CAN201 COURSEWORK 2 REPORT

1st Jing YIN
ID number:2034355

2nd Yumeng Zhang
ID number:2035352

3rd Janis Anerauds
ID number:2032936

***Abstract*—This project requires us to create a simple SDN network topology using Mininet and perform adaptive traffic engineering by using the capabilities of SDN remote controllers and real-time traffic manipulation. Many mature SDN network topologies have been implemented in the market. Thanks to the SDN network architecture, business automation and rapid innovation are realized, the network is simplified, and network path traffic is also optimized. Referring to the mature SDN network architecture, we designed our own products. We first use a virtual machine to build an SDN network system, and then create two controller applications to implement forwarding and redirection respectively. The forwarding function enables the client to send information to server1. The redirection function implements the forwarding of information sent by the client to server1 to server2. It also implements the simulated flow control function using SDN flow entries. During the test, we first use the client to send information of the same data size, measure the delay time of forward and redirect multiple times, analyze the randomness and stability of the data, and then test the delay of forward and redirect when the client sends different data sizes time. The test environment is defined as a Linux virtual machine. The test results are quite good, we successfully used Mininet to create a simple SDN network topology and realized the simulated traffic control function, so we plan to develop a more powerful product for the future.**

I. Introduction

A. *project task specification*

SDN is a reconstruction of the traditional network architecture. It is reconstructed from the original distributed control network architecture to a centralized control network architecture, which breaks the traditional network's insufficient ability to flexibly adjust traffic paths, complex network protocol implementation, and difficulty in operation and maintenance. The limitations of large size and slow upgrade speed of new network services. In this network project, Mininet is used to create a simple SDN network topology, and SDN flow entries are used to simulate flow control functions.

B. *challenge*

Many difficulties were encountered during the development process. For example, the mastery of the interaction process in the SDN network system and the realization of the redirection function.

C. *practice relevance*

The results of the project also have many potential applications. Big data center management, mobile network service solution, basically any use case with massive amounts of data, scalability options and large needs to monitor precisely and split the data or transfer data on packet level.

D. *contributions (key points that you did for this coursework)*

We mainly use virtual machines and mininet to build a virtual SDN network system. And use ryu to compile two controller applications to grasp the global information of the network, and install the switch flow table to realize the forwarding and redirection functions. And through the testing process, we got the conclusion that forwarding performance is higher than redirection performance.

II. Related Work

Classical IP networks based on physical IP routers use a distributed routing mechanism to forward traffic. Typically, the router selects the output interface by finding the shortest path to the target node. The shortest path algorithm uses static metrics to estimate the cost of a path. One of the widely used path measures is the number of hops a packet must make to its destination; The other is the sum of the administrative weights of the links that make up the path. But the most popular shortest path approach, based on simple static metrics, often leads to inefficient use of network resources.

It is also worth mentioning that static shortest path routing causes problems in the event of a failure – the router looks for a new shortest path as a candidate for forwarding traffic, and when such a path is found, the total traffic from the interrupted path moves to the new path. This behavior can lead to congestion of the newly chosen path while other existing paths between the source-destination pairs of interest remain underutilized. This example shows the benefits of allocating traffic between multiple paths. There are two main engineering approaches, namely MPLS-TE [1] and OSPF-ECMP [2], designed to address the above shortage. But MPLS-TE was considered too complex and slow a solution to dynamically reconfigure paths. The ECMP (equal-cost multipath) extension of the OSPF routing protocol also has some limitations. For example, it cannot be used to divert traffic between parallel paths of different capacities. In general, ECMP does not support configurable load ratios between equal-cost paths, which can lead to congestion on paths carrying more traffic [3].

SDN is accompanied by a fundamental change in network architecture based on the separation between the control plane (physical or logical centralized unit – SDN controller) and the forwarding plane (switch) (ONF, Software-Defined Networking: New Specification for Networking, 2012) [4]. This concept opens up more possibilities for solving the above network traffic engineering problems. Today's SDN network is mainly based on the OpenFlow protocol [5], which is the communication language between controllers and switches. Such a network architecture provides several benefits:

1) Compared with the existing distributed routing protocol, the centralized control plane provides a shorter path setting time [6].
2) By limiting the number of control plane devices and simplifying the data plane device architecture, the complexity and cost of the network can be reduced. In addition, network management is less complex in this case.
3) The centralized controller becomes the brain of the whole network. Using the global network view, it can effectively decide where to forward the flow to meet the application requirements, while optimizing the resource utilization of the forwarding plane. In addition, routing and TE mechanisms need not be implemented in all nes – centralized traffic engineering can be implemented as a function of the controller alone.
4) SDN is a more powerful general-purpose network tool because it allows for deeper network management and monitoring, and it allows for better network automation. Traffic engineering works well in SDN because the network topology and link state are known to the SDN controller; As a result, all existing paths and their current loads can be easily identified, and more than one route can be used for business forwarding between any source-destination pair. The controller can change the configuration of the flow switching table (FST) on all switches in real time to properly redirect the flow. Of course, SDN can not only implement forwarding and redirection, but also include each packet management and data analysis to provide the best network solution for the larger use case.

III. Design

A. network system design

This section aims to design a simple SDN network system and two controller applications, respectively, to achieve forwarding and redirection. The network system design is shown in Figure 1. In the forwarding plane, there are three hosts, one running the client program as a client and the other two running the server program as a server. It is worth noting that the client here only knows server 1. The three hosts are provided with a data forwarding function by an SDN switch. The switch contains a flow table. When the switch receives a packet, it forwards the packet to the host based on the longest prefix-matching principle. When the switch encounters a packet that it does not know how to process, it sends the packet to a logically centralized remote controller over a secure channel. The controller understands the global network information and can install flow entries for the switch. The controller uses the southbound API and follows the OpenFlow protocol to communicate with the switch. The controller also connects to the network control application through northbound API.

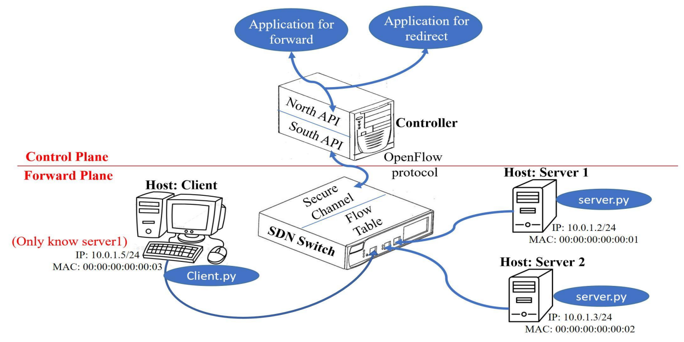


Fig. 1. Network System Design diagram

B. controller application design

The general architecture of the two controller applications is shown in Figure 2. First of all, we will explain the application of the controller, which realize the forwarding function. The controller needs to declare the protocol version and then create a mac address table containing mac addresses and corresponding port information. After the controller and the switch finish the handshake, the controller needs to install a table miss entry in the switch to inform the switch that it can directly send the data packet to the controller when it cannot process it. The “Switch handler” in the figure is used to do this. In the figure, “add flow” works with the switch handler to install the default entry. In addition, when the controller receives “packet in” (packets that cannot process) from the switch, it needs a “packet in handler” to implement the forwarding function. In the figure, ‘add flow1’ works with “packet in handler” to install new entries. The controller application that implements the redirection function is very similar so we won’t go into the details here.

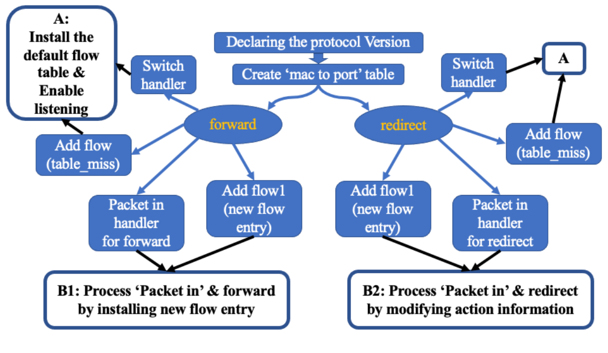

A: Install the default flow table & Enable listening
Switch handler
Declaring the protocol Version
Create ‘mac to port’ table
Switch handler
A
forward
redirect
Add flow (table_miss)
Add flow (table_miss)
Packet in handler for forward
Add flow1 (new flow entry)
Add flow1 (new flow entry)
Packet in handler for redirect
B1: Process ‘Packet in’ & forward by installing new flow entry
B2: Process ‘Packet in’ & redirect by modifying action information

Fig. 2. Controller application design diagram

The diagram 3 shows the workflow and kernel algorithm of the switch handler for the two controller applications. When creating an entry, we set the priority of the entry to 0, the matching field to full matching (that is, nothing is matched), and the action to ”Send the packet to the controller”.

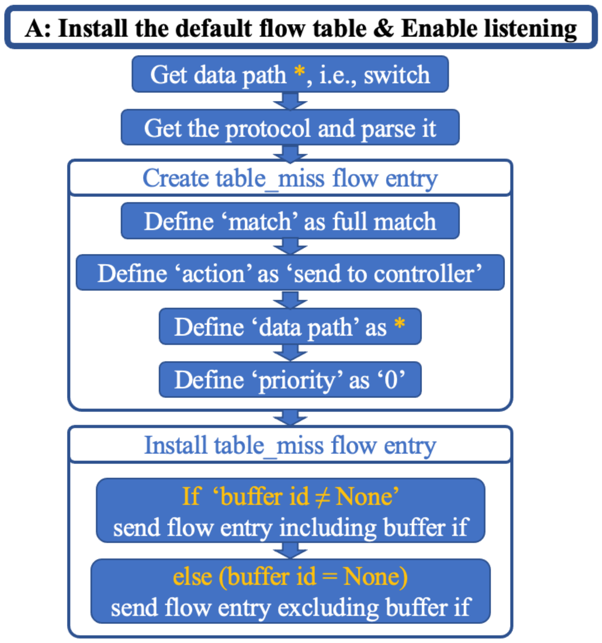

A: Install the default flow table & Enable listening
Get data path *, i.e., switch
Get the protocol and parse it
Create table_miss flow entry
Define 'match' as full match
Define 'action' as 'send to controller'
Define 'data path' as *
Define 'priority' as '0'
Install table_miss flow entry
If 'buffer id ≠ None'
send flow entry including buffer if
else (buffer id = None)
send flow entry excluding buffer if

Fig. 3. Switch handler design diagram

The figure 4 shows the workflow and kernel algorithm of packet in handler applied for the controller with forwarding function. Note that when creating an entry, we set the priority to 1.

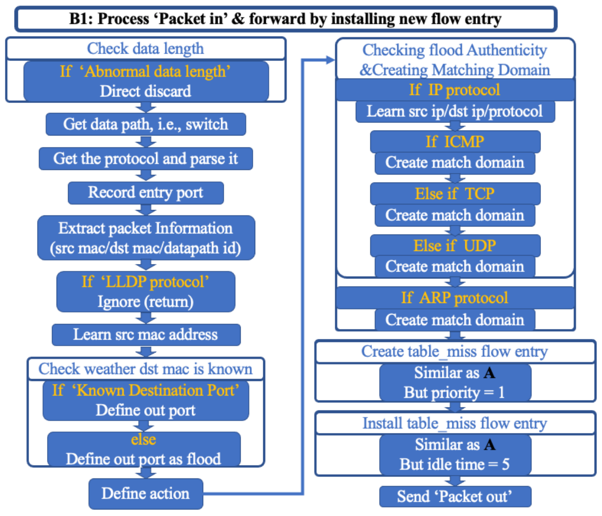

B1: Process 'Packet in' & forward by installing new flow entry
Check data length
If 'Abnormal data length'
Direct discard
Get data path, i.e., switch
Get the protocol and parse it
Record entry port
Extract packet Information
(src mac/dst mac/datapath id)
If 'LLDP protocol'
Ignore (return)
Learn src mac address
Check weather dst mac is known
If 'Known Destination Port'
Define out port
else
Define out port as flood
Define action
Checking flood Authenticity
&Creating Matching Domain
If IP protocol
Learn src ip/dst ip/protocol
If ICMP
Create match domain
Else if TCP
Create match domain
Else if UDP
Create match domain
If ARP protocol
Create match domain
Create table_miss flow entry
Similar as A
But priority = 1
Install table_miss flow entry
Similar as A
But idle time = 5
Send 'Packet out'

Fig. 4. 'Packet in handler' for forwarding design diagram

The packet in handler of the controller application that implements the redirection function is very similar to figure 4. Figure 5 shows the application workflow and kernel algorithm when "Checking flood Authenticity & Creating Matching". We implement redirection by modifying the action information. Note that server2 must disguise itself as server1 when sending response information to ensure that the client can receive response message successfully.

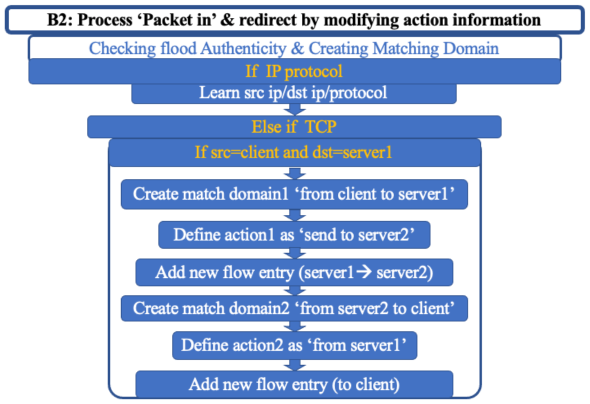

B2: Process 'Packet in' & redirect by modifying action information
Checking flood Authenticity & Creating Matching Domain
If IP protocol
Learn src ip/dst ip/protocol
Else if TCP
If src=client and dst=server1
Create match domain1 'from client to server1'
Define action1 as 'send to server2'
Add new flow entry (server1→ server2)
Create match domain2 'from server2 to client'
Define action2 as 'from server1'
Add new flow entry (to client)

Fig. 5. 'Packet in handler' for redirection design diagram

IV. Implementation

A. Implementation steps

The host operating system we used for implementation is macOS, the CPU is 2.3 GHz Quad-Core Intel Core i7, and the memory is 16 GB. In this host environment, we use the virtual box to create ubuntu virtual environment to achieve all compilation. The implementation steps are mainly divided into three parts.

Firstly, we use the terminal to create a python file named "networkTopo". Python mininet library is used to build the network architecture shown in figure 1. First create a network with IP base 10.0.1.0/24. Then three hosts, a switch and a controller were added. The controller type is remote controller. Then set the IP and MAC addresses for the hosts. Finally, make a connection using the makeTerm library.

Secondly, we create a python file named "ryu_forward" with the terminal and use the python ryu library to compile the controller application according to the design drawing in 3.2 to realize the forward function. The SDN controller software used here is Ryu. First declare the protocol version, in this case OpenFlow1.3. Then create a method to define the mac address table and another method that delivers default entry and enables listening. The ryu controller and switch are connected in four stages, namely HANDSHAKE_DISPATCHER, CONFIG_DISPATCHER, MAIN_DISPATCHER and DEAD_DISPATCHER. After a successful handshake and going to CONFIG_DISPATCHER phase, the controller automatically sends feature request messages. After receiving the feature request message, the switch replies with the message, which is EventOFPSwitchFeatures. This method is called when the controller receives the EventOFPSwitchFeatures and has not yet transitioned to the state of MAIN_DISPATCHER. So here we first call "set_ev_cls" to specify the event category. To deliver a flow table, create another methond "add_flow". Finally, create a methond to implement the forwarding function. Call "set_ev_cls" to specify MAIN_DISPATCHER as the event category. The action of sending flow entries also requires another methond 'add_flow1' to implement. "add_flow1" differs from "add_flow" in that it sets the idle timeout to 5 seconds.

Thirdly, use the terminal to create a python file named "ryu_forward", and use the python ryu library to compile the controller application according to the design diagram in 3.2 to achieve the redirection function. The compilation method for this application is very similar to the one above. The redirection is mainly achieved by modifying the actions in the flow table.

B. program flow chart

The flow chart of network system construction, remote controller operation, server program operation, client program operation, forwarding and redirection is shown in Figure 6. Among them, Controller A and Switch A implement configuration default entry. Server A opens the server program. Client A sends a packet. Switch B and Controller B implement the forwarding function. Switch B and Controller C implement the redirection function. The process of forwarding the response message sent by the server is very similar, so some details have been omitted from the figure.

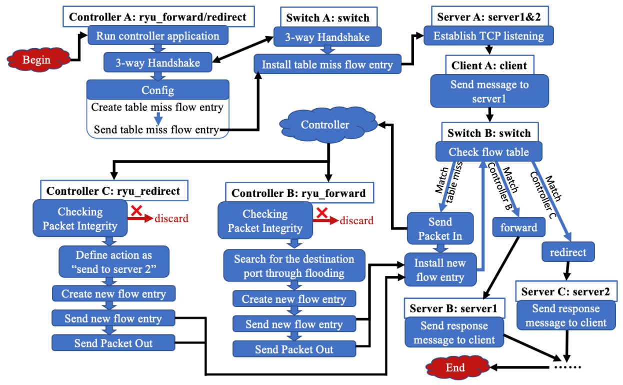


Fig. 6.

C. Programming skills and difficulties

During the implementation, we used OOP programming techniques, and we created a lot of classes and nesting. Each class has its own intelligence, such as implementing forwarding and redirection. We also documented the implementation process. This is very important in group work, which helps non-programmers understand code quickly.

In the process of implementation, we have encountered some difficulties. The biggest difficulty is to grasp the interaction process of the network system. Only by sorting out the interactive flow can we insert relevant applications at correct nodes in the flow to achieve the purpose of forwarding and redirection. Finally, this problem was solved by drawing a flowchart. Another difficulty is that after implementing the redirection from client to server2, the client cannot receive the return information from server2. Later, through group discussions, we worked together to solve the problem.

V. Testing and Results

A. Test environment

The test environment is the same as the implementation environment.

B. Testing steps

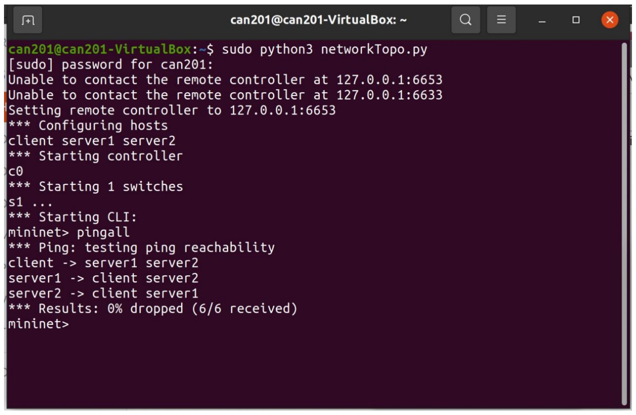
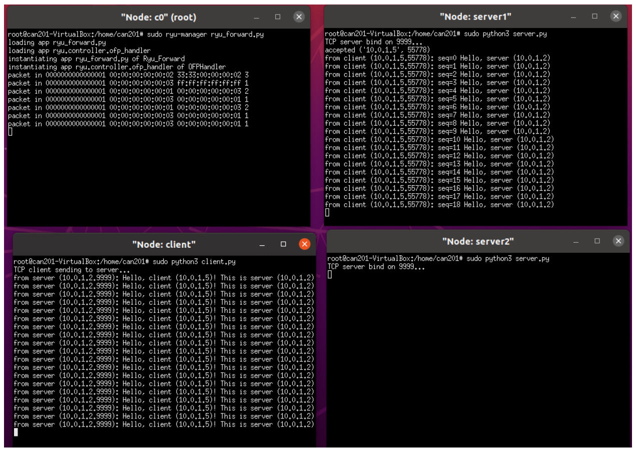

```
can201@can201-VirtualBox: ~
can201@can201-VirtualBox:~$ sudo python3 networkTopo.py
[sudo] password for can201:
Unable to contact the remote controller at 127.0.0.1:6653
Unable to contact the remote controller at 127.0.0.1:6633
Setting remote controller to 127.0.0.1:6653
*** Configuring hosts
client server1 server2
*** Starting controller
c0
*** 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>
```

Fig. 7. Pingall test

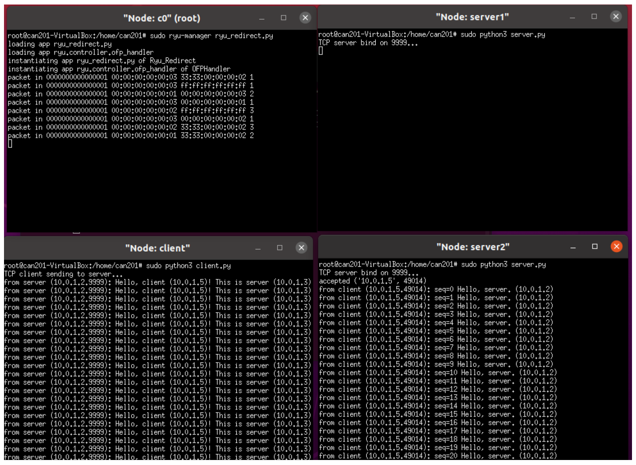

```
"Node: c0" (root)
root@can201-VirtualBox:/home/can201# sudo ryu-manager ryu_forward.py
loading app ryu_forward.py
loading app ryu.controller.ofp_handler
instantiating app ryu_forward.py of Ryu_Forward
instantiating app ryu.controller.ofp_handler of OFPHandler
packet in 0000000000000001 00:00:00:00:00:02 33:33:00:00:00:02 3
packet in 0000000000000001 00:00:00:00:00:03 ff:ff:ff:ff:ff:ff 1
packet in 0000000000000001 00:00:00:00:00:01 00:00:00:00:00:03 2
packet in 0000000000000001 00:00:00:00:00:03 00:00:00:00:00:01 1
packet in 0000000000000001 00:00:00:00:00:01 00:00:00:00:00:03 2
packet in 0000000000000001 00:00:00:00:00:03 00:00:00:00:00:01 1
packet in 0000000000000001 00:00:00:00:00:03 00:00:00:00:00:01 1
```

```
"Node: server1"
root@can201-VirtualBox:/home/can201# sudo python3 server.py
TCP server bind on 9999...
accepted ('10.0.1.5', 55778)
from client (10.0.1.5.55778): seq=0 Hello, server (10.0.1.2)
from client (10.0.1.5.55778): seq=1 Hello, server (10.0.1.2)
from client (10.0.1.5.55778): seq=2 Hello, server (10.0.1.2)
from client (10.0.1.5.55778): seq=3 Hello, server (10.0.1.2)
from client (10.0.1.5.55778): seq=4 Hello, server (10.0.1.2)
from client (10.0.1.5.55778): seq=5 Hello, server (10.0.1.2)
from client (10.0.1.5.55778): seq=6 Hello, server (10.0.1.2)
from client (10.0.1.5.55778): seq=7 Hello, server (10.0.1.2)
from client (10.0.1.5.55778): seq=8 Hello, server (10.0.1.2)
from client (10.0.1.5.55778): seq=9 Hello, server (10.0.1.2)
from client (10.0.1.5.55778): seq=10 Hello, server (10.0.1.2)
from client (10.0.1.5.55778): seq=11 Hello, server (10.0.1.2)
from client (10.0.1.5.55778): seq=12 Hello, server (10.0.1.2)
from client (10.0.1.5.55778): seq=13 Hello, server (10.0.1.2)
from client (10.0.1.5.55778): seq=14 Hello, server (10.0.1.2)
from client (10.0.1.5.55778): seq=15 Hello, server (10.0.1.2)
from client (10.0.1.5.55778): seq=16 Hello, server (10.0.1.2)
from client (10.0.1.5.55778): seq=17 Hello, server (10.0.1.2)
from client (10.0.1.5.55778): seq=18 Hello, server (10.0.1.2)
```

```
"Node: client"
root@can201-VirtualBox:/home/can201# sudo python3 client.py
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)
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)
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: server2"
root@can201-VirtualBox:/home/can201# sudo python3 server.py
TCP server bind on 9999...
```

Fig. 8. Forwarding case test


```
"Node: c0" (root)
root@can201-VirtualBox:/home/can201# sudo ryu-manager ryu_redirect.py
loading app ryu_redirect.py
loading app ryu.controller.ofp_handler
instantiating app ryu_redirect.py of Ryu_Redirect
instantiating app ryu.controller.ofp_handler of OFPHandler
packet in 0000000000000001 00:00:00:00:00:03 33:33:00:00:00:02 1
packet in 0000000000000001 00:00:00:00:00:03 ff:ff:ff:ff:ff:ff 1
packet in 0000000000000001 00:00:00:00:00:01 00:00:00:00:00:03 2
packet in 0000000000000001 00:00:00:00:00:03 00:00:00:00:00:01 1
packet in 0000000000000001 00:00:00:00:00:02 ff:ff:ff:ff:ff:ff 3
packet in 0000000000000001 00:00:00:00:00:03 00:00:00:00:00:02 1
packet in 0000000000000001 00:00:00:00:00:02 33:33:00:00:00:02 3
packet in 0000000000000001 00:00:00:00:00:01 33:33:00:00:00:02 2
```

```
"Node: server1"
root@can201-VirtualBox:/home/can201# sudo python3 server.py
TCP server bind on 9999...
```

```
"Node: client"
root@can201-VirtualBox:/home/can201# sudo python3 client.py
TCP client sending to server...
from server (10.0.1.2.9999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.9999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.9999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.9999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.9999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.9999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.9999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.9999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.9999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.9999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.9999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.9999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.9999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.9999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.9999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.9999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.9999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.9999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.9999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.9999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.9999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
from server (10.0.1.2.9999): Hello, client (10.0.1.5)! This is server (10.0.1.3)
```

```
"Node: server2"
root@can201-VirtualBox:/home/can201# sudo python3 server.py
TCP server bind on 9999...
accepted ('10.0.1.5', 49014)
from client (10.0.1.5.49014): seq=0 Hello, server. (10.0.1.2)
from client (10.0.1.5.49014): seq=1 Hello, server. (10.0.1.2)
from client (10.0.1.5.49014): seq=2 Hello, server. (10.0.1.2)
from client (10.0.1.5.49014): seq=3 Hello, server. (10.0.1.2)
from client (10.0.1.5.49014): seq=4 Hello, server. (10.0.1.2)
from client (10.0.1.5.49014): seq=5 Hello, server. (10.0.1.2)
from client (10.0.1.5.49014): seq=6 Hello, server. (10.0.1.2)
from client (10.0.1.5.49014): seq=7 Hello, server. (10.0.1.2)
from client (10.0.1.5.49014): seq=8 Hello, server. (10.0.1.2)
from client (10.0.1.5.49014): seq=9 Hello, server. (10.0.1.2)
from client (10.0.1.5.49014): seq=10 Hello, server. (10.0.1.2)
from client (10.0.1.5.49014): seq=11 Hello, server. (10.0.1.2)
from client (10.0.1.5.49014): seq=12 Hello, server. (10.0.1.2)
from client (10.0.1.5.49014): seq=13 Hello, server. (10.0.1.2)
from client (10.0.1.5.49014): seq=14 Hello, server. (10.0.1.2)
from client (10.0.1.5.49014): seq=15 Hello, server. (10.0.1.2)
from client (10.0.1.5.49014): seq=16 Hello, server. (10.0.1.2)
from client (10.0.1.5.49014): seq=17 Hello, server. (10.0.1.2)
from client (10.0.1.5.49014): seq=18 Hello, server. (10.0.1.2)
from client (10.0.1.5.49014): seq=19 Hello, server. (10.0.1.2)
from client (10.0.1.5.49014): seq=20 Hello, server. (10.0.1.2)
```

Fig. 9. Redirection case test

C. Testing results

1. In this test, since the delay time may be affected by factors other than the variable "byte", we measured the same data several times to illustrate the randomness and stability of the measured data.

When the sent data was 14bytes, 5 groups of data were tested by keeping other variables consistent, as shown in Figure 11

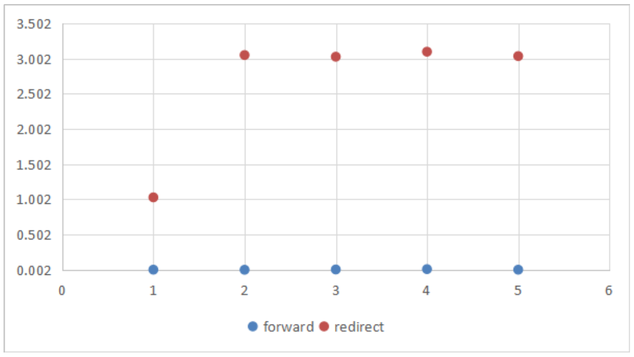


Fig. 10.

As shown in the figure, the randomness and stability of the delay time data measured by the forward are weak, while the randomness and stability of the redirect data are weak. Meanwhile, it can be seen from the scatter diagram that forward has shorter delay time and better performance than redirect.

2. In addition, the length of the message sent by the client maps the segment bytes size, so the latency should be affected by the message length. For this reason, we use Wireshark on Client to capture the packets, Test the latency for both forward and redirect performance by changing the size of the sent data (from the first SYN segment till the last ACK segment indicating the TCP 3-way) handshake is done). It is worth mentioning that as shown in the conclusion of Figure 10, the data has randomness and stability, so some abnormal data has been deleted. The results are shown in Figure 11.

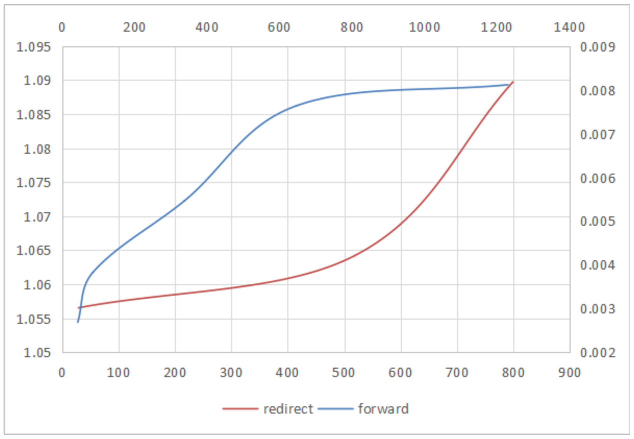


Fig. 11.

(In order to display the picture more aesthetically, we use two axes - the right vertical axis represents the forward delay time, and the left vertical axis represents the redirect delay time)

As shown in the figure, comparing the time under the forward and redirect performance with different bytes, it can be found that under the same performance, the longer the byte, the longer the time, and the byte and the required time show a positive correlation. It is also obvious that forward has shorter latency and better performance than redirect.

VI. Conclusion

what you did for this project and any future work for improvement. In this project, we created a simple SDN network topology and carried out adaptive traffic engineering by using the capabilities of SDN and the capability of real-time traffic manipulation. We first set up a virtual network system. Then two controller applications are compiled to implement forwarding and redirection respectively. in the controller application, we created the switch handler and packet in handler. switch handler Used to deliver the default flow entry and listen to switch events. packet in handler Processes unknown packets. The redirection is implemented by changing the action information. In the future, I will redesign the entire structure of the code. There is some duplicate code that makes the project difficult to modify. In addition, I'll take a few more steps to encrypt the code to provide it with security.

References

[1] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow, "Rfc3209: Rsvp-te: Extensions to rsvp for lsp tunnels," USA, 2001.
[2] K. Németh, A. Kőrösi, and G. Rétvári, "Optimal ospf traffic engineering using legacy equal cost multipath load balancing," in *2013 IFIP Networking Conference*, 2013, pp. 1–9.
[3] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella, "On the impact of packet spraying in data center networks," in *2013 Proceedings IEEE INFOCOM*, 2013, pp. 2130–2138.
[4] N. McKeown, "Software-defined networking [ol]," 2012.
[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, p. 69–74, mar 2008. [Online]. Available: https://doi.org/10.1145/1355734.1355746
[6] K. Matsui, M. Kaneda, and K. Matsuda, "Evaluation of a server-based traffic engineering architecture suitable for large-scale mpls networks," in *8th Asia-Pacific Symposium on Information and Telecommunication Technologies*, 2010, pp. 1–6.