Team Members

Name: Qi Cao
Student ID: 2360908

Name: Chengyang Du
Student ID: 2360818

Name: Haoyan Xu
Student ID: 2359977

Name: Jingchen Ding
Student ID: 2253158

Implementation of a SDN Network Topology with Traffic Control Using Mininet and Ryu

Qi Cao (2360908), Chengyang Du (2360818), Haoyan Xu (2359977), Jingchen Ding (2253158)
Information and Computer Science
School of Advanced Technology
Xi'an Jiaotong-Liverpool University
Suzhou, Jiangsu, China

***Abstract*—This project implements a Software-Defined Networking (SDN) system using Mininet and the Ryu controller to develop flow-based mechanisms for both standard traffic forwarding and traffic redirection. We present detailed procedures for the installation and management of flow entries. Furthermore, we evaluate the performance of these mechanisms by measuring TCP three-way handshake latency. Experimental results demonstrate that the redirection scheme incurs a higher initial handshake delay compared to standard forwarding, primarily due to the overhead associated with controller processing and packet header modification. These findings offer valuable insights into the performance implications of dynamic flow control within SDN architectures.**

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

I. Introduction

A. Background

Traditional networks rely on rigid, distributed control that requires inefficient manual reconfiguration [1]. In contrast, Software-Defined Networking (SDN) decouples the control plane from the data plane. This centralization enables programmatic, real-time traffic management and flexible policy deployment via a unified controller [2].

B. Task Specification

Following the concept, this project task is designed to build a SDN traffic control pipeline. The task requires building a specified **SDN network topology** with one client and two servers involved. Meanwhile, two **SDN controller** with **forwarding** flow control and **redirecting** flow control are required to be built and tested. These three files make up the whole pipeline for traffic control.

C. Challenges

Two key technical challenges were addressed in our implementation.

- **Dynamic Flow Management:** The flow entries should be created/expired dynamically based on traffic without degradation from frequent updates.
- **Transparent Redirection:** Rewriting packet headers to mask Server2's identity from the client is prone to error. This required bidirectional flow rules to maintain seamless communication.

D. Practice Relevance

This project's functionalities align with real-world SDN use cases, making it potentially applicable for production scenario. First, our project achieves **fault tolerance**. The controller redirects traffic to backup servers(e.g., Server2) if primary servers fail, minimizing service downtime, this technique can be used in cloud environments like AWS EC2. Second, this project provides **load balancing**, distributing client request across multiple servers to avoid overload. This can be implemented in high-traffic applications like e-commerce platforms. Last but not least, **security**. Our model can be used to isolate suspicious traffic by redirecting it to a monitoring server, protecting critical infrastructure without disrupting legitimate users [3].

E. Contribution

We implement a reproducible mininet topology with hard-coded IP/MAC addresses, ensuring consistent testing across environments. Additionally, two modular Ryu controllers are implemented and tested for forwarding and redirecting traffic control. Finally, we create a standardized latency measurement framework, successfully comparing forwarding/redirection performance and validating SDN overhead for small-scale networks.

II. Related Work

Research in Software-Defined Networking (SDN) has extensively addressed the mechanisms required for dynamic traffic redirection. The foundational architecture introduced by McKeown et al. [2] established the separation of control and data planes, a prerequisite that facilitates programmatic flow management and real-time traffic adjustment. Building on this capability, Wang et al. [4] proposed a transparent redirection scheme for data center load balancing. Their approach utilizes packet header rewriting (modification of IP and MAC addresses) triggered by specific TCP events to divert traffic without client-side reconfiguration—a technique directly relevant to this project's design. Furthermore, Zhang et al. [5] analyzed the performance implications of such interventions, confirming that lightweight controllers introduce negligible latency (1–2ms) during flow installation, thereby validating the feasibility of real-time redirection in latency-sensitive environments.

These works collectively confirm the feasibility of the project's approach: OpenFlow/Ryu for centralized control, header rewriting for transparency, and standard tools for latency analysis.

III. Design

A. Architecture of SDN Network

The system architecture follows a standard Software-Defined Networking (SDN) paradigm, characterized by the decoupling of the control plane from the data plane. As illustrated in Fig. 1, the topology is centralized around a primary controller which uses **OpenFlow** as protocol to communicate with the switch. One client and two servers are connected to the switch with **Client** only being aware of **Server1**.

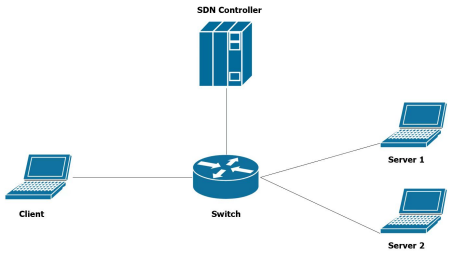

Fig. 1: SDN Topology Design Diagram

The specific components are defined as follows:

- **SDN Controller:** We implemented the controller using the Ryu framework version 4.34. The controller manages all forwarding and redirecting decisions through the **OpenFlow** protocol and handles critical events such as `Packet_In` and `Packet_Out`, acting as the centralized brain of the network. It also implements a MAC learning mechanism to build and create flow table entries dynamically.
- **SDN Switch:** The switch maintains flow tables that contain match-action rules installed by the controller. Each flow entry contains specific matching criteria (e.g. source/destination MAC, IP addresses) and corresponding actions (e.g. forward, modify). When a packet doesn't match any existing flow entry, the switch generates a `Packet_In` message to the controller requesting instructions. An idle timeout of 5 seconds is configured for all flow entries (except the table-miss entry).
- **Hosts: The network includes three hosts (Client, Server1 and Server2) with unique IP and MAC address configurations. Each host runs a Python socket application. The client initiates TCP connections to communicate with the servers, while the servers listen on port 9999.**

B. System Workflow

The system workflow operates on a reactive model, beginning with a default table-miss entry that redirects all unmatched traffic to the controller. Upon receiving a `Packet_In` message, the controller filters LLDP traffic and performs MAC address learning to update the network topology. Once the destination port is determined via the MAC-to-Port mapping (or flooded if unknown), the controller proceeds to **create and install specific flow entries** with a **5-second timeout** to handle subsequent traffic efficiently. This process is protocol-adaptive: for ICMP connectivity tests, flow rules are created matching source/destination IPs and protocols; conversely, for TCP traffic, the controller detects SYN packets to construct fine-grained 5-tuple matches.

As shown in Fig. 2, the system supports two operational modes in the workflow: forwarding and redirection.

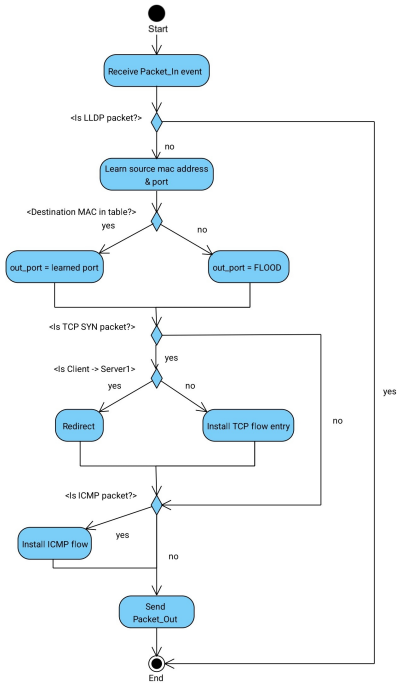

Fig. 2: General WorkFlow Activity Diagram

1) Fowarding Mode: In the basic forwarding scenario, the controller installs standard flow entries that forward packets directly to their intended destination, which in our case is **Server1**.

2) Redirection Mode: For the redirection scenario, when the controller detects a TCP SYN packet from the client destined to Server1, it implements transparent redirection by installing two bidirectional flow entries:

- **Flow 1 (Client → Server2):**Modifies the destination MAC and IP from Server1 to Server2, then forwards to Server2's port.
- **Flow 2 (Server2 → Client):**Modifies the source MAC and IP from Server2 to Server1, making responses appear to come from Server1.

The **complete redirection workflow** is further illustrated in Fig. 3, which shows the sequence of operations performed by the controller and switch. The client sends packets to Server1's

address (10.0.1.2), but the switch intercepts and redirects them to Server2 (10.0.1.3). When Server2 responds, the switch modifies the source address back to Server1's address before forwarding to the client. With this approach, the client believes it is communicating with Server1 while actually connecting to Server2.

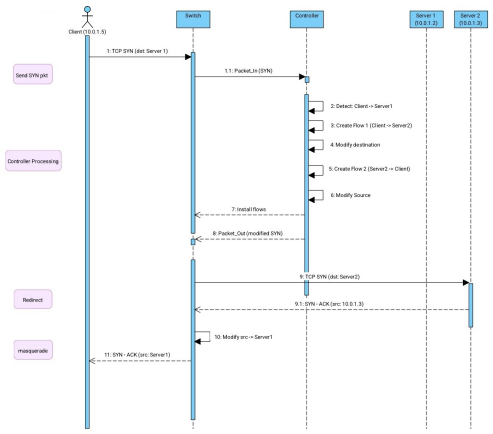

Fig. 3: Redirection Sequence Diagram

C. Algorithm

To better clear the approaches mentioned above, detailed pseudo code of both forwarding algorithm and redirecting algorithm is shown here.

Algorithm 1: Forwarding Algorithm

Input : Packet_In event ev
Extract $pkt, eth, src\_mac, dst\_mac, dpid$;
Extract ip_pkt, tcp_pkt;
if $eth.ethertype == LLDP$ **then**
  return;
$mac\_to\_port[dpid][src\_mac] \leftarrow in\_port$;
if $dst\_mac \in mac\_to\_port[dpid]$ **then**
  $out\_port \leftarrow mac\_to\_port[dpid][dst\_mac]$;
else
  $out\_port \leftarrow FLOOD$;
$actions \leftarrow [output(out\_port)]$;
if $is\_IPv4(pkt)$ ***and*** $is\_TCP(pkt)$ ***and*** $out\_port \neq FLOOD$ **then**
  $ipv4 \leftarrow pkt.get\_protocol(ipv4)$;
  $match \leftarrow \{in\_port, eth\_src, eth\_dst, ipv4\_src, ipv4\_dst\}$;
  $add\_flow(datapath, match, actions, priority = 1)$;
$send\_msg(datapath, actions, ev.data)$;

Algorithm 2: Redirecting Algorithm

Input : Packet_In event ev
Extract $pkt, eth, src\_mac, dst\_mac, dpid$;
Extract ip_pkt, tcp_pkt;
$mac\_to\_port[dpid][src\_mac] \leftarrow in\_port$;
$out\_port \leftarrow FLOOD$;
if ***TCP and Client*** $\rightarrow$ ***Server1*** **then**
  $server2\_port \leftarrow mac\_to\_port[dpid][SERVER2\_MAC]$;
  $match \leftarrow \{in\_port, CLIENT\_IP, SERVER1\_IP, tcp\_ports\}$;
  $actions \leftarrow [set\_eth\_dst(SERVER2\_MAC), set\_ipv4\_dst(SERVER2\_IP), output(server2\_port)]$;
  $add\_flow(match, actions, idle\_timeout = 5)$;
  $send\_packet\_out(actions)$;
else if ***TCP and Server2*** $\rightarrow$ ***Client*** **then**
  $server2\_port \leftarrow mac\_to\_port[dpid][SERVER2\_MAC]$;
  $match \leftarrow \{server2\_port, SERVER2\_IP, CLIENT\_IP, tcp\_ports\}$;
  $actions \leftarrow [set\_eth\_src(SERVER1\_MAC), set\_ipv4\_src(SERVER1\_IP), output(in\_port)]$;
  $add\_flow(match, actions, idle\_timeout = 5)$;
  $send\_packet\_out(actions2)$;
else
  $match \leftarrow \{in\_port, eth\_src, eth\_dst\}$;
  $actions \leftarrow [output(out\_port)]$;
  $add\_flow(match, actions, idle\_timeout = 5)$;
  $send\_packet\_out(actions)$;

IV. Implementation

A. Development Environment

The detailed development environment is listed in Table I.

TABLE I: Development Environment

Component	Specification
Virtual Machine OS	Ubuntu 20.04.4 LTS
CPU	AMD Ryzen 9 7945HX with Radeon Graphics
RAM	4GB
Python Version	Python 3.8.10
SDN Controller	Ryu 4.34

B. Programming Skills

This project utilizes these following programming skills:

- **Object-Oriented Programming (OOP)**: The project implements two controller classes `RyuForward` and `RyuRedirect` that inherit from `app_manager.RyuApp`, encapsulating network state (e.g., `mac_to_port` dictionary) and flow management methods within class instances.

- **Event-Driven Programming**: The Ryu controllers utilize the `@set_ev_cls` decorator to register event handlers for OpenFlow events such as `EventOFPSwitchFeatures`, enabling asynchronous packet processing when switches send packets to the controller.
- **Socket Programming**: The client-server communication is implemented using Python's `socket` module, where `server.py` creates a TCP server listening on port 9999 using `socket.bind()` and `socket.listen()`, while `client.py` establishes connections via `socket.connect()` to trigger TCP three-way handshakes.
- **Modular Design**: The codebase is organized into reusable functions such as `add_flow()` for flow table management and `check_handshake_completion()` for packet analysis, allowing the forwarding and redirecting algorithms to share common infrastructure while maintaining clear separation of concerns.

C. Steps of Implementation

- Carefully read the coursework requirements and make a to-do list, then assign tasks.
- Build a simple SDN network topology using Mininet Python library.
- Develop and deploy the SDN controller application using the Ryu framework, and verify basic connectivity using `ping` tests.
- Implement flow forwarding (Task 2) and flow redirection (Task 3), followed by iterative functional testing.
- Use `tcpdump` on the Client host to capture TCP 3-way handshake packets and calculate the network latency.
- Collect and analyze all test results, and finalize the project report.

D. Actual Implementation

1) Forwarding SDN Controller: When the switch receives a packet, it first checks the flow table to determine whether any flow entry matches the packet header fields. If no matching entry is found and the packet is a TCP SYN segment, the controller installs two flow entries: one for forwarding the connection request from the client to the server1, and another for handling the return traffic from the server1 to the client. Both flow entries are configured with an `idle_timeout` of 5 seconds. Similarly, if the unmatched packet is an ICMP Echo Request, a corresponding flow entry is installed. Otherwise, if a matching flow already exists, the packet is forwarded directly by the switch according to the flow table without invoking the controller.

2) Redirecting SDN Controller: The redirecting controller follows the same initial processing workflow as the forwarding controller. However, when the first TCP SYN segment is sent from the client to Server1, the controller intercepts this packet and rewrites its destination MAC and IP addresses so that it is redirected to Server2. Correspondingly, the controller installs two flow entries: one for forwarding the modified (Client → Server2) traffic, and another for rewriting the return (Server2 → Client) traffic so that it appears to originate from Server1.

V. Testing and Results

A. Testing Steps

1) Task 1: We open a new terminal in the virtual machine and run `sudo python3 networkTopo.py` to establish the simple SDN network topology. Then, on the Client, Server1, and Server2 terminals, we execute the `ifconfig` command to verify their IP and MAC addresses. The results are shown in Fig. 4 below.

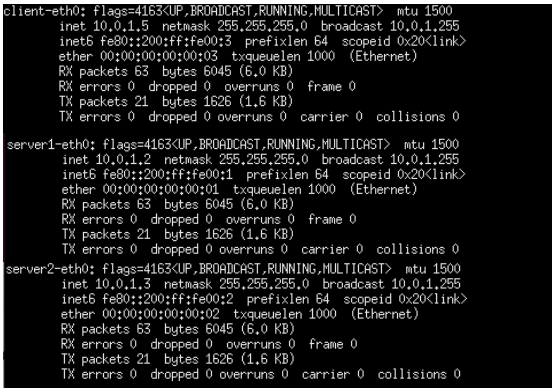

Fig. 4: IP and MAC configuration results on three hosts

2) Task 2: We run the SDN controller application using the Ryu framework in one terminal and execute `sudo python3 networkTopo.py` again in another terminal. Then, in the Mininet CLI, we run `pingall` to test the connectivity. As shown in Fig. 5, all hosts can successfully reach each other. Next, we execute `sh ovs-ofctl dump-flows s1` in Mininet to display the flow table installed on the switch. After waiting for 5 seconds, we check the flow table again. As shown in Fig. 6, since an `idle_timeout=5` is configured, all dynamically installed flow entries expire and are removed automatically.

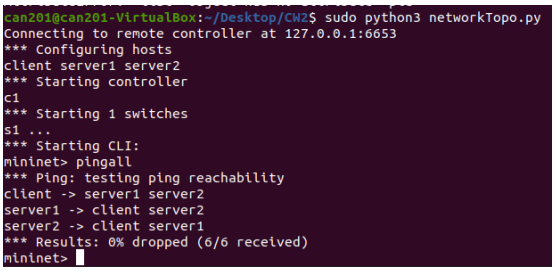

Fig. 5: Pingall results

3) Task 3 & 4.1: We run the SDN controller application `ryu_forward.py` in one terminal, and execute `sudo python3 networkTopo.py` in another terminal to start the SDN network. After confirming basic connectivity using the `pingall` command and waiting for 5 seconds to allow ICMP-related flow entries to expire, we start the server

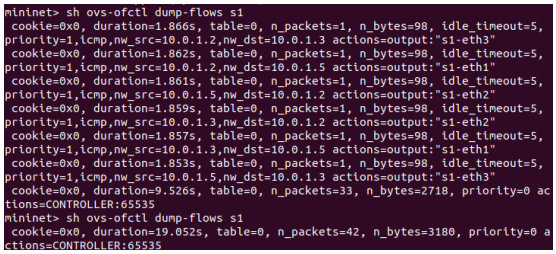

Fig. 6: Flow entries before and after flow expiration

program on both `Server1` and `Server2`, and run the client program on `Client`. The TCP communication between `Client` and `Server1` is successfully established. The flow table entries installed during this TCP session are shown in Fig. 7.

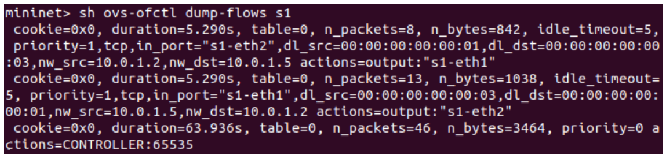

Fig. 7: Flow entries installed during TCP forwarding

4) Task 3 & Task 5.1: This task is similar to Task 3 & 4.1, except that the controller is replaced by `ryu_redirect.py`. When the client initiates a TCP connection to `Server1`, the SDN controller intercepts the first TCP `SYN` packet and dynamically redirects the flow to `Server2` by modifying the destination MAC and IP addresses. The corresponding flow entries installed on the switch are shown in Fig. 8, confirming that the redirection is transparently enforced in the data plane.

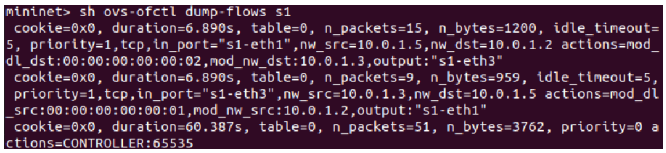

Fig. 8: Flow entries installed during TCP redirection

As a result, the TCP session is successfully established between `Client` and `Server2`, while the client still believes it is communicating with `Server1`. The communication between the client and server2 is shown in in Fig. 9

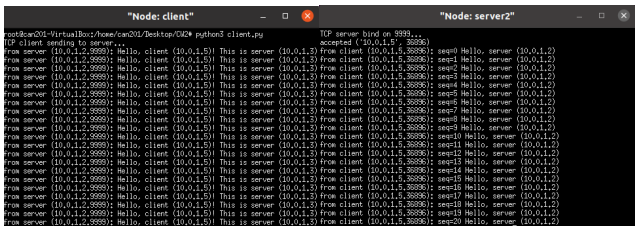

Fig. 9: Communication between Client and Server2

5) Task 4.2 & Task 5.2: We use `tcpdump` on the Client to capture TCP packets and calculate the networking latency, which is measured from the first TCP `SYN` segment sent by the Client to the final `ACK` segment that completes the TCP three-way handshake. Fig. 10 presents the latency measurement in the forwarding scenario, while Fig. 11 illustrates the corresponding results in the redirection scenario. A detailed comparison and analysis of these results will be discussed in Section V-B.

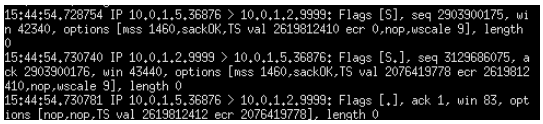

Fig. 10: TCP handshake latency in the forwarding scenario

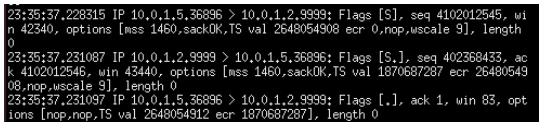

Fig. 11: TCP handshake latency in the redirection scenario

B. Testing Results

Following the successful functional verification of the forwarding and redirection mechanisms, we conducted a performance evaluation specifically focusing on the **three-way handshake latency**. A comparative analysis was performed between the standard forwarding algorithm and the traffic redirection algorithm.

Each algorithm was subjected to ten independent test runs to calculate the average latency. As anticipated, the experimental results yield an average network latency of **2.48 ms** for forwarding and **5.01 ms** for redirection. Fig. 12 and Fig. 13 demonstrate a significantly lower latency for the forwarding mechanism, around a half. Furthermore, the redirecting test exhibits a larger variance, indicating the additional instability during the process of redirecting.

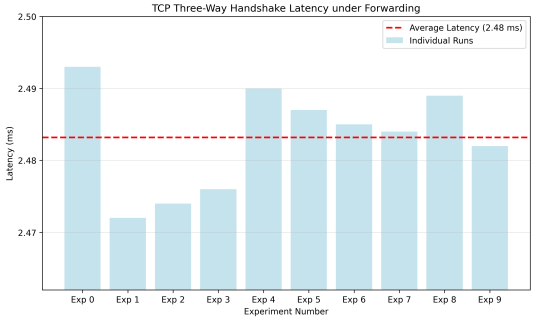

Fig. 12: Network latency in forwarding

The observed latency disparity is intuitively consistent with the operational complexity of the systems. The redirection mechanism incurs greater computational overhead prior to

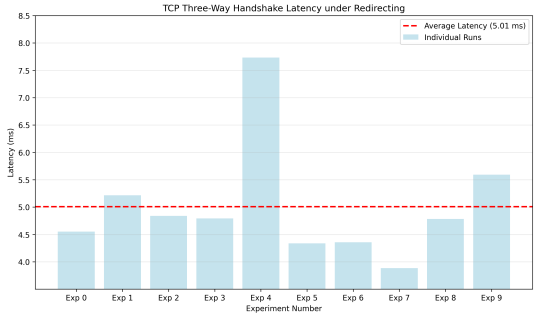

Fig. 13: Network latency in redirecting

connection establishment compared to simple forwarding. Specifically, the redirection process necessitates bidirectional header modifications—altering the source IP for return traffic and the destination IP for outgoing traffic. Consequently, the SDN controller is required to dynamically generate and update complex flow entries, introducing processing delays that increase the overall connection time.

To mitigate this additional latency, particularly in complex network topologies where the redirection overhead may scale, employing efficient **shortest-path algorithms**, such as Dijkstra or Bellman-Ford, could provide substantial performance optimizations by ensuring optimal route selection alongside traffic redirection.

VI. Conclusion

Our project successfully implemented an OpenFlow-based SDN framework utilizing Mininet and Ryu to execute dynamic flow control. Experimental validation confirms the system's capability to perform both standard packet forwarding and transparent traffic redirection. Crucially, our system achieves redirection through dynamic Ethernet and IP header rewriting, ensuring seamless end-to-end TCP connectivity while maintaining low-latency performance.

For future work, we propose three potential improvements:

- **Larger Scale** The current implementation is only validated in a relatively simple network topology, performance on more complex and larger network topology remains unknown. It can be reasonably predicted that the current framework should include more advanced algorithms like Bellman-Ford to handle larger cases.
- **Load Balancing** The system can replace the static redirection logic with round-robin or least-connections schemes that distribute new TCP sessions across any number of back-end servers, improving throughput under high load.
- **Security** Future researchers can try to augment the controller with flow rules that drop packets from blacklisted IPs or unexpected ports, shielding servers from DoS and unauthorized access.

VII. Acknowledgement

All team members contributed equally to the project.

References

[1] D. Kreutz, F. M. Ramos, P. E. Veríssimo, et al., "Software-defined networking: A comprehensive survey," Proceedings of the IEEE, vol. 103, no. 1, pp. 14–76, 2015.

[2] N. McKeown, T. Anderson, H. Balakrishnan, et al., "OpenFlow: Enabling innovation in campus networks," ACM SIGCOMM Computer Communication Review, vol. 38, no. 2, pp. 69–74, 2008.

[3] H. Yang, H. Pan, and L. Ma, "A review on software defined content delivery network: A novel combination of CDN and SDN," IEEE Access, vol. 11, pp. 43822–43843, 2023.

[4] Y. Wang, D. Li, X. Chen, and Z. Zhang, "SDN-based load balancing for data center networks," IEEE Transactions on Network and Service Management, vol. 14, no. 1, pp. 21–34, 2017.

[5] H. Zhang, J. Liu, and K. Yang, "Latency analysis of flow rule installation in software-defined networks," IEEE Access, vol. 7, pp. 123456–123467, 2019.