CAN201 COURSEWORK PART2 REPORT

YIXUAN HOU 2141978
DUAN WANG 2144479
YIFAN WEI 2142111
TAIXU LOU 2142110
MINGYUAN LI 2145618

DECEMBER 16, 2023

***Abstract*—Software-defined networking (SDN) alleviates the constraints imposed by the static architecture of traditional networks by facilitating programmable configurations, which empowers dynamic control over network devices, enhancing overall flexibility and manageability. In coursework part 2, we implement the practical application of SDN concepts, and provide insights into the capabilities and performance metrics of SDN architectures. These are done by creating an SDN network topology and an SDN controller application that is capable of forwarding and redirecting. Moreover, network latency is calculated by capturing the packages using appropriate software. Our contributions involve the establishment of an SDN network structure and the generation of SDN flow entries in different host devices to achieve traffic forwarding or redirection.**

***Index Terms*—SDN, network traffic redirection, network traffic forwarding**

I. Introduction

SDN, or software-defined networking, revolutionizes network management by enabling flexible, programmable configurations. It achieves this by segregating the data plane from the control plane within the network layer, allowing for centralized control and generalized data-plane forwarding. This approach resolves the limitations posed by the static architecture of traditional networks.

The objective of coursework part 2 is to use Mininet to create a simple SDN network topology and manipulate SDN flow entry to control (forward or redirect) the traffic. The tasks involve constructing the SDN network topology, developing an SDN controller application using the Ryu framework capable of both forwarding and redirecting and subsequently capturing packet data through Wireshark or Tcpdump. A pivotal challenge is how to install a flow entry when the packet is under TCP protocol, in other words, identifying the contents needed in the flow entry.

The potential applications with our work include load balance because the redirecting process can distribute network traffic across multiple servers or paths; and secure traffic control, which is provided by the centralized control of SDN which enables the implementation of access control, traffic filtering, and threat detection.

Our contributions cover creating an SDN network topology and creating and installing SDN flow entry under certain circumstances to reach the traffic forwarding or redirection.

II. Related Work

Network traffic redirection is the process of rerouting data packets from one destination to another, which can optimize communication, enhance network security, and facilitate loading balance. It is a crucial technique that helps distribute network load efficiently and enables the implementation of advanced security measures such as intrusion detection and content filtering. There are a number of research studies focusing on facilitating network redirection in different applications.

Zhou et al. proposed an effective request redirection strategy for utilizing the backbone bandwidth to equalize the external network traffic between servers of a video-on-demand cluster during peak workload and presented a family of video replication and placement algorithms [1]. Huang et al. designed and implemented the Mobile Edge Computing (MEC) framework with the OpenAirInterface (OAI), an open-source project of SoftRAN for future 5G networks. It is a threshold-based application-aware traffic redirection mechanism at the edge network, which effectively reduces service latency of user application and network bandwidth consumption by enabling cloud services within the proximity of mobile subscribers [2]. Kassem et al. benchmarked and evaluated redirection tools and introduced a novel data-sharing framework that supports healthcare applications by implementing different proxy tools to accomplish traffic manipulation through containerized Network Functions [3]. The research into network traffic redirection strategies is continuously evolving, as various software applications consistently utilize diverse methods of network traffic redirection to optimize the performance of related applications.

In this coursework, we use Mininet to create an SDN network topology and simulate traffic control functions, including forward and redirect.

III. Design

This section explains in detail the design of the SDN network topology, the traffic forward function, and traffic redirection function.

A. Structure of SDN Network Topology

We built a simple SDN network system that met the following design targets. Our network topology consists of a central controller (c1) and three hosts (Client, Server1, Server2) that are connected through a switch (s1).

Each host is equipped with a unique MAC address and IP address, and the switch is responsible for relaying communication between the different hosts. The controller communicates with the switch via the OpenFlow protocol to enable global control of network traffic. This design allows network administrators to centrally manage and configure network policies and make them adapt to network changes in real time through a centralized control plane. The host's connections to the control plane, data plane, and the information on the host's IP and MAC are detailed in Figure 1.

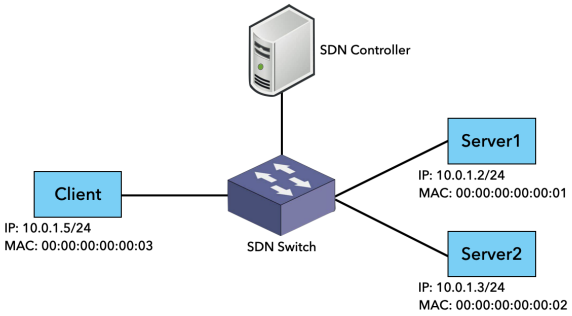

Fig. 1. SDN structure

This subsection explains how our forward function is designed to meet the requirements of task 4. Our forward function is modified based on ryubook 1.0 to achieve the goal

of forwarding packets from the Client to the Server1 [4]. The design can be divided into 4 steps as shown in Figure 2.

B. Forward Function

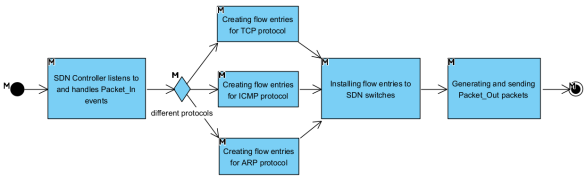

Fig. 2. Workflow of forward

SDN controller listens to and handles Packet_In events: The SDN controller application needs to listen for the Packet_In event from the SDN switch, like the TCP SYN event. Upon receiving such an event, we need to extract relevant information such as source MAC address, destination MAC address, source IP address, destination IP address, and TCP SYN segment.

Creating flow entries: Using the information received, the controller application needs to create a flow entry that defines how to handle the matched traffic to forward to the Server1. When out_port is not a broadcast port, this is the situation that we should consider because the frame has a clear target. The application should process information from different protocols in response to different match information, so we consider several cases: ICMP protocol (ping request), TCP protocol and ARP protocol. Specifically, the match created for TCP and ICMP protocols should have the Ethernet frame type, the input port, the source IPv4 address, the destination IPv4 address, and the IP protocol type. The match created for ARP protocol should have the input port, destination MAC address, and source MAC address. For the add_flow method, we consider that we need table-miss entry to pass the packet to the controller, so we increase the case when the priority is 0.

Installing flow entries to SDN switches: Once the flow entry is created, the controller application needs to install it on the SDN switch so that the switch can process traffic according to the defined rules. We leverage the add_flow function to install the flow entries. In addition, to set an idle timeout of 5 seconds to flow entry, we modified the add_flow function to set the idle_time to 5. Specifically, when constructing the flow entry modification message, the OFPFlowMod function is called with an added parameter idle_time.

Generating and sending Packet_Out packets: Finally, the application needs to construct a Packet_Out packet which contains some necessary information such as buffer ID, input port, output action to a specific port, etc. This packet is to inform the SDN switch to start processing the corresponding traffic according to the new flow entry. Then, the controller application sends it back to the SDN switch.

To illustrate our forward algorithm more visually, we list the pseudo-code to show the forward process specifics, as shown in algorithm 1. Specifically, the input parameters for match and out functions mentioned above are omitted in order to eliminate the repetition of the description. In addition, the ICMP, ARP protocols and bufferId are ignored because they are not modified in our design.

Algorithm 1: packet_in_handler_forward

```
   Data: self, ev
   Result: out
1  actions←[parser.OFPActionOutput(out_port)];
2  if out_port != ofproto.OFPP_FLOOD then
3  |  if eth.ethertype == ether_types.ETH_TYPE_IP then
4  |  |  if ip_protocol == in_proto.IPPROTO_TCP then
5  |  |  |  match←parser.OFPMatch();
6  |  |  end
7  |  end
8  |  self.add_flow(datapath, 1, match, actions);
9  end
10 if msg.buffer_id == ofproto.OFP_NO_BUFFER then
11 |  data←msg.data;
12 end
13 out←parser.OFPPacketOut();
14 datapath.send_msg(out);
```

C. Redirect Function

This subsection explains how our redirect function is designed to meet the requirements of the task 5. And our redirect function is modified based on ryubook 1.0 to achieve the goal of changing the destination Ethernet address from Server1 to Server2, where the Client sends the data and redirecting the source Ethernet address(Server2) sent to the Client to Server1 [4]. The design can be divided into 5 steps as follows and shown in Figure 10.

SDN listens to and handles Packet_In events:

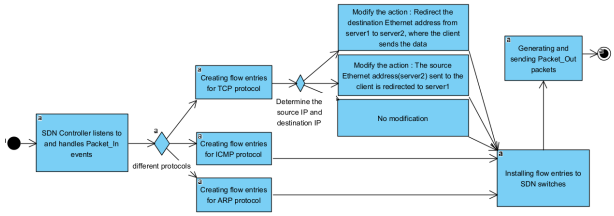

Fig. 3. Workflow of redirect

Similar to the forward function, the Packet_In event from the SDN switch must be listened for by the SDN controller application. Then we can get pertinent data, including the source MAC address, destination MAC address, source IP address, destination IP address, and TCP SYN segment.

Creating flow entries: The controller application should provide a flow entry that specifies how to handle the matched traffic to redirect it to Server 2. When out_port is not a broadcast port, this is the situation that we should consider because the frame has a clear target. We take into consideration several protocol scenarios,

including the ICMP protocol (ping request), TCP protocol, and ARP protocol. The program should process data from different protocols in response to different match information. If the protocol is TCP, there need further steps in the next stage to redirect. If the protocol is ICMP and ARP, the matching setting should be the same as the forward function. For the add_flow method, we consider that we need table-miss entry to pass the packet to the controller, so we increase the case when the priority is 0.

Creating redirection flow entries:

When the IP protocol is TCP, we should redirect traffic that from Client to Server1 is redirected to Server2. We consider three scenarios of traffic from Client to Server1, traffic replying to Client from Server2, and other traffic. For the former situation, we first determine whether the MAC address of Server2 is in the MAC address mapping table maintained by the controller. And if there is a Server2 MAC address, we set out_port to be the output port of Server2, and if it is not, we set it to broadcast to the entire network. Then we set the source and destination IP addresses the match to the corresponding addresses in Packet_In. Next, we modify the action to achieve redirection: change the destination MAC address and the destination IPv4 address to the corresponding address of the Server2, and change the output port to the set out_port.

For the latter situation, we similarly determine whether the MAC address of the Client exists in the MAC address mapping table and similarly set out_port to be the output port of the Client or do the broadcast. After setting the match, we modify the action to achieve redirection: change the source MAC address and the source IPv4 address to the corresponding address of the Server1, and change the output port to the set out_port. For other traffic, we set them as same as the forward function.

Installing flow entries to SDN switches:

The controller must install the flow entry on the SDN switch once it has been constructed for the switch to handle traffic by the established rules. Similarly, we changed the add_flow method to set the idel_time to 5 to provide an idle timeout of 5 seconds for flow input. In particular, an additional parameter idel_time is sent to the OFPFlowMod function.

Generating and sending Packet_Out packets:

Lastly, the program should create a Packet_Out packet with the required data, including the buffer ID, input port, output action to a particular port, and so on. This packet instructs the SDN switch to begin handling the relevant traffic by the newly added flow entry. It is then sent back to the SDN switch by the controller application.

To illustrate our redirect algorithm more visually, we list the pseudo-code to show the redirect process specifics, shown in algorithm 2. Specifically, the input parameters for match and out functions mentioned above are omitted in order to eliminate the repetition of the description. In addition, the ICMP, ARP protocols, bufferId are ignored because they are not modified in our design.

Algorithm 2: packet_in_handler_redirect

```
Data: self, ev
Result: out
actions←[parser.OFPActionOutput(out_port)];
if out_port != ofproto.OFPP_FLOOD then
    if eth.ethertype == ether_types.ETH_TYPE_IP then
        if ip_protocol == in_proto.IPPROTO_TCP then
            if ip_src == client_ip and ip_dst ==
             server1_ip then
                if server2_mac in mac_to_port[dpid]
                 then
                    out_port←[server2_mac];
                else
                    out_port←ofproto.OFPP_FLOOD;
                end
                match←parser.OFPMatch();
                actions←[eth_dst←server2_mac,
                 ipv4_dst←server2_ip, port←out_port];
            end
            else if ip_src == server2_ip and ip_dst ==
             client_ip then
                if client_mac in mac_to_port[dpid]
                 then
                    out_port←[client_mac];
                else
                    out_port←ofproto.OFPP_FLOOD;
                end
                match←parser.OFPMatch();
                actions←[eth_src←server1_mac,
                 ipv4_src←server1_ip, port←out_port];
            end
            else
                match←parser.OFPMatch();
            end
        end
    end
    add_flow(datapath, 1, match, actions);
end
if msg.buffer_id == ofproto.OFP_NO_BUFFER then
    data←msg.data;
end
out←parser.OFPPacketOut();
datapath.send_msg(out);
```

IV. Implementation

A. Host Environment

The implementation of the project was carried out on a Linux virtual machine to create a controlled and isolated networking environment. The Linux distribution used was Ubuntu 20.04, providing a stable and widely supported platform for SDN development.

B. The development software or tools

1) IDE: The primary integrated development environment (IDE) chosen for this project was PyCharm, a powerful Python-centric IDE. PyCharm's advanced features, such as code completion, debugging, and project navigation, facilitated efficient development and debugging of SDN-related Python scripts.

2) Python libraries: Mininet is used to create realistic and scalable network topologies within a single machine, facilitating the emulation of complex network scenarios. Moreover, Ryu is selected as the SDN controller framework, Ryu provides a robust foundation for building SDN applications and controlling network behavior.

3) SDN Controller software: Ryu, a popular and open-source SDN controller, was chosen for its flexibility and extensive set of features. It allowed for the development of custom SDN applications, enabling fine-grained control over network behavior.

C. Further steps of implementation

The current implementation of the network project successfully demonstrates the deployment and manipulation of a SDN using Mininet and Ryu. The following describes the specific steps involved in the implementation process:

1) Network Topology Setup: The implementation begins by executing the Network_Topo.py script within a virtual machine running the Linux operating system. This script leverages Mininet to create a custom network topology comprising two servers, one client, one switch, and an SDN controller.

2) Controller Rule Configuration: Following the network creation, the ryu_forward.py script is run on the SDN controller's terminal. This script, utilizing the Ryu library, establishes communication rules between the Client and Server1 while specifying relevant flow table entries. These rules are instrumental in governing the traffic flow within the network.

3) Server and Client Communication Establishment: server.py and client.py scripts are executed on Server1, Server2, and the Client, enabling TCP communication based on the rules specified in ryu_forward.py. This step realizes communication between the Client and Server1 as directed by the SDN controller.

4) Traffic Redirection: The subsequent phase involves running the ryu_redirect.py script on the SDN controller's terminal. This script, also utilizing the Ryu library, redefines the communication path between the Client and servers. Notably, it manipulates the communication direction, making the Client believe it is communicating with Server1, while in reality, it is connected to Server2.

5) Altered Communication Flow Verification: server.py and client.py scripts are rerun on Server1, Server2, and the Client. The impact of the ryu_redirect.py script becomes evident, as the Client, under the influence of redirection, now receives messages from Server2 when sending messages intended for Server1. Similarly, when Server2 sends messages to the Client, the Client erroneously associates them with Server1.

D. Programming skills

1) OOP: The implementation of the SDN project heavily leveraged Object-Oriented Programming (OOP) principles to enhance code organization, maintainability, and reusability. Key aspects of OOP were applied in the development of custom SDN applications within the Ryu framework.

2) Parallel: Parallel programming techniques were incorporated to optimize the handling of concurrent events within the SDN environment. As SDN involves the simultaneous processing of multiple network events, parallel programming skills were crucial for efficient and responsive controller behavior.

E. Actual implementation of the traffic redirection function

The implementation of the traffic redirection function within the SDN controller is designed to dynamically alter the communication path between network entities. This functionality is achieved through the following steps:

1) Traffic Inspection and IP Protocol Analysis: The system monitors incoming network traffic, specifically focusing on IP protocols. Upon identifying an IP packet, the controller extracts relevant information, including source and destination IP addresses, and the IP protocol.

2) ICMP and TCP Protocol Handling: In the case of ICMP and TCP protocols, the controller evaluates the packet's content to discern the nature of communication. For TCP traffic initiated by the Client towards Server1, the controller intervenes to redirect the communication towards Server2. Conversely, for traffic originating from Server2 destined for the Client, the controller manipulates the communication to make it appear as if it is coming from Server1.

3) Rule Generation for OpenFlow Switch: The controller dynamically generates OpenFlow rules to be applied to the relevant switch based on the observed traffic patterns. These rules specify the conditions under which redirection occurs. For instance, when the Client communicates with Server1, the rule is crafted to alter the destination MAC address to that of Server2, ensuring the redirection of traffic.

4) Actions for Traffic Redirection: To enforce the redirection, the controller defines a set of actions to be executed by the OpenFlow switch. These actions include modifying the destination MAC and IP addresses and directing the packet towards the appropriate output port, thereby rerouting the traffic according to the redirection strategy.

5) ARP Protocol Handling: The implementation also considers ARP protocols, allowing the controller to appropriately handle Address Resolution Protocol requests and responses. This ensures the consistency of MAC address associations within the network during the redirection process.

F. Difficulties

1) Difficulty1: ARP Protocol Handling in ryu_redirect.py: The necessity of checking the Address Resolution Protocol (ARP) in the ryu_redirect.py script arises from potential errors that may occur if ARP protocols are not appropriately addressed. Failure to handle ARP requests and responses can

lead to inconsistencies in MAC address associations, resulting in operational issues. In order to solve this difficuty, the implementation includes specific handling for ARP protocols within the ryu_redirect.py script. By incorporating ARP-related conditions and actions, the script ensures that ARP messages are processed correctly, maintaining the integrity of MAC address mappings during traffic redirection. This approach addresses potential errors and contributes to the overall stability of the redirection functionality.

2) Difficulty2: Redirecting Traffic from Server2 to Client Issue: Redirecting traffic from Server2 to the Client introduces the challenge of altering the communication path while maintaining consistency in address mappings. This requires careful consideration of source and destination addresses to achieve a seamless redirection process. The solution to redirecting traffic from Server2 to the Client involves creating specific conditions and actions in the ryu_redirect.py script. When traffic is initiated from Server2 to the Client, the script dynamically generates OpenFlow rules that modify the source MAC and IP addresses to emulate communication from Server1. Additionally, the script specifies the appropriate output port to direct the traffic towards the Client. By implementing these conditions and actions, the redirection process is effectively facilitated, allowing traffic from Server2 to reach the Client while adhering to the predefined redirection strategy.

V. Testing and Results

We used a series of tests to evaluate the performance of the program. Below is the test environment, the design and implementation of the tests, and the test results.

A. Testing Environment

To ensure the reliability of the test data, all tests in this project will be run on three separate physical machines. The following table records the similarities and differences between the two machines in terms of software and hardware.

TABLE I
Testing Environment

ID	CPU	OS	Linux Kernel
1	Intel i5-9300H	Ubuntu Linux x86_64	5.15.0-89-generic
2	Intel i7-1165G7	Ubuntu Linux x86_64	5.15.0-89-generic
3	Apple Silicon M1	Ubuntu Linux x86_64	5.15.0-89-generic

B. Test Step

This study employs Wireshark as a sophisticated testing tool to meticulously evaluate network traffic, scrutinize network protocols, and precisely calculate network latency within SDN environments. The comprehensive test phase is outlined below. **Network Topology Establishment:** The initial step involves the meticulous creation of an SDN network topology incorporating a singular client, controller, switch, and two servers. Subsequently, a meticulous examination of the IP addresses and MAC addresses of both the Client and servers is conducted to ascertain the integrity of the network components. Wireshark is then initialized to facilitate comprehensive data collection. The following picture shows the IP address and MAC address of the Client, Server1 and Server2.

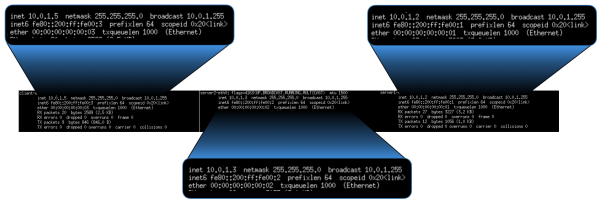

Fig. 4. IP and MAC

Data Recording and Execution: In the subsequent step, Wireshark is enabled to initiate data recording. During this phase, no data is expected to be collected. The forward file is executed on the controller, followed by the execution of server code for both Server1 and Server2. A brief pause of 5 seconds ensues before running the client code on the Client side. The data collection on Wireshark is subsequently halted. To simulate redirect scenarios, the redirect file is executed with identical actions.

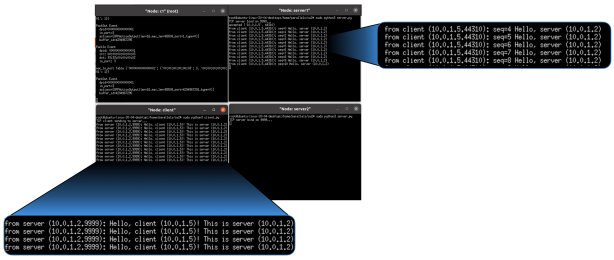

Fig. 5. Forwarding Test

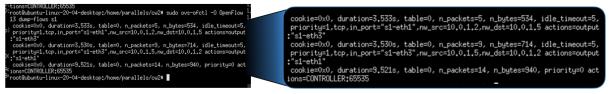

Fig. 6. Flow Table of Forwarding Case

Fig. 7. Redirection Test

Wireshark Data Collection: The last step involves the meticulous collection of data from Wireshark. This includes the examination of elapsed time since the initial frame's arrival within each packet's TCP stream. The results are then presented as a new column in Wireshark. Additionally, iRTT

Fig. 8. Flow Table of Redirection Case

values, which are the networking latency, are scrutinized in [SEQ/ACK analysis]. A visual representation of iRTT and timestamps is provided for clarity and further analysis.

Python Script Development: The final step necessitates the creation of Python scripts to process and compare the data collected through Wireshark. These scripts are designed to analyze latency metrics, evaluate network performance, and facilitate a comparative assessment of the observed data, providing a comprehensive understanding of the SDN environment's behavior.

C. Test Result

Forward Test: The forward testing results reveal the network latency of the SDN process, measured ten times across three distinct computer hosts. Notably, host 1 consistently exhibited the highest network latency among the three hosts, with the 10th test recording the peak latency. Tests 7 and 5 exhibited closely comparable network latency, while the initial three tests demonstrated similar latency values. Examining the sixth test, the disparity in network latency between host 1 and hosts 2 and 3 was minimal. Similarly, the 9th test showed the smallest difference in network latency between host 1 and host 2. Throughout the nearly 10 tests in the forward test, hosts 2 and 3 consistently demonstrated nearly identical results, with the most significant gap observed in the eighth test. Notably, the third, sixth, and seventh tests produced nearly identical results for hosts 2 and 3. We posit that the discernible variations in test outcomes among the three hosts stem from disparities in their respective processors.

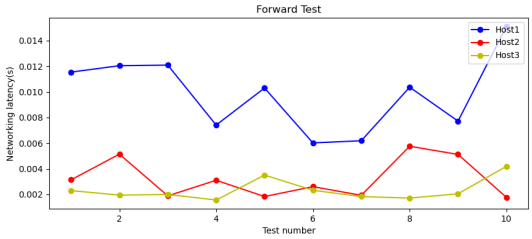

Fig. 9. Forward Result

Redirect Test: The networking latency of the SDN redirection process is tested in three hosts by 10 times for each. According to the line graph, the networking latency of the three hosts fluctuates between 0 and about 0.02s, with the host 2 hovering over the other two lines and the host 3 retaining the lowest at most times. For host 1, in most cases the result fluctuates apparently, between about 0s and 0.0075s. The peak value is at the 8th test, which is about 0.0160s. The best performance occurs at the 6th test, with a value of nearly 0.0001s. Regarding host 2, the result fluctuates in the widest range, between about 0.0050s and 0.0180s. The highest value appears at the 7th test, about 0.0185s; the lowest value is the first test, almost 0.0050s. The value of host 3 changes in a small range, with an average value of about 0.0045s, except a peak value at the 6th test, which is more than 0.0075s and is about twice the value in other tests.

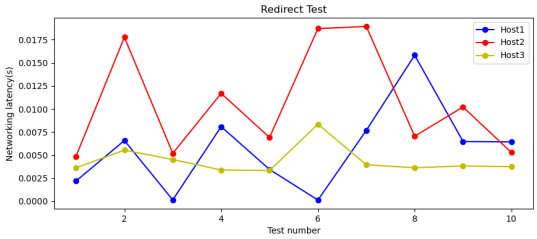

Fig. 10. Redirect Result

TABLE II
TESTING RESULTS

Host	Forward Average Time(s)	Redirect Average Time(s)
1	0.009833276	0.00535858542
2	0.002342353	0.00439141090
3	0.003228751	0.01066300370

VI. CONCLUSION

In coursework part 2, the completion of the five tasks has provided a comprehensive understanding of SDN principles and their practical implementations. We program SDN network topology and SDN controller applications using Mininet Python library and Ryu framework respectively, and successfully integrate socket client and socket server programs into the SDN network. SDN controller applications, including forward and redirect, are utilized to handle TCP SYN segments and manipulate traffic flow between hosts. Additionally, we also measure networking latency during the TCP 3-way handshake by using Wireshark.

Further exploration is needed on how to enhance network security through SDN, encompassing the development of intrusion detection, traffic monitoring, and real-time response mechanisms. Moreover, it is necessary to explore more flexible approaches on dynamic network management to accommodate the ever-changing network conditions and requirements.

REFERENCES

[1] X. Zhou and C.-Z. Xu, "Request redirection and data layout for network traffic balancing in cluster-based video-on-demand servers," in *Parallel and Distributed Processing Symposium, International*, vol. 3. IEEE Computer Society, 2002, pp. 0127–0127.

[2] S.-C. Huang, Y.-C. Luo, B.-L. Chen, Y.-C. Chung, and J. Chou, "Application-aware traffic redirection: A mobile edge computing implementation toward future 5g networks," in *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2)*. IEEE, 2017, pp. 17–23.

[3] J. A. Kassem, O. Valkering, A. Belloum, and P. Grosso, "Epi framework: Approach for traffic redirection through containerised network functions," in *2021 IEEE 17th International Conference on eScience (eScience)*. IEEE, 2021, pp. 80–89.

[4] OSRG. (Accessed: Dec. 15, 2023) Ryu book - chapter 3: Switching hub. [Online]. Available: https://osrg.github.io/ryu-book/en/html/switching_hub.html