Network Traffic Regulation via SDN Controller

Shuchen Ji
2034172

Xinxin Fan
2039125

Shiqi Huang
2034174

***Abstract*—This project aims to create a simple SDN network topology and simulate the flow control process with SDN flow entries. In doing so, a simple SDN network topology was first built using the Mininet python library. Secondly, two SDN controller applications based on the Ryu framework were developed. The first controller application was designed to allow all traffic to be sent to Server 1, and the second controller application made the traffic sent to Server 1 be redirected to Server 2. After completing the code for both controller applications, the networking latency on Client was captured and calculated. Finally, some conclusions were reached after testing.**

I. Introduction

Software-defined networking (SDN) is a new network innovation architecture to realize network virtualization. Its biggest feature is that it is flexible, allowing users to control addresses, routing, security and topology rather than hardware vendors. The generation of SDN goes through three stages: active network, control separation and OpenFlow. The concept of SDN was put forward in the Clean Slate Program to separate the control plane [1] and the data plane, central control and network programming with well-defined interfaces.

In this project, there are many challenges to be solved. We first used the Mininet Python library to build a simple network topology to run on a virtual machine. We then designed and ran an SDN controller application based on the Ryu framework. Finally, we need to run server.py on two servers and client.py on the client.

Some potential applications can be implemented through this project:

- Using SDN assistance technology to determine and guide the optimal route to execute information business in heterogeneous vehicle organizations [2].
- Controlling the traffic load of WiFi access points through SDN to improve the service quality of smart city applications [3].

To solve these challenges, we searched for relevant work to further understand the forwarding and redirection functions of SDN controller applications. Then, through the design of the network system design diagram, workflow and pseudo code, we implemented the forwarding and redirection work on Client, Server1and Server2, and tested their networking latency respectively. Finally, we do some future outlooks according to our present work.

II. Related Work

With the rapid development of the Internet, the traditional network has become a limitation. Due to the limited physical topology that can be formed, the traditional network traffic control performance deteriorates dramatically when the traffic reaches a certain amount [1]. Therefore, SDN is proposed to separate the control plane and data plane and carry out centralized control, which redefines the traditional network management mode.

There are many reports and studies on SDN controller in-depth research. A multimedia service architecture supporting local traffic redirection is proposed by Hwang to improve the quality of service in Ethernet Passive Optical Network [4]. In addition, Shaikh and Darekar determined the performance of SDN controller through different networking scenarios [5]. Hamano‘s team proposed a defense mechanism based on traffic redirection to deal with DoS/DDoS or worm attacks [6].

Through these studies, we will design an SDN flow controller in this project to control the flow as efficiently and safely as possible.

III. Design

A. The network system design diagram

A simple network topology was built to simulate the traffic regulation process by SDN switch and SDN controller. In this topology, one client and two servers were created. The three hosts were all connected to the SDN switch, which is responsible for the forwarding service of data packets. A remote SDN controller was added to this topology to regulate the traffic by maintaining the flow table. The topology is shown in Fig 1.

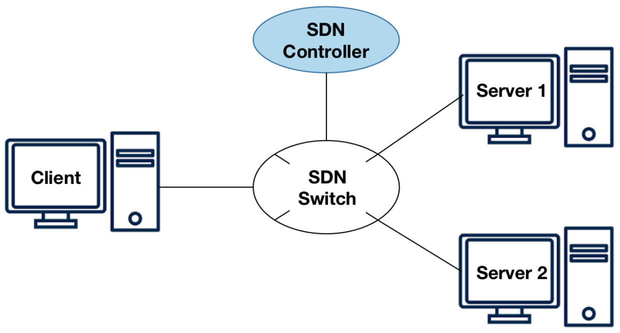


Fig. 1. The network system design diagram

B. The workflow

When Client tries to connect to Server 1, the Client will send a TCP SYN segment to the SDN switch. If the flow table in the switch does not have a flow entry that matches the received segment, the switch will send a Packet_in SDN packet based on the OpenFlow protocol and forward it to the controller. Then, the controller will create a flow entry and send a Packet_out SDN packet to the SDN switch. At the same time, the controller will install this flow entry to its flow table to avoid the flood. The switch will read the flow entry and forward the TCP SYN segment to Server 1. The process of creating and installing the flow entry is shown in Fig 2. The process is reversed when Server 1 sends the SYN-ACK segment to reply to Client.

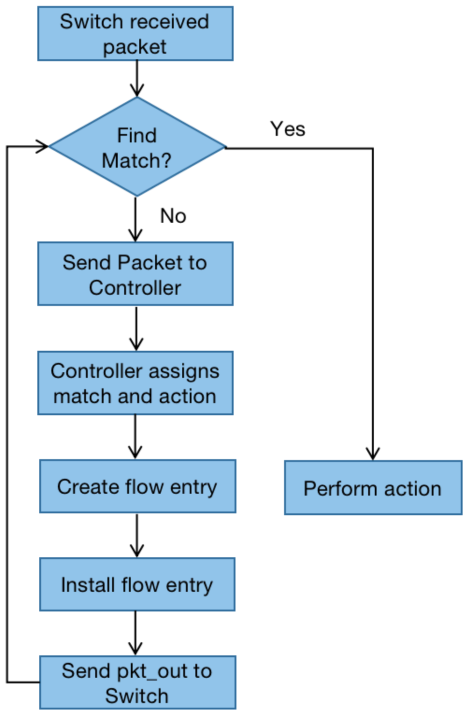


Fig. 2. The workflow of creating and installing the flow entry

C. *The algorithm*

The pseudo-code below shows how the SDN controller redirects traffic to server 2 while Client communicates with Server 1. The function presented will first learn the input port mac address to avoid the flood. To redirect the traffic, if the source IP address in the packet points to Client and the destination in the match is Server 1, then the function will alter the destination in action to Server 2. A similar process is done when creating a flow entry to send the packet to Client. Then, the controller will install these flow entries to its flow table.

Algorithm 1 Kernel Pseudo Code of Network Traffic Redirection

```
Input in_port, out_port, packet, Client['mac', 'ip'], Server1['mac', 'ip'], Server2['mac', 'ip']
Output packet_out packet with flow entry
 1: mac_to_port[dpid][src] = in_port
 2: if dst in mac_to_port[dpid] then
 3:     out_port = mac_to_port[dpid][dst]
 4: else flood all ports
 5: if out_port != flood_all_ports and protocol = TCP_protocol then
 6:     if Server2['mac'] in mac_to_port[dpid] and srcip == Client['ip'] then
 7:         out_port = mac_to_port[dpid][Server2['mac']]
 8:         Change the IP destination to Server1 in Match
 9:         Change the destination eth_dst and ipv4_dst to Server2 in Action
10:     else if Client['mac'] in mac_to_port[dpid] and srcip == Server2['ip'] then
11:         out_port = mac_to_port[dpid][Client['mac']]
12:         Change the IP destination to Client in Match
13:         Change the eth_src and ipv4_src to Server1 in Action
14:     else Flood all ports
15: Check valid buffer ID
16: if Buffer id exists then
17:     Add flow entry with buffer.ID
18: else if then
19:     Add flow entry without buffer.ID to flow table
20: Create packet_out packet to send
21: Send out the packet
```

IV. Implementation

A. *Working environment*

The project is developed in the following host environment:

- Mac OS [host CPU: Apple M1 Pro, physical memory: 16GB, operation system: arm64]
- Ubuntu [CPU: Intel(R) Core (TM) i7-11800H CPU @ 2.30GHz, physical memory: 3.8 Gi, operation system: 64-bit)]

The development software used is PyCharm 2022.2.3 (Professional Edition). The Ryu controller and python libraries such as Mininet, socket and time are used for this project.

B. *Implementation steps*

Here are the implementation steps of the program□

After Client connects to Server1, Client sends a TCP SYN segment to the SDN switch. Then the traffic redirection function will be performed according to the following steps (The flow chart is shown in Fig 3):

1. If the flow table in the SDN switch does not match the flow entry of the received segment, a Packet_In SDN packet will be sent from the switch to the controller.

2. The program parses the data package and extracts the source mac address, destination mac address and port information. Since all the following traffic sent from Client to Server1 need to be redirected to Server2, some IP and mac information have been modified and the out_port also changed from Server1 to Server2.
3. Then the controller creates a flow entry and sends a Packet_out SDN packet to the switch.
4. If the flow table in the SDN switch matches the flow entry of the received segment, the switch can forward the TCP SYN segment to Server2. Otherwise, step 1 to step 4 need to be executed repeatedly.
5. After Server2 successfully receives the request from Client, Server2 also need to reply to Client. Therefore, the operations similar to steps 1, 2, 3, and 4 will be executed with the reverse traffic forwarding path.
6. After completing all the above steps, all the traffic sent from Client to Server1 will be redirected to Server2 successfully.

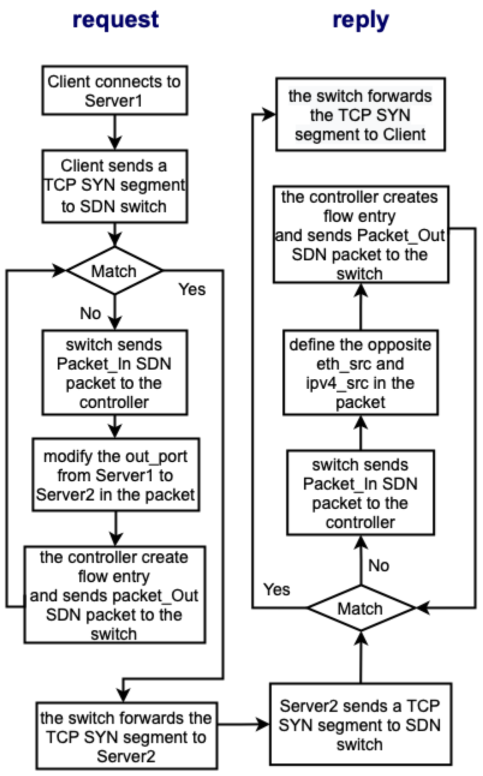

request
reply
Client connects to Server1
Client sends a TCP SYN segment to SDN switch
Match
Yes
No
switch sends Packet_In SDN packet to the controller
modify the out_port from Server1 to Server2 in the packet
the controller create flow entry and sends packet_Out SDN packet to the switch
the switch forwards the TCP SYN segment to Server2
Server2 sends a TCP SYN segment to SDN switch
Match
Yes
No
switch sends Packet_In SDN packet to the controller
define the opposite eth_src and ipv4_src in the packet
the controller creates flow entry and sends Packet_Out SDN packet to the switch
the switch forwards the TCP SYN segment to Client

Fig. 3. The flow chart of the traffic redirection function

C. Difficulties

While doing this project, some obstacles were met. For instance, the error information called 'local variable 'match' referenced before the assignment' appeared when the code was executed. It implies the variable 'match' is directly referenced without being assigned. However, the code has defined the value of 'match'. After searching for some documents, the reason was founded. The code only executed the part of the response, and there was no data that could reply to Client, so there was no value for 'match'.

V. Testing and Results

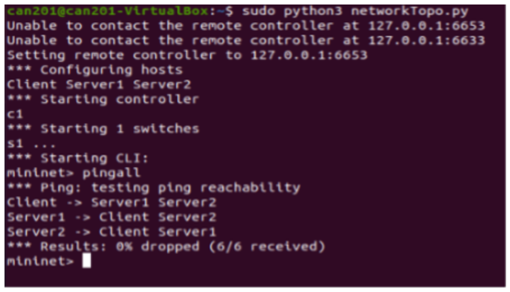
A. testing environment

- one virtual machine of ubuntu system
- Ubuntu [CPU: Intel(R) Core (TM) i7-11800H CPU @ 2.30GHz, physical memory: 3.8 Gi, operation system: 64-bit)]

B. testing steps

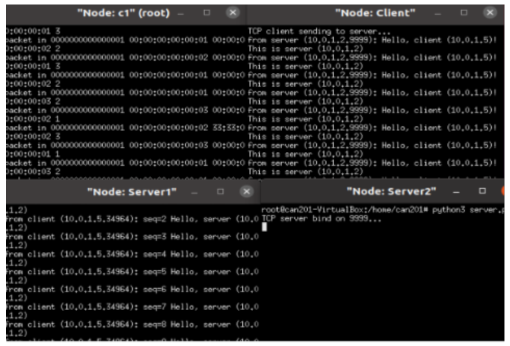
1) Forwarding:

1. Running 'sudo python3 networkTopo.py' at the terminal. Then, five sub windows will pop up on the screen.
2. In the controller, enter 'ryu manager ryu_forward.py'. Then go back to the Mininet window, and print the command 'pingall'. If the result is 0% dropped, as shown in the screenshot below, every node is reachable with each other.



```
can201@can201-VirtualBox:~$ sudo python3 networkTopo.py
Unable to contact the remote controller at 127.0.0.1:6653
Unable to contact the remote controller at 127.0.0.1:6633
Setting remote controller to 127.0.0.1:6653
*** Configuring hosts
Client Server1 Server2
*** Starting controller
c1
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
Client -> Server1 Server2
Server1 -> Client Server2
Server2 -> Client Server1
*** Results: 0% dropped (6/6 received)
mininet> 
```

3. Create a new window and enter 'sudo wireshark' to capture the packets. In Server1 and Server2, enter python3 server.py. In Client, enter 'python3 client.py'. (Notice that wait 5 seconds after ping because of the idle timeout and then start running the client.py)
Here are the running results:



"Node: c1" (root)

```
0:00:00:01 3
packet in 0000000000000001 00:00:00:00:00:01 00:00:0
0:00:00:02 2
packet in 0000000000000001 00:00:00:00:00:02 00:00:0
0:00:00:01 3
packet in 0000000000000001 00:00:00:00:00:01 00:00:0
0:00:00:02 2
packet in 0000000000000001 00:00:00:00:00:01 00:00:0
0:00:00:03 2
packet in 0000000000000001 00:00:00:00:00:03 00:00:0
0:00:00:02 1
packet in 0000000000000001 00:00:00:00:00:02 33:33:0
0:00:00:02 3
packet in 0000000000000001 00:00:00:00:00:03 00:00:0
0:00:00:01 1
packet in 0000000000000001 00:00:00:00:00:01 00:00:0
0:00:00:03 2
```

"Node: Client"

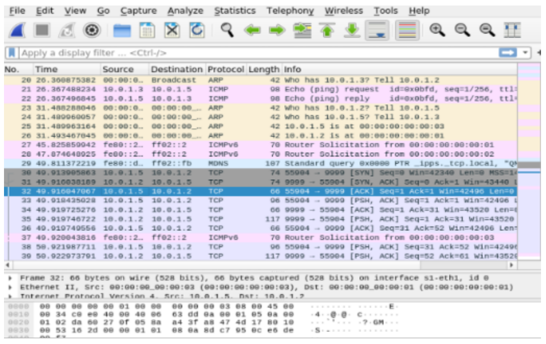
```
TCP client sending to server...
From server (10.0.1.2,9999): Hello, client (10.0.1.5)!
This is server (10.0.1.2)
From server (10.0.1.2,9999): Hello, client (10.0.1.5)!
This is server (10.0.1.2)
From server (10.0.1.2,9999): Hello, client (10.0.1.5)!
This is server (10.0.1.2)
From server (10.0.1.2,9999): Hello, client (10.0.1.5)!
This is server (10.0.1.2)
From server (10.0.1.2,9999): Hello, client (10.0.1.5)!
This is server (10.0.1.2)
From server (10.0.1.2,9999): Hello, client (10.0.1.5)!
This is server (10.0.1.2)
From server (10.0.1.2,9999): Hello, client (10.0.1.5)!
This is server (10.0.1.2)
From server (10.0.1.2,9999): Hello, client (10.0.1.5)!
This is server (10.0.1.2)
```

"Node: Server1"

```
.1.2)
From client (10.0.1.5,34964): seq=2 Hello, server (10.0
.1.2)
From client (10.0.1.5,34964): seq=3 Hello, server (10.0
.1.2)
From client (10.0.1.5,34964): seq=4 Hello, server (10.0
.1.2)
From client (10.0.1.5,34964): seq=5 Hello, server (10.0
.1.2)
From client (10.0.1.5,34964): seq=6 Hello, server (10.0
.1.2)
From client (10.0.1.5,34964): seq=7 Hello, server (10.0
.1.2)
From client (10.0.1.5,34964): seq=8 Hello, server (10.0
.1.2)
```

"Node: Server2"

```
root@can201-VirtualBox:/home/can201# python3 server.p
TCP server bind on 9999...
█
```

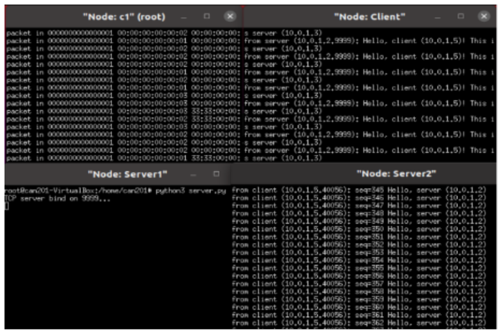

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.  Time          Source       Destination  Protocol Length Info
  20 26.360875382  00:00:0..    Broadcast    ARP        42 Who has 10.0.1.3? Tell 10.0.1.2
  21 26.367488234  10.0.1.3     10.0.1.5     ICMP       98 Echo (ping) request  id=0x0bfd, seq=1/256, ttl=
  22 26.367496845  10.0.1.5     10.0.1.3     ICMP       98 Echo (ping) reply    id=0x0bfd, seq=1/256, ttl=
  23 31.488288046  00:00:0..    00:00:00_..  ARP        42 Who has 10.0.1.2? Tell 10.0.1.5
  24 31.489960057  00:00:0..    00:00:00_..  ARP        42 Who has 10.0.1.5? Tell 10.0.1.3
  25 31.489963164  00:00:0..    00:00:00_..  ARP        42 10.0.1.5 is at 00:00:00:00:00:03
  26 31.493467045  00:00:0..    00:00:00_..  ARP        42 10.0.1.2 is at 00:00:00:00:00:01
  27 45.825859042  fe80::2..    ff02::2      ICMPv6     70 Router Solicitation from 00:00:00:00:00:01
  28 47.874648925  fe80::2..    ff02::2      ICMPv6     70 Router Solicitation from 00:00:00:00:00:02
  29 49.811372219  fe80::d..    ff02::fb     MDNS      107 Standard query 0x0000 PTR _ipps._tcp.local, "QM
  30 49.913905863  10.0.1.5     10.0.1.2     TCP        74 55904 → 9999 [SYN] Seq=0 Win=42340 Len=0 MSS=14
  31 49.916038189  10.0.1.2     10.0.1.5     TCP        74 9999 → 55904 [SYN, ACK] Seq=0 Ack=1 Win=43440 L
  32 49.916047067  10.0.1.5     10.0.1.2     TCP        66 55904 → 9999 [ACK] Seq=1 Ack=1 Win=42496 Len=0
  33 49.918435028  10.0.1.5     10.0.1.2     TCP        96 55904 → 9999 [PSH, ACK] Seq=1 Ack=1 Win=42496 L
  34 49.919725276  10.0.1.2     10.0.1.5     TCP        66 9999 → 55904 [ACK] Seq=1 Ack=31 Win=43520 Len=0
  35 49.919746722  10.0.1.2     10.0.1.5     TCP       117 9999 → 55904 [PSH, ACK] Seq=1 Ack=31 Win=43520
  36 49.919749556  10.0.1.5     10.0.1.2     TCP        66 55904 → 9999 [ACK] Seq=31 Ack=52 Win=42496 Len=
  37 49.920043816  fe80::2..    ff02::2      ICMPv6     70 Router Solicitation from 00:00:00:00:00:03
  38 50.921987711  10.0.1.5     10.0.1.2     TCP        96 55904 → 9999 [PSH, ACK] Seq=31 Ack=52 Win=42496
  39 50.922973791  10.0.1.2     10.0.1.5     TCP       117 9999 → 55904 [PSH, ACK] Seq=52 Ack=61 Win=43520

▸ Frame 32: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface s1-eth1, id 0
▸ Ethernet II, Src: 00:00:00_00:00:03 (00:00:00:00:00:03), Dst: 00:00:00_00:00:01 (00:00:00:00:00:01)
▸ Internet Protocol Version 4, Src: 10.0.1.5, Dst: 10.0.1.2

0000  00 00 00 00 00 01 00 00  00 00 00 03 08 00 45 00   ........ ......E.
0010  00 34 c0 e0 40 00 40 06  63 dd 0a 00 01 05 0a 00   .4..@.@. c.......
0020  01 02 da 60 27 0f 05 8a  a4 3f a8 47 4d 17 80 10   ...`'... .?.GM...
0030  00 53 16 2d 00 00 01 01  08 0a 8d c7 95 0c e6 de   .S.-.... ........
```

2) *Redirection:* Redirection executes the same operations as forwarding (replace ryu_ forward.py with ryu_ redirect.py). Here are the running results:

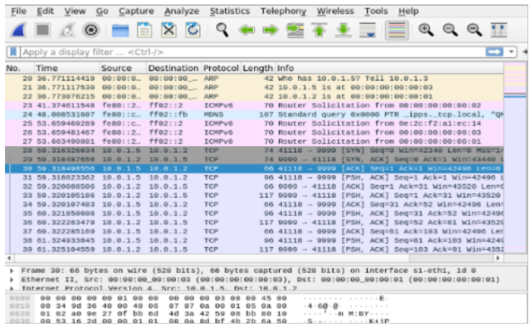


```
packet in 0000000000000001 00:00:00:00:00:02 00:00:00:00:
packet in 0000000000000001 00:00:00:00:00:01 00:00:00:00:
packet in 0000000000000001 00:00:00:00:00:02 00:00:00:00:
packet in 0000000000000001 00:00:00:00:00:02 00:00:00:00:
packet in 0000000000000001 00:00:00:00:00:02 00:00:00:00:
packet in 0000000000000001 00:00:00:00:00:01 00:00:00:00:
packet in 0000000000000001 00:00:00:00:00:02 00:00:00:00:
packet in 0000000000000001 00:00:00:00:00:01 00:00:00:00:
packet in 0000000000000001 00:00:00:00:00:03 00:00:00:00:
packet in 0000000000000001 00:00:00:00:00:03 00:00:00:00:
packet in 0000000000000001 00:00:00:00:00:03 33:33:00:00:
packet in 0000000000000001 00:00:00:00:00:02 33:33:00:00:
packet in 0000000000000001 00:00:00:00:00:03 00:00:00:00:
packet in 0000000000000001 00:00:00:00:00:02 00:00:00:00:
packet in 0000000000000001 00:00:00:00:00:02 00:00:00:00:
packet in 0000000000000001 00:00:00:00:00:02 00:00:00:00:
packet in 0000000000000001 00:00:00:00:00:01 33:33:00:00:
```

```
s server (10.0.1.3)
from server (10.0.1.2,9999): Hello, client (10.0.1.5)! This i
s server (10.0.1.3)
from server (10.0.1.2,9999): Hello, client (10.0.1.5)! This i
s server (10.0.1.3)
from server (10.0.1.2,9999): Hello, client (10.0.1.5)! This i
s server (10.0.1.3)
from server (10.0.1.2,9999): Hello, client (10.0.1.5)! This i
s server (10.0.1.3)
from server (10.0.1.2,9999): Hello, client (10.0.1.5)! This i
s server (10.0.1.3)
from server (10.0.1.2,9999): Hello, client (10.0.1.5)! This i
s server (10.0.1.3)
from server (10.0.1.2,9999): Hello, client (10.0.1.5)! This i
s server (10.0.1.3)
from server (10.0.1.2,9999): Hello, client (10.0.1.5)! This i
s server (10.0.1.3)
```

```
root@can201-VirtualBox:/home/can201# python3 server.py
TCP server bind on 9999...
```

```
from client (10.0.1.5,40056): seq=345 Hello, server (10.0.1.2)
from client (10.0.1.5,40056): seq=346 Hello, server (10.0.1.2)
from client (10.0.1.5,40056): seq=347 Hello, server (10.0.1.2)
from client (10.0.1.5,40056): seq=348 Hello, server (10.0.1.2)
from client (10.0.1.5,40056): seq=349 Hello, server (10.0.1.2)
from client (10.0.1.5,40056): seq=350 Hello, server (10.0.1.2)
from client (10.0.1.5,40056): seq=351 Hello, server (10.0.1.2)
from client (10.0.1.5,40056): seq=352 Hello, server (10.0.1.2)
from client (10.0.1.5,40056): seq=353 Hello, server (10.0.1.2)
from client (10.0.1.5,40056): seq=354 Hello, server (10.0.1.2)
from client (10.0.1.5,40056): seq=355 Hello, server (10.0.1.2)
from client (10.0.1.5,40056): seq=356 Hello, server (10.0.1.2)
from client (10.0.1.5,40056): seq=357 Hello, server (10.0.1.2)
from client (10.0.1.5,40056): seq=358 Hello, server (10.0.1.2)
from client (10.0.1.5,40056): seq=359 Hello, server (10.0.1.2)
from client (10.0.1.5,40056): seq=360 Hello, server (10.0.1.2)
from client (10.0.1.5,40056): seq=361 Hello, server (10.0.1.2)
from client (10.0.1.5,40056): seq=362 Hello, server (10.0.1.2)
```


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

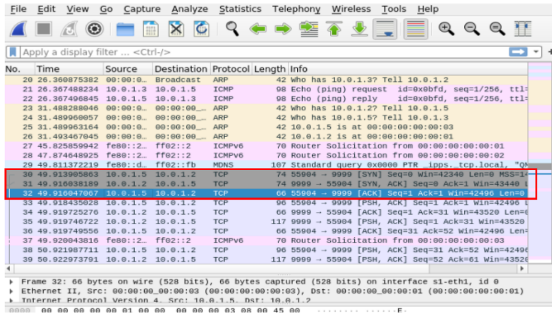
```
No.  Time          Source       Destination  Protocol Length Info
  20 36.771114419  00:00:0..    00:00:00_..  ARP        42 Who has 10.0.1.5? Tell 10.0.1.3
  21 36.771117530  00:00:0..    00:00:00_..  ARP        42 10.0.1.5 is at 00:00:00:00:00:03
  22 36.773076215  00:00:0..    00:00:00_..  ARP        42 10.0.1.2 is at 00:00:00:00:00:01
  23 41.374611548  fe80::2..    ff02::2      ICMPv6     70 Router Solicitation from 00:00:00:00:00:02
  24 48.008531607  fe80::c..    ff02::fb     MDNS      107 Standard query 0x0000 PTR _ipps._tcp.local, "QM
  25 53.659469289  fe80::c..    ff02::2      ICMPv6     70 Router Solicitation from 0e:2c:f2:a1:ec:14
  26 53.659481467  fe80::2..    ff02::2      ICMPv6     70 Router Solicitation from 00:00:00:00:00:03
  27 53.663490881  fe80::2..    ff02::2      ICMPv6     70 Router Solicitation from 00:00:00:00:00:01
  28 59.316326634  10.0.1.5     10.0.1.2     TCP        74 41118 → 9999 [SYN] Seq=0 Win=42340 Len=0 MSS=14
  29 59.318487656  10.0.1.2     10.0.1.5     TCP        74 9999 → 41118 [SYN, ACK] Seq=0 Ack=1 Win=43440 L
  30 59.318498556  10.0.1.5     10.0.1.2     TCP        66 41118 → 9999 [ACK] Seq=1 Ack=1 Win=42496 Len=0
  31 59.318823362  10.0.1.5     10.0.1.2     TCP        96 41118 → 9999 [PSH, ACK] Seq=1 Ack=1 Win=42496 L
  32 59.320088506  10.0.1.2     10.0.1.5     TCP        66 9999 → 41118 [ACK] Seq=1 Ack=31 Win=43520 Len=0
  33 59.320105186  10.0.1.2     10.0.1.5     TCP       117 9999 → 41118 [PSH, ACK] Seq=1 Ack=31 Win=43520
  34 59.320107483  10.0.1.5     10.0.1.2     TCP        66 41118 → 9999 [ACK] Seq=31 Ack=52 Win=42496 Len=
  35 60.321850608  10.0.1.5     10.0.1.2     TCP        96 41118 → 9999 [PSH, ACK] Seq=31 Ack=52 Win=42496
  36 60.322263479  10.0.1.2     10.0.1.5     TCP       117 9999 → 41118 [PSH, ACK] Seq=52 Ack=61 Win=43520
  37 60.322285169  10.0.1.5     10.0.1.2     TCP        66 41118 → 9999 [ACK] Seq=61 Ack=103 Win=42496 Len
  38 61.324933845  10.0.1.5     10.0.1.2     TCP        96 41118 → 9999 [PSH, ACK] Seq=61 Ack=103 Win=4249
  39 61.325104559  10.0.1.2     10.0.1.5     TCP       117 9999 → 41118 [PSH, ACK] Seq=103 Ack=91 Win=4352

▸ Frame 30: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface s1-eth1, id 0
▸ Ethernet II, Src: 00:00:00_00:00:03 (00:00:00:00:00:03), Dst: 00:00:00_00:00:01 (00:00:00:00:00:01)
▸ Internet Protocol Version 4, Src: 10.0.1.5, Dst: 10.0.1.2

0000  00 00 00 00 00 01 00 00  00 00 00 03 08 00 45 00   ........ ......E.
0010  00 34 9d 36 40 00 40 06  87 87 0a 00 01 05 0a 00   .4.6@.@. ........
0020  01 02 a0 9e 27 0f bb 6d  4d 3a 42 59 08 bb 80 10   ....'..m M:BY....
0030  00 53 16 2d 00 00 01 01  08 0a 8d bf 4b 2b 6a 50   .S.-.... ....K+jP
```

C. *testing results*

The red box in the figure below shows the TCP three-way handshake packets in Wireshark:


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.  Time          Source       Destination  Protocol Length Info
  20 26.360875382  00:00:0..    Broadcast    ARP        42 Who has 10.0.1.3? Tell 10.0.1.2
  21 26.367488234  10.0.1.3     10.0.1.5     ICMP       98 Echo (ping) request  id=0x0bfd, seq=1/256, ttl=
  22 26.367496845  10.0.1.5     10.0.1.3     ICMP       98 Echo (ping) reply    id=0x0bfd, seq=1/256, ttl=
  23 31.488288046  00:00:0..    00:00:00_..  ARP        42 Who has 10.0.1.2? Tell 10.0.1.5
  24 31.489960057  00:00:0..    00:00:00_..  ARP        42 Who has 10.0.1.5? Tell 10.0.1.3
  25 31.489963164  00:00:0..    00:00:00_..  ARP        42 10.0.1.5 is at 00:00:00:00:00:03
  26 31.493467045  00:00:0..    00:00:00_..  ARP        42 10.0.1.2 is at 00:00:00:00:00:01
  27 45.825859042  fe80::2..    ff02::2      ICMPv6     70 Router Solicitation from 00:00:00:00:00:01
  28 47.874648925  fe80::2..    ff02::2      ICMPv6     70 Router Solicitation from 00:00:00:00:00:02
  29 49.811372219  fe80::d..    ff02::fb     MDNS      107 Standard query 0x0000 PTR _ipps._tcp.local, "QM
  30 49.913905863  10.0.1.5     10.0.1.2     TCP        74 55904 → 9999 [SYN] Seq=0 Win=42340 Len=0 MSS=14
  31 49.916038189  10.0.1.2     10.0.1.5     TCP        74 9999 → 55904 [SYN, ACK] Seq=0 Ack=1 Win=43440 L
  32 49.916047067  10.0.1.5     10.0.1.2     TCP        66 55904 → 9999 [ACK] Seq=1 Ack=1 Win=42496 Len=0
  33 49.918435028  10.0.1.5     10.0.1.2     TCP        96 55904 → 9999 [PSH, ACK] Seq=1 Ack=1 Win=42496 L
  34 49.919725276  10.0.1.2     10.0.1.5     TCP        66 9999 → 55904 [ACK] Seq=1 Ack=31 Win=43520 Len=0
  35 49.919746722  10.0.1.2     10.0.1.5     TCP       117 9999 → 55904 [PSH, ACK] Seq=1 Ack=31 Win=43520
  36 49.919749556  10.0.1.5     10.0.1.2     TCP        66 55904 → 9999 [ACK] Seq=31 Ack=52 Win=42496 Len=
  37 49.920043816  fe80::2..    ff02::2      ICMPv6     70 Router Solicitation from 00:00:00:00:00:03
  38 50.921987711  10.0.1.5     10.0.1.2     TCP        96 55904 → 9999 [PSH, ACK] Seq=31 Ack=52 Win=42496
  39 50.922973791  10.0.1.2     10.0.1.5     TCP       117 9999 → 55904 [PSH, ACK] Seq=52 Ack=61 Win=43520

▸ Frame 32: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface s1-eth1, id 0
▸ Ethernet II, Src: 00:00:00_00:00:03 (00:00:00:00:00:03), Dst: 00:00:00_00:00:01 (00:00:00:00:00:01)
▸ Internet Protocol Version 4, Src: 10.0.1.5, Dst: 10.0.1.2

0000  00 00 00 00 00 01 00 00  00 00 00 03 08 00 45 00   ........ ......E.
```

It is clear from this figure that the networking latency for completing a TCP three-way handshake is 49.916047067-49.913905863=0.002141204 seconds.

Therefore, we tested the forwarding and redirection functions five times respectively and drew a line graph based on their networking latency.

- The results for forwarding are 0.002141204, 0.003934636, 0.001722118, 0.002896580, 0.004296568 (s).
- The results for redirection are 0.003851649, 0.002171922, 0.003790239, 0.002360271, 0.003252459 (s).

Fig4 shows the line chart of the networking latency for forwarding and redirection.

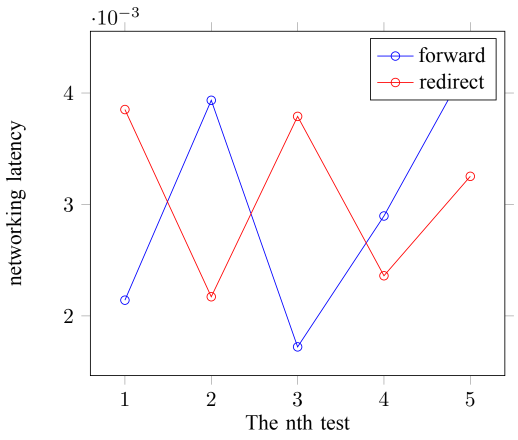


Fig. 4. Networking latency

The average networking latency for forwarding is 0.00299822 seconds.

The average networking latency for redirection is 0.00308531 seconds.

VI. Conclusion

In this project, an SDN network topology was programmed based on Mininet. The Ryu framework was implemented to construct two controller applications to forward and redirect the traffic between hosts based on OpenFlow protocols. The networking latency was also tested for forwarding and redirection by Wireshark.

Because of the unstable network latency for forwarding and redirection from testing, future work will focus on minimizing the networking latency by changing the controller placements [7] or alleviating network congestion by using a traffic redirection to divert traffic load from congested paths [8]. It is also important to focus on the security of SDN. Based on the complexity of SDN, it may allow an attacker to operate control or blind the defender in any way. In addition, the unique flexibility of SDN may bring additional complexity, which may magnify the conflict between network objectives and security requirements. Therefore, it is necessary to address these issues to improve network security.

Acknowledgment

Shuchen Ji (2034172) contributes 42% to the project, Xinxin Fan (2039125) contributes 42% to the project, and Shiqi Huang (2034174) contributes 16% to the project.

References

[1] M. Erel, E. Teoman, Y. Özçevik, G. Seçinti, and B. Canberk, "Scalability analysis and flow admission control in mininet-based sdn environment," in *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, 2015, pp. 18–19.

[2] H. Tao, J. M. Zain, S. B. Band, B. Sundaravadivazhagan, A. Mohamed, H. A. Marhoon, O. O. Ogbonnia, and P. Young, "Sdn-assisted technique for traffic control and information execution in vehicular adhoc networks." *Computers and Electrical Engineering*, vol. 102, 2022.

[3] B. Kurungadan and A. Abdrabou, "Using software-defined networking for data traffic control in smart cities with wifi coverage †." *Symmetry (20738994)*, vol. 14, no. 10, p. N.PAG, 2022.

[4] H. I-Shyan and A. Liem, "A multimedia services architecture supporting local traffic redirection in passive optical network." *The 2012 11th International Conference on Optical Communications and Networks (ICOCN), Optical Communications and Networks (ICOCN), 2012 11th International Conference on*, pp. 1 – 4, 2012.

[5] M. Z. Shaikh and S. H. Darekar, "Performance analysis of various open flow controllers by performing scalability experiment on software defined networks," in *2018 3rd International Conference on Inventive Computation Technologies (ICICT)*, 2018, pp. 783–787.

[6] T. Hamano, R. Suzuki, T. Ikegawa, and H. Ichikawa, "A redirection-based defense mechanism against flood-type attacks in large-scale isp networks." *APCC/MDMC '04. The 2004 Joint Conference of the 10th Asia-Pacific Conference on Communications and the 5th International Symposium on Multi-Dimensional Mobile Communications Proceeding, Communications, 2004 and the 5th International Symposium on Multi-Dimensional Mobile Communications Proceedings. The 2004 Joint Conference of the 10th Asia-Pacific Conference on, Communications and multi-dimensional mobile communications*, vol. 2, p. 543, 2004.

[7] G. Ramya and R. Manoharan, "Enhanced optimal placements of multi-controllers in sdn." *Journal of Ambient Intelligence and Humanized Computing*, vol. 12, no. 7, pp. 8187 – 8204, 2021.

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