XJTLU: eXploring Joint Traffic and Latency Under SDN

MingXuan Hu 2252534 Rui Sang 2251576 ZiQi Liu 2253863

ZhiXin Li 2254577 ZhengHao Zhou 2257629

***Abstract*—This project explores the innovative use of Software Defined Networking (SDN) to dynamically manage and optimize network traffic within a simulated environment. Leveraging the Mininet emulator and the Ryu controller framework, a simple SDN topology was constructed, and custom controller applications were developed to implement forwarding and redirection mechanisms. The project investigated network latency under different scenarios using Wireshark, analyzing the performance of traffic forwarding and redirection. Results demonstrate the efficiency and adaptability of SDN in managing dynamic traffic paths, with insights into latency variations and practical applications such as load balancing and secure traffic control. The findings highlight the potential of SDN to address complex networking challenges and enhance operational efficiency in real-world scenarios.**

***Index Terms*—Software Defined Networking (SDN), Mininet, Ryu Controller, Network Traffic Management, Latency Measurement.**

I. Introduction

A. Background

Software defined networking, also known as SDN, is an innovative network architecture. It separates the network control function from the forwarding function, achieving programmability of control. This architecture makes the underlying infrastructure transparent and abstract to applications and network services, and the network can be viewed as a logical or virtual entity. This method enables network operators to dynamically adjust traffic and optimize network performance in real-time, thereby addressing the limitations of traditional network models [1]. In this project, SDN is utilized to explore traffic control mechanisms in simulated network environments through the Mininet emulator and Ryu controller framework.

B. Challenge

This project addresses several technical challenges across five key tasks. Task 1 involves constructing a simple SDN network topology using the Mininet Python library, ensuring specific IP and MAC address configurations for the client and servers. Task 2 requires programming an SDN controller application with Ryu to ensure network reachability, implementing a flow entry with a 5-second idle timeout for connectivity verification. Task 3 integrates socket programming by running provided client and server programs within the SDN topology, ensuring proper traffic flow after ICMP-induced flow entries expire. Task 4 focuses on developing an SDN controller application to dynamically create flow entries upon receiving TCP SYN segments, allowing traffic from the client to Server1 and measuring network latency using packet capture tools. Task 5 extends this by redirecting traffic from Server1 to Server2 and measuring the associated latency, further demonstrating the controller's ability to manage dynamic traffic paths.

We sincerely thank all members of the CAN201 faculty team.

C. Practice Relevance

The methods and solutions developed in this project have significant practical relevance in various network applications. One potential application is load balancing, where the dynamic flow control capabilities of SDN can distribute network traffic efficiently across multiple servers, optimizing resource utilization and enhancing performance. Additionally, the project's approach can be applied to secure traffic control, enabling the implementation of advanced security policies that dynamically adapt to network conditions and threats. Another application is in network optimization for data centers, where SDN can streamline operations by automating traffic management and reducing latency [2]. These applications demonstrate the versatility and impact of SDN in addressing complex networking challenges and improving overall network efficiency.

D. Contribution

This project successfully achieved several key objectives through the completion of the defined tasks. Firstly, the SDN network topology was accurately created using the Mininet Python library, with correct IP and MAC address configurations for the Client, Server1, and Server2, as verified in both the Forwarding and Redirection cases. The Ryu-based SDN controller application, *ryu_forward.py*, was effectively implemented, enabling successful ping operations between the Client and both servers, demonstrating network reachability. Additionally, the flow entries were accurately displayed on the switch, and traffic was correctly routed to Server1 following the expiration of ICMP-induced flow entries. In the Redirection case, the *ryu_redirect.py* application successfully facilitated ping operations, and traffic was redirected to Server2, with flow entries correctly reflected on the switch. These contributions illustrate a robust understanding of SDN principles, showcasing the ability to effectively manage and manipulate network traffic within a simulated environment.

II. RELATE WORK

In recent years, the development of SDN has stimulated extensive research on network traffic management and redirection. The work of Kitindi et al. [3] has made significant contributions in this field by introducing the concept of network virtualization, which enables flexible traffic management by decoupling network services from underlying hardware, thereby enabling more efficient use of network resources. Another study by Al-Sadi et al. [4] explored the implementation of traffic redirection strategies in SDN supported wide area networks (WANs). They developed algorithms that utilize the programmability of SDN to reroute traffic based on network conditions, enhancing the resilience and adaptability of WAN. These studies collectively emphasize the importance of SDN in promoting network traffic redirection, supporting the objectives and contributions outlined in our project.

III. DESIGN

A. Architecture of the network system

This network system is based on the SDN architecture and use Mininet to create a star topology, as shown in Figure 1. The system consists of a SDN switch, a controller, and three hosts (Client, Server1, and Server2) connected through the switch. The SDN switch forwards packets from the host to their designated destinations by checking the flow tables and reporting the event to the SDN controllers,which uses protocols like OpenFlow, enabling the server to instruct the switch on where to route packets.

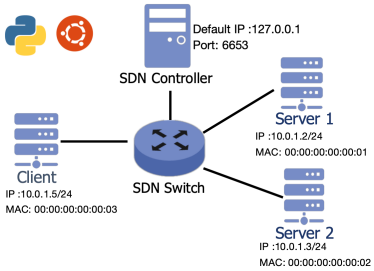

Fig. 1. The network topology

B. Workflow of Program

First, Figures 2 and 3 present the sequence diagrams for the controller under two functions: forwarding and redirecting. Here we both present the **two processes** of sending information before and after the creation of flow entry for comparison.

1) Forwarding function: The forwarding function in `ryu_forward.py` implements efficient packet forwarding by processing packet-in messages, generating flow table rules, installing flow table rules, and sending packet-out messages. When the client sends a packet to the server and the switch does not find a matching stream entry, it generates a packet-in message and sends it to the controller. The controller processes packets, extracts key packet information (such as source IP address, destination IP address, and protocol), and generates

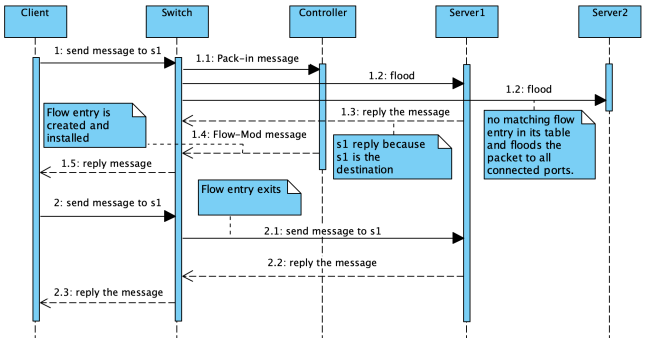

Fig. 2. Sequence diagram of forwarding

matching rules and actions. These rules are sent to the switch via Flow-Mod messages for subsequent packet forwarding. The initial packet is forwarded to the destination port using a packet-out message to avoid redundant processing. We also set `idle_timeout` to prevent flow entries from remaining in the table for too long.

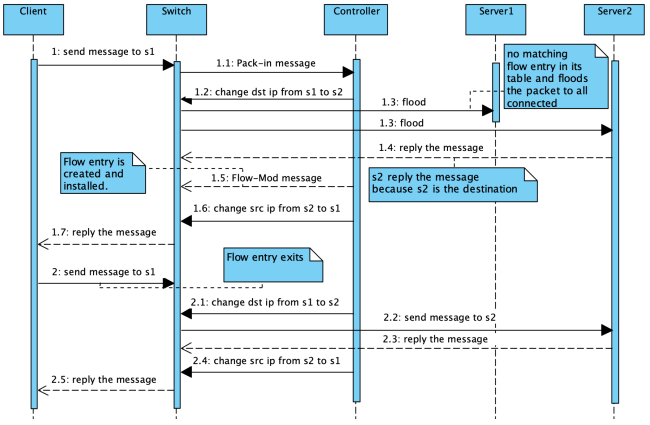

Fig. 3. Sequence diagram of redirecting

2) Redirecting function: The redirection function in `ryu_redirect.py` also extends forwarding by enabling dynamic redirection of destination addresses (for example, from Server1 to Server2). When the switch receives a packet, it sends a packet-in message to the controller. The controller analyzes the packet information and determines whether to redirect the packet based on the source and destination IP addresses. If the target is Server1, it changes the destination to Server2, creates a new matching rule (modifying the target MAC and IP address), and sends it to the switch via a Flow-Mod message. The initial packet is forwarded to Server2 using a packet-out message, and subsequent packets are forwarded directly according to the flow table. If Server2 answers, the controller changes the source address to Server1.

C. Algorithm

Pseudo codes Algorithm 1 and Algorithm 2 respectively represent the main algorithms of the forwarding and redirecting function under the SDN controller.

Algorithm 1 Forward Algorithm

Input: `ev`: Event object containing packet-in data
Output: Check whether a new flow rule should be installed and created

Parse Ethernet packet: `src`, `dst`, `ethertype`
if `dst` exists in `mac_to_port` **then**
    `out_port` ← `mac_to_port[dst]`
else
    `out_port` ← FLOOD
end if
`actions` ← Output to `out_port`
if `protocol == in_proto.IPPROTO_TCP` **then**
    Parse TCP header: `src_port`, `dst_port`
    if `TCP SYN detected` **then**
        Log TCP SYN: "`TCP SYN detected: src` → `dst`"
        `match` ← Match TCP fields (`src_ip`, `dst_ip`, `src_port`, `dst_port`)
        Install flow `add_flow1()`
        Send packet out immediately and **return**
    else
        `match` ← Match TCP fields (`src_ip`, `dst_ip`, `src_port`, `dst_port`)
    end if
end if
if `out_port` ≠ FLOOD **then**
    Install flow with `add_flow1()` ▷ Avoid future packet-in events
end if
`datapath.send_msg()`

Algorithm 2 Redirect Algorithm

Input: `ev`: Event object containing packet-in data
Output: Packets are processed and forwarded with flow entries installed

if `eth.ethertype == ETH_TYPE_IP` **then**
    `ipv4_pkt` ← `pkt.get_protocol(ipv4.ipv4)`
    `ip_src, ip_dst` ← `ipv4_pkt.src, ipv4_pkt.dst`
    if `ip_src == client['ip']` **then**
        if `server2['mac']` in mac_to_port[dpid] **then**
            `out_port` ← `mac_to_port[dpid][server2['mac']]`
        else
            `out_port` ← `OFPP_FLOOD`
        end if
        Install flow for IP source matching client
        Set actions: `eth_dst`, `ipv4_dst`, and `out_port`
    end if
    if `ip_src == server2['ip']` **then**
        if `client['mac']` in mac_to_port[dpid] **then**
            `out_port` ← `mac_to_port[dpid][client['mac']]`
        else
            `out_port` ← `OFPP_FLOOD`
        end if
        Install flow for IP source matching server2
    end if
    Install Flow and Send Packet:
    if `msg.buffer_id != OFP_NO_BUFFER` **then**
        Install flow entry with buffer ID
    else
        Install flow entry without buffer ID
    end if
end if

IV. Implementation

A. Development Environment

The developmemt environment table is shown in table I

TABLE I
Development Environment

Field	Description
Virtual Machine Operating System	Ubuntu 20.04.4
CPU	12th Gen Intel(R) Core(TM) i5-12500H
GPU	NVIDIA GeForce RTX 3050 Ti Laptop GPU
RAM	16G
IDE	JetBrains PyCharm Community 2023.3
Python Version	Python 3.8
SDN controller	Ryu 4.34, Wireshark

B. Implementation Step

1) The project began by thoroughly analyzing the requirements, identifying the need to construct a network topology, implement forwarding and redirection logic using *Ryu*, and evaluate performance metrics.
2) Using Mininet, a simple SDN topology was created with one switch *s1*, a *client*, and two servers *server1* and *server2*. Each device was assigned specific IP and MAC addresses.
3) Draw the sequence diagrams of forward and redirect to outline the interactions between the client, switch, servers, and SDN controllers.
4) Read and understand the simple_switch_13.py, then test the code to ensure bidirectional reachability by handling *Packet_In* events and installing flow entries with a 5-second *idle_timeout*.
5) Test wether the ICMP flows were validated to expire after 5 seconds before TCP communication began.
6) Modify the simple_switch_13.py to detect TCP SYN packets, install flow entries, and forward traffic from the client to server1 dynamically.
7) Wireshark was used to capture packets and measure the latency during the TCP three-way handshake in the forwarding scenario.
8) Write the *Redirect Algorithm* which can redirect client traffic to *server2* by modifying packet headers and dynamically installing flow entries.
9) Use Wireshark to capture packets and measure the

latency during the TCP three-way handshake

C. Programme Flow charts

The flow chart of whole process is shown as follow in Figure 4.

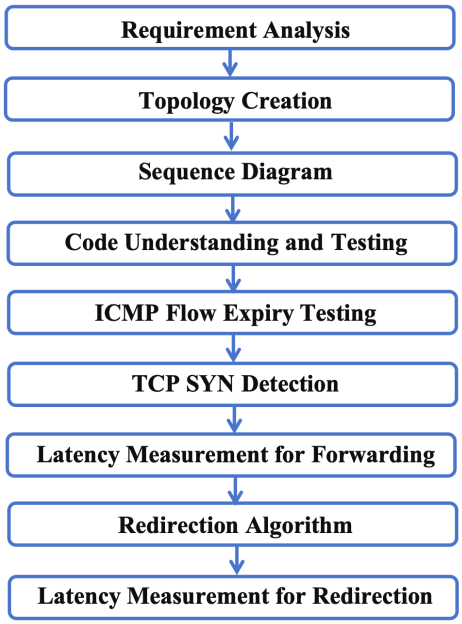

Fig. 4. Flowchart of Implementation

D. Programming Skills

- **Object-Oriented Programming (OOP)** The implementation followed OOP principles to define clear responsibilities for each component, such as topology creation and flow control, improving code modularity and maintainability.
- **Robustness** Various exceptional cases, such as packet truncation and invalid buffer IDs, were considered and addressed to enhance the stability and reliability of the program.
- **Performance Analysis** Applied packet capturing tools like Wireshark to measure and analyze network latency.

E. Actual Implementation

- **Network Topology Construction** The network topology was created using Mininet, including one switch *s1*, a *client*, and two servers *server1* and *server2*. Each device was assigned the specified IP and MAC addresses, ensuring accurate addressing and seamless communication.
- **SDN Controller for Connectivity test** A Ryu-based SDN controller code is obtained through modifying the *simple_switch_13.py* to handle connectivity and manage traffic forwarding. The controller dynamically installed flow entries with a 5-second *idle_timeout* to manage resources efficiently. After confirming connectivity using ICMP ping and ensuring that flow entries expired post-timeout, the *client.py* program was deployed to send TCP traffic to *server1*, where the controller forwarded the traffic as expected. The forwarding functionality was verified by checking flow entries on the switch and observing packet reception on *server1*.
- **Forwarding Algorithm and Latency Measurement** Enhancing the SDN controller by slice modification to detect TCP SYN packets from *client* to *server1* and automatically install followed flow entries for forwarding. This approach reduced flooding by learning and caching MAC addresses during the first interaction. Then we used Wireshark to measure the latency during the TCP three-way handshake. The latency was calculated as the time between the first SYN packet and the final ACK packet.
- **Redirection Algorithm and Latency Measurement** Then we write a new controller to implement redirection, by intercepting packets destined for *server1* and modified their headers to redirect traffic to *server2*. Similarly, packets from *server2* are camouflage to appear as originating from *server1*, ensuring invisibility to the clients. Flow entries were dynamically installed to optimize performance and minimize flooding. Finally we use Wireshark again to measure the latency of the TCP three-way handshake during redirection to analyze performance differences between forward algorithm and redirect algorithm.

F. Difficulties and Solutions

At the beginning, it is quite hard for us to understand the structure and funcitonality of the SDN controller and *simple_switch_13.py*. The complexity of *simple_switch_13.py* code further added to the difficulty. To overcome this, we try our best to dissected the *simple_switch_13.py*, focusing on key functions like handling *Packet_In* events and installing flow entries. This process provided a clear understanding of the controller workflow, enabling us to extend and customize the code. Another difficult is the assignment required the controller to create a flow entry upon detecting the first TCP SYN packet, the key difficulty lay in identifying TCP SYN packets within the *Packet_In* event and designing flow entries to dynamically forward packets without unnecessary flooding. Through careful analysis, we successfully implemented this feature. The controller now detects TCP SYN packets using if funcion, dynamically creates flow entries with *idle_timeout* for efficient resource management.

V. Test and Result

A. Test Environment

The environment of testing is identical to development environment.

B. Test Plan

1) Run the SDN network topology created using the Mininet Python library (`networkTopo.py`) and verify that the network components are created correctly. Confirm that the IP and MAC addresses are properly assigned by checking their respective terminal outputs.

2) Run `ryu_forward.py` and test the connectivity of each node by verifying that they can ping each other. Then, on the switch terminal, run `sudo ovs-ofctl dump-flows s1 -O OpenFlow13` to inspect the flow entry and ensure that an idle timeout of 5 seconds is configured.
3) After completing the above tests, wait 5 seconds to ensure that the flow entries created by ICMP ping packets have been removed. Then, run both `ryu_forward.py` and `networkTopo.py`, followed by executing `server.py` on both `server1` and `server2` terminals, and `client.py` on the client terminal. Verify whether the client can successfully send traffic using sockets to `server1` and receive replies.
4) Before running `client.py`, open a new terminal and execute `sudo tcpdump -i any tcp port 9999 -w handshake.pcap` to capture packet data using Wireshark. Wireshark is used to monitor, capture, and analyze network packets, as well as calculate the network latency of the TCP three-way handshake.
5) Run `ryu_redirect.py` and repeat the tests from steps 3 and 4 to ensure that the traffic sent by the client to `server1` is redirected to `server2`. Capture a new Wireshark file to analyze and calculate network latency under the redirection scenario.
6) Repeat steps 3, 4, and 5 multiple times to calculate network latency under different conditions. Use the collected data to plot graphs and analyze performance metrics.

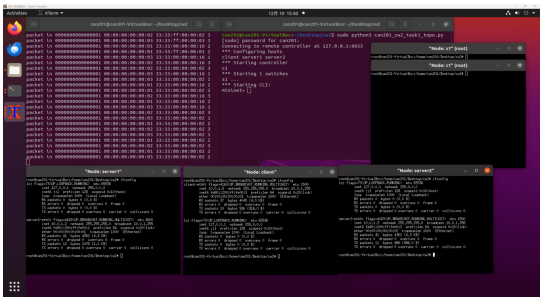

Fig. 5. Test Result of Task 1

C. Test Result

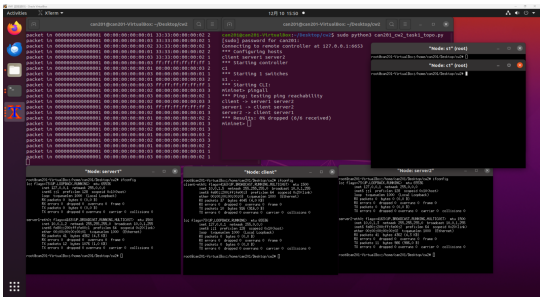

Fig. 6. Test Result of Reachability

From Figures 5 and 6, it is evident that the SDN network topology was successfully established, with IP and MAC addresses properly assigned. All components were able to communicate with one another. The flow entries were configured correctly with appropriate idle timeout settings, ensuring that the given socket-based client and server programs functioned as expected on the corresponding terminals within the network topology.

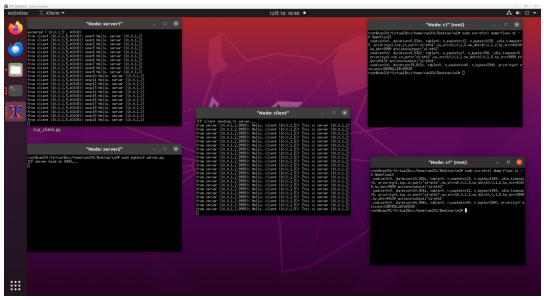

Fig. 7. Test Result of Forwarding

From Figures 7 and 8, when the SDN controller application `ryu_forward.py` was executed, the client's traffic was correctly forwarded to `server1`. Similarly, when `ryu_redirect.py` was executed, the traffic was successfully redirected to `server2`.

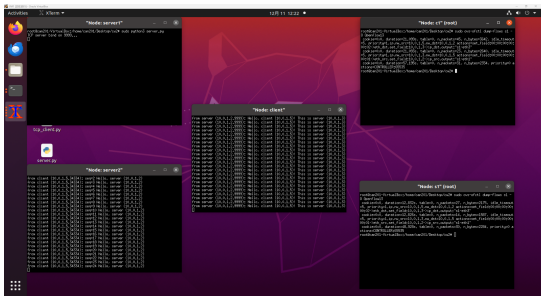

Fig. 8. Test Result of Redirecting

For latency measurement, Wireshark was used to capture TCP SYN and ACK segments during traffic transmission. From the captured packets, it was observed that the network latency was approximately 2.72 ms. As the source and destination of the third handshake matched those of the first handshake, and their network protocol type was IP, the third handshake bypassed the controller and was processed directly by the switch's flow table. Consequently, the time interval between the second and third handshakes was very short, indicating that the primary contributor to network latency was the time gap between the first and second handshakes.

Notably, as Figure 9 shows, the packet capture file revealed that two separate TCP three-way handshakes occurred during redirection. The first handshake established a connection with the original server by sending a request, while the second handshake occurred after redirection, where a new connection request was sent to the redirected server.

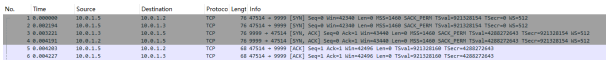

Fig. 9. Packet Capture File of Redirecting

After conducting 15 experimental tests, the latency variations for traffic forwarding and redirection were plotted, as shown in Figure 10. The results indicate that network latency during redirection is significantly higher than that of forwarding. Furthermore, the latency under redirection exhibits greater fluctuations, suggesting that redirection performance is influenced by more factors, leading to higher variability.

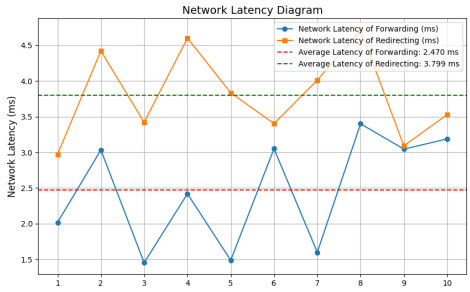

Fig. 10. Network Latency Diagram

The experimental results suggest that the higher latency observed during redirection is due to the additional steps involved compared to forwarding. Specifically, during redirection, the client must send requests to the redirected target address to establish a new connection, which incurs additional time. If there is a large physical distance between two servers, the difference in network latency between forwarding and redirecting will be larger.

VI. Conclusion

This project successfully implemented and evaluated SDN-based traffic management mechanisms in a simulated network environment. By utilizing the Ryu controller framework, forwarding and redirection functions were developed to dynamically manage traffic, achieving efficient resource utilization and reliable network connectivity. Performance analysis revealed that redirection introduces higher latency compared to forwarding due to additional connection establishment steps. These findings underline the versatility of SDN in optimizing network operations, with applications in load balancing, security, and data center management. Future work could focus on enhancing redirection efficiency and exploring its impact on larger, more complex networks.

References

[1] A. A. Okon, K. M. Sallam, Md. Farhad Hossain, N. Jagannath, A. Jamalipour, and K. S. Munasinghe, "Enhancing Multi-Operator Network Handovers With Blockchain-Enabled SDN Architectures," *IEEE Access*, vol. 12, no. 2169–3536, pp. 82848–82866, 2024, doi: 10.1109/access.2024.3411708.

[2] Q. Liao and Z. Wang, "Energy Consumption Optimization Scheme of Cloud Data Center Based on SDN," *Procedia Computer Science*, vol. 131, no. 1877-0509, pp. 1318–1327, 2018, doi: 10.1016/j.procs.2018.04.327.

[3] E. J. Kitindi, S. Fu, Y. Jia, A. Kabir, and Y. Wang, "Wireless Network Virtualization With SDN and C-RAN for 5G Networks: Requirements, Opportunities, and Challenges," *IEEE Access*, vol. 5, no. 2169–3536, pp. 19099–19115, 2017, doi: 10.1109/access.2017.2744672.

[4] Ameer Mosa Al-Sadi, A. Al-Sherbaz, J. Xue, and S. J. Turner, "Routing algorithm optimization for software defined network WAN," *2016 Al-Sadeq International Conference on Multidisciplinary in IT and Communication Science and Applications (AIC-MITCSA)*, pp. 1–6, May 2016, doi: 10.1109/aic-mitcsa.2016.7759945.