

An SDN-based network system for TCP Traffic Forwarding and Redirection: Design and Evaluation

Pengkai Chen 2251486
School of Advanced Technology
Xi'an Jiaotong-Liverpool University
SuZhou, China
Pengkai.Chen22@student.xjtlu.edu.cn

Yilin Li 2255705
School of Advanced Technology
Xi'an Jiaotong-Liverpool University
SuZhou, China
Yilin.Li2202@student.xjtlu.edu.cn

Peiling Tu 2251487
School of Advanced Technology
Xi'an Jiaotong-Liverpool University
SuZhou, China
Peiling.Tu22@student.xjtlu.edu.cn

Jinhong Jiang 2251601
School of Advanced Technology
Xi'an Jiaotong-Liverpool University
SuZhou, China
Jinhong.Jiang22@student.xjtlu.edu.cn

Abstract—This project focuses on developing a network system based on the Software-Defined Networking (SDN) architecture. The Mininet simulation tool is utilized to construct the network topology while the SDN controller algorithm is designed based on the Ryu framework. The TCP traffic forward & redirect functions are developed, successfully embedded into the controller algorithm, and tested for performance evaluation. Ultimately, the network latency is calculated based on the test result.

Keywords—SDN, Mininet, Ryu, TCP traffic, forward, redirect

I. INTRODUCTION

The rapid development of information technology has made the Internet essential to modern society. Traditional networks are fixed and can not handle the increasing number of network devices and the expanding scale of network systems. The SDN addresses these issues by separating the control plane from the data plane, enabling centralized management, and simplifying network infrastructure maintenance [1].

This project aims to develop an SDN-based network system, which can be specifically divided as follows.

- Use Mininet to construct an SDN network topology based on the task sheet requirement.
- Design two SDN controller algorithms based on the Ryu framework: the first algorithm should be capable of forwarding TCP traffic to the server 1 host while the second algorithm should be designed to redirect TCP traffic to the server 2 host.
- Run Wireshark/Tcpdump on the client host to capture the packet for calculating the network latency of the TCP three-way handshake.

This project has potential applications in various domains. The SDN's redirection feature could distribute traffic across servers to reduce congestion and mitigate attacks by redirecting problematic traffic.

This project designs an SDN-based network system. The network topology and SDN controller algorithm are developed using the Mininet and Ryu frameworks respectively. A TCP traffic forwarding function is designed, deployed, and evaluated through packet capture. Based on the testing result, the network latency is calculated.

II. RELATED WORK

A. Mininet

The Mininet is a network emulator designed for research and education in SDN and network virtualization, emulating network environments using virtual hosts, switches, and controllers [2]. It supports SDN experiments with Linux namespaces and Open vSwitch, allowing dynamic traffic management and testing of custom network policies, traffic redirection, and control applications.

B. Ryu framework

Ryu is a lightweight SDN framework supporting OpenFlow and other protocols for centralized network control. It enables dynamic flow adjustments for tasks like congestion mitigation, load balancing, and DDoS defense [3], offering flexibility and programmability for SDN research and implementation.

C. SDN

The SDN revolutionizes network management by decoupling data and control planes, enabling centralized control, programmability, and dynamic traffic redirection [4]. This approach enhances flexibility and efficiency by optimizing traffic forwarding and control plane resource management [5].

D. OpenFlow

Introduced by the Open Networking Foundation in 2008, the OpenFlow technology enables flexible network traffic redirection [6]. By separating the control plane from the data plane, OpenFlow allows centralized controllers to dynamically manage flow tables in switches, enhancing network programmability and adaptability [4, 7, 8].

III. DESIGN

A. Network System Design

The basic architecture of the network system is demonstrated in Figure 1, comprising two network planes.

1) Data plane

The data plane will handle the actual forwarding and processing of data packets. Several network components are added based on the given network topology diagram in the CW2 task sheet, which are the client host, server 1&2 host,

switch, and SDN controller. The Mininet framework is employed to implement the network topology at the code level.

2) Control plane

The control plane plays a crucial role in managing the logical decision-making and configuration of network devices. In this network system, the SDN controller algorithm, built upon the Ryu framework, is designed to provide packet forwarding guidance and manage flow table entries on the switch.

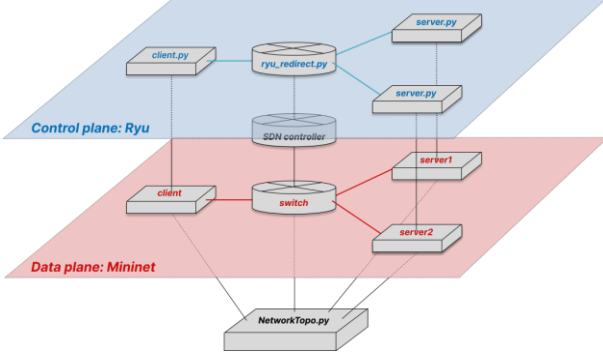


Fig. 1. The architecture of the network system

B. Programming Work Flow

1) Initialization of the Mininet network topology

The general workflow of constructing the Mininet network topology is manifested in Figure 2.

- Initially, an empty network topology will be initialized after the code execution.
- Subsequently, various network components will be integrated into the topology. The network properties will be specifically configured to meet the task requirements.
- After the configuration, the network topology will be automatically built and started.

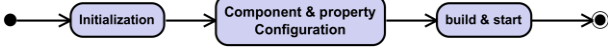


Fig. 2. The workflow of initializing the network topology

2) Working mechanism of the SDN controller algorithm

The working mechanism of the Ryu SDN controller algorithm is demonstrated in Figure 3.

- Firstly, a default flow table entry will be configured for sending all unmatched (have no matching flow table entry) packets to the controller.
- Once a packet is sent to the controller, it will be unpacked. The packet information will be retrieved for verification.
- Based on the packet information, the extracted Mac address and physical port data of the source host will be learned by the switch.
- Ultimately, a flow table entry will be created based on the packet information and configured on the switch if the output port is not defined as **Flood**.

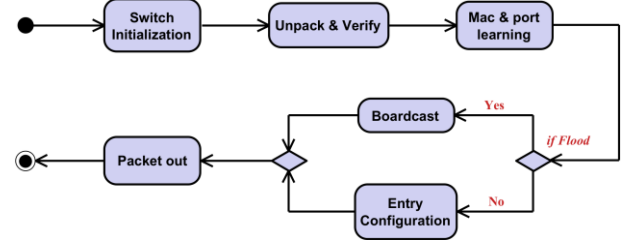


Fig. 3. The workflow of the SDN controller algorithm

C. Kernel Pseudo Code

The kernel pseudo code of the network traffic forward & redirect function is shown in Figure 4.

```

Algorithm 1: Packet In Handler Function Algorithm
Input: self, ev
1 Function: packet_in_handler
2 Extract Packet Data Using ev
3 if output port is not defined as Flood then
4   if packet is ARP type then
5     ARP Packet Handling
6   end
7 else if packet is IP type then
8   Retrieve IP Packet Data
9   if packet is ICMP type then
10    ICMP Packet Handling
11  end
12  if packet is TCP type then
13    Retrieve TCP Packet Data
14    if IP destination is server.1.IP and MAC destination is server.1.Mac then
15      match ← parser.OFPMatch(eth_type = ether.types.ETH.TYPE_IP,
16                               ip_proto = ip.protocol,
17                               ip_v4_src = client.IP,
18                               ip_v4_dst = server.1.IP,
19                               tcp_src = tcp.src.port,
20                               tcp_dst = tcp.dst.port)
21      out_port ← self.mac.to_port[dpid][server.1.mac]
22      action ← [parser.OFPActionSetField(eth_dst = server.2.mac),
23               parser.OFPActionSetField(ipv4_dst = server.1.IP),
24               parser.OFPActionOutput(port = out_port)]
25    end
26  else if IP destination is client.IP and MAC destination is client.Mac then
27    match ← parser.OFPMatch(eth_type = ether.types.ETH.TYPE_IP,
28                             ip_proto = ip.protocol,
29                             ip_v4_src = server.2.IP,
30                             ip_v4_dst = client.IP,
31                             tcp_src = tcp.src.port,
32                             tcp_dst = tcp.dst.port)
33    out_port ← self.mac.to_port[dpid][client.mac]
34    action ← [parser.OFPActionSetField(eth_src = server.1.mac),
35             parser.OFPActionSetField(ipv4_src = server.1.IP),
36             parser.OFPActionOutput(port = out_port)]
37  end
38  end
39  end
40 end
41 Configure Flow Table Entry Using match And action
42 Form Output Packet And Send It To Switch For Transmission
43 end function

```

Fig. 4. The kernel pseudo code of the network traffic redirect algorithm

IV. IMPLEMENTATION

A. Host Environment

The Linux virtual machine powered by the VirtualBox is used for software development and implementation. The detailed information is illustrated in Table 1.

TABLE I. HOST ENVIRONMENT

Laptop	Huawei MateBook 14
Operation system	Ubuntu (64-bit)
CPU	Intel i5-1340P

B. Software Development Tool

The Mininet network simulation tool is applied to build network topology. The SDN controller algorithm is developed based on the Ryu framework using the PyCharm IDE, supported by the Python language (version: 3.8). Several Python modules are imported from the Ryu library during the development, listed in Figure 5.

```

from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet, ipv4, in_proto, tcp
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types

```

Fig. 5. The imported Python modules

C. Step of Implementation

The workflow of implementing the whole network system accords with the flowchart exhibited in Figure 6. The detailed implementation of the network topology and the SDN controller algorithm will be discussed in sequence.

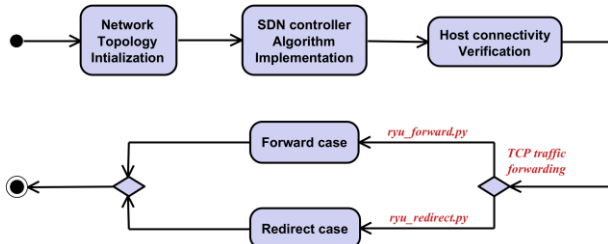


Fig. 6. The program flowchart of the implementation process

1) Network topology initialization

Initially, a blank network topology will be created with the subnet IP address and subnet mask set to be '10.0.1.0/24'.

Then, the switch and SDN controller will be added. The 'fail' mode of the switch will be configured to 'secure' to ensure that its forwarding decisions align precisely with the SDN controller's algorithm.

Subsequently, the client and server hosts will be configured, the network link, network interface, and IP & Mac address of which will be specifically set.

The network will be automatically built and started after the configuration. The Xterm terminal will be invoked for further code execution.

2) SDN controller algorithm implementation

Initially, a default flow table entry will be set which will send all the unmatched packets to the controller. A Python dictionary object will be created for recording the host Mac address and physical port data during the packet handling process.

3) Host connectivity verification

a) ARP request

Firstly, an ARP request packet sent from the client host will be received by the switch and sent to the controller.

The packet will be unpacked while the client host's Mac address and physical port data will be recorded by the controller. Then, the output packet will be formed and sent back to the switch through the data path by the controller. The switch will broadcast the ARP request packet to all server hosts for retrieving the ARP response.

b) ARP response

Then, the ARP response packet sent from the target server will be received by the switch and sent to the controller.

The ARP response packet will be unpacked while the server host's Mac address and physical port data will be recorded by the controller. Given the client host's Mac address

and physical port have already been recorded, the switch will directly forward the packet to the client host.

Subsequently, the controller will configure a flow table entry on the switch.

c) ICMP request

After the ARP addressing process, an ICMP request packet sent by the client host will be received by the switch and sent to the controller.

The ICMP request packet will be unpacked. Given the target server host's MAC address and physical port have already been recorded during the ARP addressing process, the packet will be directly forwarded to the target server host.

Then, a flow table entry will be configured on the switch to handle subsequent ICMP request packet forwarding operations.

d) ICMP response

The ICMP response packet sent by the target server host will be received by the switch and sent to the controller.

The ICMP response packet will be unpacked. Given the client host's Mac address and physical port have already been recorded, the packet will be forwarded directly to the client host by the switch.

Based on the packet information, another one-way flow table entry will be configured on the switch to handle subsequent ICMP response packet forwarding operations.

e) Further packet forwarding

After the configuration of two flow table entries, ICMP request & response packets can be efficiently and directly forwarded to the destination by the switch without invoking the controller.

4) TCP traffic forwarding

a) Code execution

The client-side and server-side applications will be started on the client host and server 1 host, sending and receiving TCP traffic segments respectively.

b) TCP SYN segment

The TCP SYN segment sent by the client host will be first received by the switch and sent to the controller.

The packet will be unpacked. Based on the recorded Mac address and physical port data during the ping testing process, the packet will be directly sent to the server 1 host's physical port by the switch.

Then, a flow table entry will be configured on the switch to handle subsequent TCP segments sent from the client host to the server 1 host.

c) TCP SYN/ACK segment

The TCP SYN/ACK segment sent by the server 1 host will be received by the switch and sent to the controller.

The packet will be unpacked by the packing handling algorithm. Based on the recorded Mac address and physical port data during the ping testing stage, the packet will be directly sent to the client host's physical port by the switch.

Then, a flow table entry will be configured on the switch to handle subsequent TCP packets sent from the server 1 host to the client host.

d) TCP ACK segment

Ultimately, the TCP ACK segment will be sent by the client host to the server 1 host to complete the TCP three-way handshake process.

Once the TCP ACK segment is received by the switch, it will be directly forwarded to the server 1 host based on the configured flow table entry.

5) TCP traffic redirect

a) Code execution

The client-side and server-side applications will be started on the client host and server 2 host, sending and receiving TCP traffic segments respectively.

b) TCP SYN segment

The TCP SYN segment sent by the client host will be first received by the switch and sent to the controller.

The packet will be unpacked. Then, the ethernet & IP header information of the packet will be checked for packet type verification.

For the packet sent to the MAC & IP address of the server 1 host, the controller will redefine the forwarding action and overwrite the ethernet & IP header information of the packet using the server 2 host's physical port and MAC & IP address, ensuring that the TCP SYN segment can be successfully received by the server 2 host.

The TCP packet will thereby be sent to the server 2 host's physical port while a flow table entry will be configured on the switch to redirect subsequent TCP segments sent from the client host to the server 1 host.

c) TCP SYN/ACK segment

The TCP SYN/ACK segment sent by the server 2 host will be received by the switch and sent to the controller.

The packet will be unpacked and the ethernet & IP header information of which will be verified.

Given the packet is sent to the MAC & IP address of the client host, the controller will overwrite the ethernet & IP header information using the server 1 host's MAC & IP address, ensuring the TCP SYN/ACK segment can be successfully received by the client host.

The TCP packet will be subsequently sent to the client host's physical port while a flow table entry will be configured on the switch to redirect subsequent TCP segments sent from the server 2 host to the client host.

d) TCP ACK segment

Ultimately, the TCP ACK segment will be sent by the client host to the server 1 host to complete the TCP three-way handshake process.

Once the TCP ACK segment is received by the switch, it will be directly redirected to the server 2 host based on the configured flow table entry.

D. Implementation Difficulties

The biggest difficulty of the implementation process is the realization of the TCP traffic redirection algorithm. Initially, only the physical output port will be redefined by the controller for redirecting the packet, in which case, however, the TCP segment will not be successfully received by the

target host due to the mismatch of the ethernet & IP header information.

Consequently, the SDN controller algorithm has been redesigned to enable the functionality of the packet header modification.

E. Programming Skills

1) Program modular design

The whole SDN controller algorithm is divided into several specific functions, enhancing the readability and maintainability of the code.

2) Abstraction

Complex operations such as the initialization of switch features and the configuration of flow table entries are specifically encapsulated.

3) Object-oriented Programming (OOP)

The SDN controller algorithm is designed based on the OOP methods by inheriting the functionality of the parent class 'app_manager.RyuApp', successfully achieving the customization and extension of the Ryu framework features.

4) Event-drive programming

The event-driven mechanism of the Ryu framework is leveraged, enabling the controller to react dynamically to network changes to ensure adaptability and responsiveness in handling OpenFlow events.

V. TESTING

A. Testing Environment

The Linux virtual machine supported by the VirtualBox is leveraged for testing the Mininet network topology and the SDN controller algorithm.

B. Testing Steps

1) Mininet network topology initialization

By entering the command 'sudo python3 networkTopo.py' in the terminal, the Python file networkTopo.py is successfully executed for creating the Mininet network topology, shown in Figure 7. The Xterm control pane is invoked for further code execution.

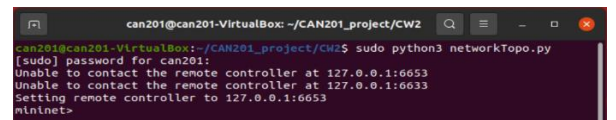


Fig. 7. The execution result of the network topology code

2) Host's IP & Mac address verification

After the initialization of the network topology, the IP & Mac address of each host are verified by entering 'ifconfig' command in the terminal, demonstrated by Figure 17&18&19 in the appendix.

3) Ryu SDN controller algorithm implementation

By entering the command 'sudo ryu-manager ryu_forward.py' in the terminal, the Ryu SDN controller algorithm is successfully implemented, as shown in Figure 8.

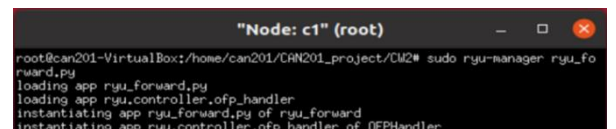


Fig. 8. The implementation of the Ryu SDN controller algorithm

4) Ping testing

After the implementation of the SDN controller algorithm, the ping testing is conducted. By entering the command 'ping 10.0.1.2' and 'ping 10.0.1.3' in the terminal, it can be observed from Figure 20&21 in the appendix that the ICMP packet sent by the client host is successfully received by the server 1&2 hosts.

By entering the command 'sudo ovs-ofctl dump-flows s1 -O OpenFlow13' in the terminal, the configured flow table entries are verified, as shown in Figure 22&23 in the appendix.

5) TCP traffic forward

After the ping testing, the server application will be implemented on the server 1&2 host for receiving the TCP segment by entering the command 'sudo python3 server.py' in the terminal. The client application will be implemented on the client host after 5 seconds by entering the command 'sudo python3 client.py' in the terminal.

The configured flow table entry is verified by entering the command 'sudo ovs-ofctl dump-flows s1 -O OpenFlow13' in the terminal, as demonstrated by Figure 24&25 in the appendix.

a) Forwarding case

It can be seen from Figure 9&10 that the TCP segment sent by the client host is successfully forwarded to the server 1 host.

```

"Node: client"
root@can201-VirtualBox:/home/can201/CAN201_project/CW2# sudo python3 client.py
TCP client sending to server...
from server (10.0.1.2.3999): Hello, client (10.0.1.5)! This is server (10.0.1.2)
from server (10.0.1.2.3999): Hello, client (10.0.1.5)! This is server (10.0.1.2)
from server (10.0.1.2.3999): Hello, client (10.0.1.5)! This is server (10.0.1.2)
from server (10.0.1.2.3999): Hello, client (10.0.1.5)! This is server (10.0.1.2)
from server (10.0.1.2.3999): Hello, client (10.0.1.5)! This is server (10.0.1.2)
from server (10.0.1.2.3999): Hello, client (10.0.1.5)! This is server (10.0.1.2)
from server (10.0.1.2.3999): Hello, client (10.0.1.5)! This is server (10.0.1.2)
from server (10.0.1.2.3999): Hello, client (10.0.1.5)! This is server (10.0.1.2)

```

Fig. 9. The TCP traffic forwarding result (client host)

```

"Node: server_1"
root@can201-VirtualBox:/home/can201/CAN201_project/CW2# sudo python3 server.py
TCP server bind on 9999...
accepted ('10.0.1.5', 38792)
from client (10.0.1.5.38792): seq=0 Hello, server (10.0.1.2)
from client (10.0.1.5.38792): seq=1 Hello, server (10.0.1.2)
from client (10.0.1.5.38792): seq=2 Hello, server (10.0.1.2)
from client (10.0.1.5.38792): seq=3 Hello, server (10.0.1.2)
from client (10.0.1.5.38792): seq=4 Hello, server (10.0.1.2)
from client (10.0.1.5.38792): seq=5 Hello, server (10.0.1.2)
from client (10.0.1.5.38792): seq=6 Hello, server (10.0.1.2)

```

Fig. 10. The TCP traffic forwarding result (server 1 host)

b) Redirecting case

It can be seen from Figure 11&12 that the TCP segment sent by the client is successfully redirected to the server 2 host.

```

"Node: client"
root@can201-VirtualBox:/home/can201/CAN201_project/CW2# sudo python3 client.py
TCP client sending to server...
from server (10.0.1.2.3999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.3999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.3999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.3999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.3999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.3999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.3999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.3999): Hello, client (10.0.1.5)! This is server (10.0.1.3)

```

Fig. 11. The TCP traffic redirecting result (client host)

```

"Node: server_2"
root@can201-VirtualBox:/home/can201/CAN201_project/CW2# sudo python3 server.py
TCP server bind on 9999...
accepted ('10.0.1.5', 38798)
from client (10.0.1.5.38798): seq=0 Hello, server (10.0.1.2)
from client (10.0.1.5.38798): seq=1 Hello, server (10.0.1.2)
from client (10.0.1.5.38798): seq=2 Hello, server (10.0.1.2)
from client (10.0.1.5.38798): seq=3 Hello, server (10.0.1.2)
from client (10.0.1.5.38798): seq=4 Hello, server (10.0.1.2)
from client (10.0.1.5.38798): seq=5 Hello, server (10.0.1.2)

```

Fig. 12. The TCP traffic redirecting result (server 2 host)

6) TCP packet capture

By entering the command 'sudo tcpdump -i s1-eth1 -w data.pcap' in the terminal, packets transmitted through the s1-eth1 interface will be captured by the Tcpdump, the data of which will be stored in a pcap file. It can be observed that 28 packets have been successfully captured based on Figure 13.

```

can201@can201-VirtualBox: ~/CAN201_project/CW2/TCP_rec...
can201@can201-VirtualBox:~/CAN201_project/CW2/TCP_record/forward$ sudo tcpdump -i s1-eth1 -w data_1.pcap
tcpdump: listening on s1-eth1, link-type EN10MB (Ethernet), capture size 262144 bytes
^C28 packets captured
28 packets received by filter
0 packets dropped by kernel

```

Fig. 13. The utilization of tcpdump for packet capturing

7) Network latency calculation

By interpreting the saved pcap file shown in Figure 14, the network latency of the TCP 3-way handshake process can be calculated.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.1.5	10.0.1.2	TCP	74	58796 → 9999 [SYN] Seq=0 Win=0 Len=0
2	0.004156	10.0.1.2	10.0.1.5	TCP	74	9999 → 58796 [SYN, ACK] Seq=0 Ack=1 Len=0
3	0.004168	10.0.1.5	10.0.1.2	TCP	66	58796 → 9999 [ACK] Seq=1 Ack=1 Len=0

Fig. 14. The network communication data

C. Testing Result

To ensure the generalizability of the experimental results, ten repeated experiments were conducted. The experimental result is manifested in Figure 15&16 as follows.

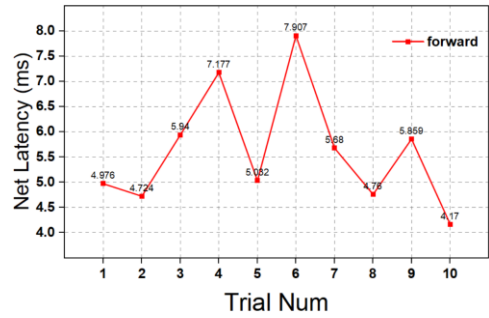


Fig. 15. The network latency of the TCP three-way handshake (forward)

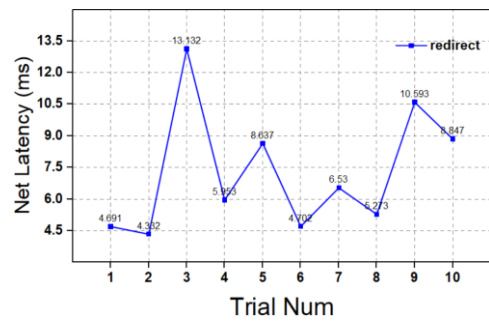


Fig. 16. The network latency of the TCP three-way handshake (redirect)

Based on the data demonstrated above, the average network latency of the forward & redirect cases can be calculated, which are 5.620 milliseconds and 7.269 milliseconds. Therefore, it can be concluded that the forward case exhibits a slight time difference compared to the redirect case. This may be mainly due to the header modification operation conducted by the redirect algorithm.

VI. CONCLUSION

In summary, an SDN-based network system is developed and evaluated in this project. The network topology is constructed using the Mininet while the SDN controller

algorithm is designed based on the Ryu framework, comprising the function of TCP traffic forward & redirect. The network latency is calculated based on the packet capture data, demonstrating the validity of the algorithm.

For future work, the deployment of the SDN controller algorithm in real-world network environments should be conducted to further evaluate and improve its stability.

ACKNOWLEDGMENT

Pengkai Chen (2251486) contributes 25% to the project, Yilin Li (2255705) contributes 25% to the project, Peiling Tu (2251487) contributes 25% to the project and Jinhong Jiang (2251601) contributes 25% to the project.

REFERENCES

- [1] K. Benzekki, A. El Fergougui, and A. Elbelrhiti Elaloui, "Software - defined networking (SDN): a survey," *Security and communication networks*, vol. 9, no. 18, pp. 5803-5833, 2016.
- [2] R. L. S. de Oliveira, A. A. Shinoda, C. M. Schweitzer, and L. Rodrigues Prete, "Using Mininet for emulation and prototyping Software-Defined Networks," 2014 IEEE Colombian Conference on Communications and Computing (COLCOM), pp. 1-6, 2014.
- [3] K. Giotis, G. Androulidakis, and B. S. Maglaris, "A scalable anomaly detection and mitigation architecture for legacy networks via an OpenFlow middlebox," *Secur. Commun. Networks*, vol. 9, pp. 1958-1970, 2016.
- [4] A. Lara, A. Kolasani, and B. Ramamurthy, "Network Innovation using OpenFlow: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 16, pp. 493-512, 2014.
- [5] M. He, M.-Y. Huang, and W. Kellerer, "Optimizing the Flexibility of SDN Control Plane," *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pp. 1-9, 2020.
- [6] W. Braun and M. Menth, "Software-Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices," *Future Internet*, vol. 6, pp. 302-336, 2014.
- [7] P. Wang, J. Lan, and S. Chen, "OpenFlow based flow slice load balancing," *China Communications*, vol. 11, pp. 72-82, 2014.
- [8] J. Miguel-Alonso, "A Research Review of OpenFlow for Datacenter Networking," *IEEE Access*, vol. 11, pp. 770-786, 2023.

APPENDIX

```

"Node: client"
root@can201-VirtualBox:/home/can201/CN201_project/CN2# ifconfig
client-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.0.1.5 netmask 255.255.0.0 broadcast 10.0.1.255
    inet6 fe80::200:fff:fe01: prefixlen 64 scopeid 0x20<link>
    ether 00:00:00:00:00:05 txqueuelen 1000 (Ethernet)
    RX packets 27 bytes 4012 (4.0 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 12 bytes 996 (996.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

Fig. 17. The client host IP & Mac address verification

```

"Node: server_1"
root@can201-VirtualBox:/home/can201/CN201_project/CN2# ifconfig
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

server_1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.0.1.2 netmask 255.255.255.0 broadcast 10.0.1.255
    inet6 fe80::200:fff:fe01: prefixlen 64 scopeid 0x20<link>
    ether 00:00:00:00:00:01 txqueuelen 1000 (Ethernet)
    RX packets 23 bytes 4285 (4.2 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 12 bytes 996 (996.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

Fig. 18. The server 1 host IP & Mac address verification

```

"Node: server_2"
root@can201-VirtualBox:/home/can201/CN201_project/CN2# ifconfig
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

server_2-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.0.1.3 netmask 255.255.255.0 broadcast 10.0.1.255
    inet6 fe80::200:fff:fe02: prefixlen 64 scopeid 0x20<link>
    ether 00:00:00:00:00:02 txqueuelen 1000 (Ethernet)
    RX packets 23 bytes 4285 (4.2 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 13 bytes 1065 (1.0 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

Fig. 19. The server 2 host IP & Mac address verification

```

"Node: client"
root@can201-VirtualBox:/home/can201/CN201_project/CN2# ping 10.0.1.2
PING 10.0.1.2 (10.0.1.2) 56(84) bytes of data:
 64 bytes from 10.0.1.2: icmp_seq=1 ttl=64 time=6.14 ms
 64 bytes from 10.0.1.2: icmp_seq=2 ttl=64 time=0.247 ms
 64 bytes from 10.0.1.2: icmp_seq=3 ttl=64 time=0.055 ms
 64 bytes from 10.0.1.2: icmp_seq=4 ttl=64 time=0.074 ms
 64 bytes from 10.0.1.2: icmp_seq=5 ttl=64 time=0.063 ms
 64 bytes from 10.0.1.2: icmp_seq=6 ttl=64 time=0.442 ms
^C
--- 10.0.1.2 ping statistics ---
 6 packets transmitted, 6 received, 0% packet loss, time 510ms
rtt min/avg/max/mdev = 0.055/1.170/6.142/2.227 ms

```

Fig. 20. The ping testing result (client >> server 1)

```

"Node: client"
root@can201-VirtualBox:/home/can201/CN201_project/CN2# ping 10.0.1.3
PING 10.0.1.3 (10.0.1.3) 56(84) bytes of data:
 64 bytes from 10.0.1.3: icmp_seq=1 ttl=64 time=4.89 ms
 64 bytes from 10.0.1.3: icmp_seq=2 ttl=64 time=0.176 ms
 64 bytes from 10.0.1.3: icmp_seq=3 ttl=64 time=0.069 ms
 64 bytes from 10.0.1.3: icmp_seq=4 ttl=64 time=0.078 ms
 64 bytes from 10.0.1.3: icmp_seq=5 ttl=64 time=0.082 ms
 64 bytes from 10.0.1.3: icmp_seq=6 ttl=64 time=0.081 ms
^C
--- 10.0.1.3 ping statistics ---
 6 packets transmitted, 6 received, 0% packet loss, time 5094ms
rtt min/avg/max/mdev = 0.078/0.893/4.893/1.785 ms

```

Fig. 21. The ping testing result (client >> server 2)

```

"Node: s1" (root)
root@can201-VirtualBox:/home/can201/CN201_project/CN2# sudo ovs-ofctl -O OpenFlow13 dump-flows s1
cookie=0x0, duration=5.252s, table=0, n_packets=5, n_bytes=490, idle_timeout=5, priority=1,icmp,in_port="s1-eth1",nw_src=10.0.1.5,nw_dst=10.0.1.2 actions=output:"s1-eth2"
cookie=0x0, duration=6.249s, table=0, n_packets=5, n_bytes=490, idle_timeout=5, priority=1,icmp,in_port="s1-eth2",nw_src=10.0.1.2,nw_dst=10.0.1.5 actions=output:"s1-eth1"
cookie=0x0, duration=1.158s, table=0, n_packets=0, n_bytes=0, idle_timeout=5, priority=1,arp,in_port="s1-eth2",dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:03 actions=output:"s1-eth1"
cookie=0x0, duration=1.156s, table=0, n_packets=0, n_bytes=0, idle_timeout=5, priority=1,arp,in_port="s1-eth1",dl_src=00:00:00:00:00:00:03,dl_dst=00:00:00:00:00:01 actions=output:"s1-eth2"
cookie=0x0, duration=8.229s, table=0, n_packets=5, n_bytes=364, priority=0 actions=CONTROLLER:65535

```

Fig. 22. The configured flow table entries (ping: client >> server 1)

```

"Node: s1" (root)
root@can201-VirtualBox:/home/can201/CN201_project/CN2# sudo ovs-ofctl -O OpenFlow13 dump-flows s1
cookie=0x0, duration=40.355s, table=0, n_packets=39, n_bytes=3822, idle_timeout=5, priority=1,icmp,in_port="s1-eth1",nw_src=10.0.1.5,nw_dst=10.0.1.3 actions=output:"s1-eth2"
cookie=0x0, duration=40.353s, table=0, n_packets=39, n_bytes=3822, idle_timeout=5, priority=1,icmp,in_port="s1-eth3",nw_src=10.0.1.3,nw_dst=10.0.1.5 actions=output:"s1-eth1"
cookie=0x0, duration=117.363s, table=0, n_packets=26, n_bytes=1540, priority=0 actions=CONTROLLER:65535

```

Fig. 23. The configured flow table entries (ping: client >> server 2)

```

"Node: s1" (root)
root@can201-VirtualBox:/home/can201/CN201_project/CN2# sudo ovs-ofctl -O OpenFlow13 dump-flows s1
cookie=0x0, duration=5.923s, table=0, n_packets=13, n_bytes=1038, idle_timeout=5, priority=1,tcp,nw_src=10.0.1.5,nw_dst=10.0.1.2,tp_src=38828,tp_dst=9999 actions=output:"s1-eth2"
cookie=0x0, duration=5.917s, table=0, n_packets=10, n_bytes=966, idle_timeout=5, priority=1,tcp,nw_src=10.0.1.2,nw_dst=10.0.1.5,tp_src=9999,tp_dst=38828 actions=output:"s1-eth1"
cookie=0x0, duration=309.394s, table=0, n_packets=52, n_bytes=2808, priority=0 actions=CONTROLLER:65535

```

Fig. 24. The configured flow table entries (TCP forward)

```

"Node: s1" (root)
root@can201-VirtualBox:/home/can201/CN201_project/CN2# sudo ovs-ofctl -O OpenFlow13 dump-flows s1
cookie=0x0, duration=17.141s, table=0, n_packets=36, n_bytes=2924, idle_timeout=5, priority=1,tcp,nw_src=10.0.1.5,nw_dst=10.0.1.2,tp_src=38832,tp_dst=9999 actions=set_field:00:00:00:00:00:02->eth_dst,set_field:10.0.1.3->ip_dst,output:"s1-eth3"
cookie=0x0, duration=17.134s, table=0, n_packets=21, n_bytes=2253, idle_timeout=5, priority=1,tcp,nw_src=10.0.1.3,nw_dst=10.0.1.5,tp_src=9999,tp_dst=38832 actions=set_field:00:00:00:00:00:01->eth_src,set_field:10.0.1.2->ip_src,output:"s1-eth1"
cookie=0x0, duration=159.921s, table=0, n_packets=23, n_bytes=1438, priority=0 actions=CONTROLLER:65535

```

Fig. 25. The configured flow table entries (TCP redirect)