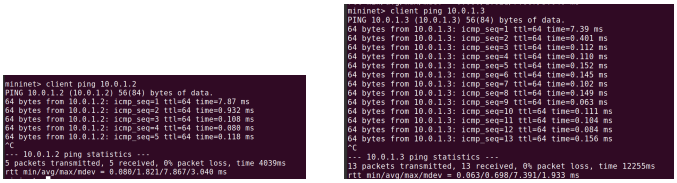


(c) server 2

Fig. 7: IP and MAC Address Configuration

Task 2.2: We run ryu_redirect.py on Controller, and use Client to ping Server1's IP address and Server2's IP address. Fig.8 shows the Client successfully pinging both Server1 and Server2 under the controller's ryu_redirect.py application.

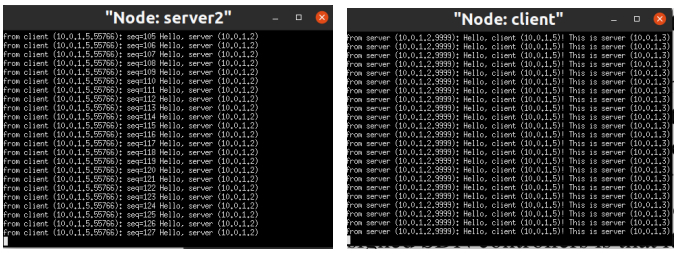
Task 2.3: We run server.py on both Server1 and Server2, and also run client.py on Client after the previous ICMP ping incurred flow entry's idle timeout (i.e., 5 seconds). Similar to the forwarding case, once the previous ICMP-based flow entries expire after the idle timeout, we initiate new traffic patterns to observe how the controller installs redirection rules.



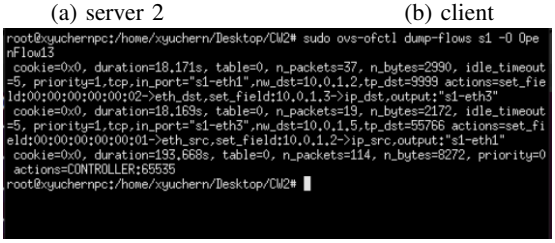
(a) server 1 (b) server 2

Fig. 8: Ping Test from Client to Servers

In Fig. 9, set_field:00:00:00:00:00:02 to eth_dst changes the packet's Ethernet destination address (eth_dst) to 00:00:00:00:00:02, and set_field:10.0.1.3 to ip_dst modifies the packet's IP destination address (ip_dst) to 10.0.1.3. By applying these modifications, traffic originally destined for Server1 (10.0.1.2) is redirected to Server2 (10.0.1.3). Similarly, for the return path, the rules use the same notation to rewrite the source fields: set_field:00:00:00:00:00:01 to eth_src updates the Ethernet source address to Server1's MAC, and set_field:10.0.1.2 to ip_src resets the IP source address to Server1's IP. As a result, when the packet returns to the client, it appears as though it originated directly from Server1, preserving transparency. These arrow-marked field rewrites thus ensure seamless traffic redirection without alerting the client to any changes. By running server.py on Server2 and client.py on the Client, we observe that despite the Client attempting to communicate with Server1's IP address, the resulting output on Server2 confirms that it is receiving the Client's data. This validates that the redirection mechanism is functioning as intended: the Client remains unaware of the redirection, while the controller and switch seamlessly collaborate to deliver the Client's packets to Server2.



(a) server 2 (b) client



(c) flow entry

Fig. 9: Network Configuration and Flow Table Entry

C. Testing Results

As shown in the figures above, all the requirements of this task have been demonstrated. The established SDN network

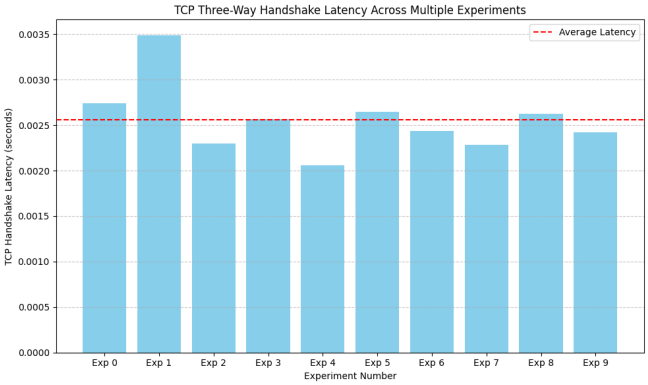
topology consists of one client, two servers, and one controller, and all nodes can communicate with each other normally. In addition, the client and both servers have correct IP and MAC addresses that meet the specified requirements. Flow table entries with a 5-second idle timeout have been correctly configured, and the client.py and server.py programs run properly on the client and the servers. When running the ryu forward.py program on the controller's terminal, communication from the client to Server1 is correctly forwarded. On the other hand, after running the ryu redirect.py program on the controller, the entire flow path from the client to Server1 is altered and redirected to Server2, without the client's knowledge. In other words, the program successfully implements source address spoofing, making the client believe that the packets still originate from Server1.

A comparison was made between the forward and redirect situations, and the Three-way handshake time was tested ten times. By calculating the average, the average forwarding time was 0.00264s and the average redirecting time was 0.00309s. The chart is as follows. This indicates that the average latency of redirection is significantly higher than that of forwarding. In addition, both networks' delay curves display some fluctuations and relatively uniform variance. This result can be explained by the fact that, when handling the TCP protocol, redirection involves additional requirements and operations due to the differentiation of communication endpoints by source and destination IP addresses. Furthermore, to implement redirection, the SDN controller must dynamically modify flow entries, potentially causing delays associated with re-installing these entries. To enhance network latency performance during redirection, effective flow-entry distribution algorithms can be employed to reduce unnecessary messages in the control plane. Moreover, employing shortest-path algorithms such as Dijkstra or Bellman-Ford during redirection can shorten the data packet transmission path, thereby reducing latency.

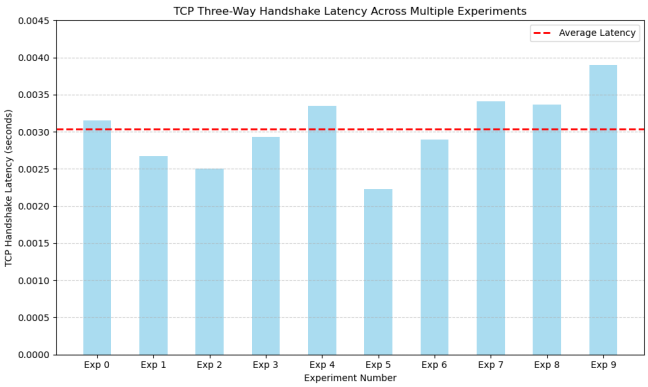
VI. CONCLUSION

In this report, we demonstrated basic SDN functionality using Mininet and Ryu, confirming successful forwarding and seamless traffic redirection. The initial setup allowed a client to reach two servers directly, validated through ICMP and TCP tests. Introducing redirection rules ensured that traffic intended for one server was transparently rerouted to another, showcasing SDN's agility and dynamic adaptability in maintaining stable connectivity.

However, these tests were conducted within a simple topology and focused on basic traffic patterns, limiting their applicability to more complex real-world scenarios. Future work should explore larger networks, more diverse protocols, and advanced features such as security services, QoS policies, load balancing, and resilience under high-load conditions. This broader scope would provide a deeper understanding of SDN's capabilities in robust, scalable, and intelligent traffic management.



(a) Latency for forwarding case



(b) Latency for redirecting case

Fig. 10: Latency Comparison Between Server 1 and Server 2

ACKNOWLEDGMENT

All group members contribute the same percentage.

REFERENCES

[1] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[2] T. Chand, B. Sharma, and M. Kour, "Trcctp: A traffic redirection based congestion control transport protocol for wireless sensor networks," in *2015 IEEE SENSORS*, 2015, pp. 1–4.

[3] K. Hans, L. Ahuja, and S. K. Muttoo, "Significant factors for detecting malicious redirections," in *2017 2nd International Conference on Telecommunication and Networks (TEL-NET)*, 2017, pp. 1–4.