Xi'an Jiaotong-Liverpool University
西交利物浦大学

Xi'an Jiaotong-Liverpool University

School of Advanced Technology

A Simple SDN Network Topology and Traffic Control Applications

Kaijie Lai 2034675

Zhen Ma 2034590

Ruochen Qi 2035507

December 18, 2022

A Simple SDN Network Topology and Traffic Control Applications

Zhen Ma
School of advanced technology
Xi'an jiaotong-Liverpool University
Suzhou, China
Zhen.Ma20@student.xjtlu.edu.cn

Ruochen Qi
School of advanced technology
Xi'an jiaotong-Liverpool University
Suzhou, China
Ruochen.Qi20@student.xjtlu.edu.cn

Kaijie Lai
School of advanced technology
Xi'an jiaotong-Liverpool University
Suzhou, China
Kaijie.Lai20@student.xjtlu.edu.cn

***Abstract*—Software Defined Network is an important network architecture in computer networks and is of great importance in network control. In order to better investigate the process of network configuration, control and management through it, we tried to use it to manipulate traffic without the awareness of the client and finally achieved the goal with high transmission efficiency.**

***Index Terms*—Software defined network (SDN), Mininet, Ryu, traffic operation, Python**

I. Introduction

With the continuous development of the Internet, the traditional structure, i.e., network equipment, operating system and network applications form a closed system and are dependent on each other, tightly coupled structure gradually cannot meet the needs of network development. In this context, Software Defined Network (SDN) has emerged. It allows direct programming and control of the network and has the characteristics of open programmability, separation of the control plane from the data plane, and logical centralized control of the network. For simple use of SDN, Mininet and Ryu are often used for assistance. Both are written in Python, the former is a lightweight network emulator while the latter is an open source SDN controller. In this project, the team will build a simple SDN network topology using the Python-based Mininet, focusing on SDN traffic control, and attempting to complete a total of five tasks as follows:

- Building a simple SDN network topology using the Mininet library.
- Each node is accessible to each other using the Ryu framework SDN controller.
- Applying socket client and server programs to the above SDN network topology.
- Forward traffic from the Client to Server 1 using SDN to Server 1 and calculate network latency.
- Redirect traffic from the Client to Server 1 to Server 2 using SDN and calculate network latency.

The project allows for stable and secure traffic control between multiple devices, facilitating automated and intelligent network management. As a contribution to the project, our group first constructed the SDN network topology, then deployed the socket clients and servers, and finally forwarded and redirected the Client's traffic. The report will first describe the work involved, then the design and implementation of the project, and finally the test results.

II. Related work

The bloat of traditional network systems has long been a concern in academia, and as a result SDN is constantly being researched and published by universities and companies as a representative of new network systems. This has provided us with various reference materials for our project.

In the focus of the task, traffic control, wang [1] provides in his article a way of managing network traffic that can effectively evaluate time through the application of SDN on the IoT. In another article, Hasan [2] provides a method for using Mininet to assist in creating an SDN network topology and controlling the network behavior using SDN, more closely related to the project our group is working on. In addition, Tivig's team used the Ryu framework when building the SDN network topology and analyzed the framework's code in detail [3], providing guidance on the project.

The related work above provides analysis and coding techniques for traffic control using SDN. This facilitated the design, implementation and testing of our entire project.

III. Design

This section will describe the entire process of designing the solution including a SDN network topology architecture and a work-flow of the redirecting function.

A. Network Architecture of Design

Figure 1 illustrates a architecture of a SDN network topology which support both direct forward and redirect. It is consists of one client and two servers, every host has a corresponding port on the switch which can support uploading and downloading at the same time. Then the main data format in switch is a Flow Table that is consists by many entries. Every entry is also called a flow table entry field who have 7 parts which are Match Fields, Priority, Counters, Instruction, Timeouts, Cookies and flags. In file forwarding and redirecting tasks, we need to use some specific data part in match field like IP port, IP source, IP destination, TCP source, TCP destination, etc. If data can be recognize in switch at first, it will be transferred to controller to have a check and make some actions in controllers.

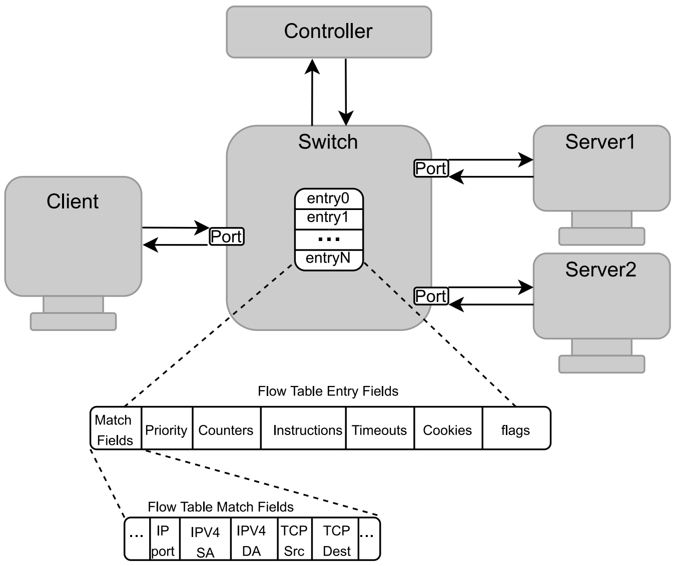


Fig. 1. SDN Network Architecture

This is a complete activity which performs in this architecture and more information of the specific steps in redirection will be explained in the next part.

B. Work-flow Solution of Design

In this section the design of our work-flow solution will be explained in detail. Figure 2 is a Activity Diagram which shows a more detailed explanation of how to Redirect a file to the server from above. Here is a description of the key steps involved:

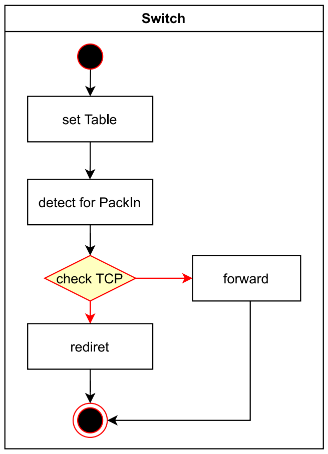


Fig. 2. Workflow in Switch

1) Firstly, after receiving the data from client, data will be set in a format suited for Flow Entry.
2) Then parts which are needed to be detected will be installed the flow entry and extracted as Packet In.
3) After that, it will be taken a TCP check to redirect those compliant messages form source to the correct destination.

C. Algorithm

The main algorithms of redirecting are as follows:

Algorithm 1 Algorithm of Redirecting

```
Input:   Flow Table Data

Switch:
    if $ip\_src = client\_ip$ and $ip\_dst = server\_1\_ip$
        if match $server\_mac$
            $out\_port \leftarrow mac\_address$
        else
            OFPP_FLOOD
        $match \leftarrow$ paser.OFPMATCH()
        $actions \leftarrow$ paser.OFPAActionsSetField()
                     paser.OFPAActionsOutput()
    elseif $ip\_src = server\_2\_ip$ and $ip\_dst = client\_ip$
        if match $client\_mac$
            $out\_port \leftarrow mac\_address$
        else
            OFPP_FLOOD
        $match \leftarrow$ paser.OFPMATCH()
        $actions \leftarrow$ paser.OFPAActionsSetField()
                     paser.OFPAActionsOutput()

Output: Redirect Transfer
```

IV. Implementation

A. Host Environment

The following environment are used for the implementation of this procedure:

TABLE I
HOST IMPLEMENTATION ENVIRONMENT

CPU	AMD Ryzen 9 5900HX
OS	Arch Linux x86_64
Linux Kernel	6.0.8
Memory	16G
Python	3.10.8
IDE	Pycharm 2022.2

B. Programming Skills

1) **Object-Oriented Programming (OOP):** In both forward.py and redirect.py, we define a class SimpleSwitch13(app_manager.RyuApp) to have a better maintainability.
2) **Decorator:** We set a few of Decorator in before methods to add a function without changing the source code of the function being decorated and set some preconditions for accessing and calling functions.

C. Actual Implementation

```
if srcip == client and dstip == server_1:
  if server_2_m in self.mac_to_port[dpid]:
    out_port = self.mac_to_port[dpid][server_2_m]
  else:
    out_port = ofproto.OFPP_FLOOD

  match = parser.OFPMatch(eth_type=ether_types.
  ETH_TYPE_IP, ipv4_src=srcip, ipv4_dst=dstip)

  actions = [parser.OFPActionSetField(eth_dst=
  server_2_m),
           parser.OFPActionSetField(ipv4_dst=
  server_2),
           parser.OFPActionOutput(port=out_port)]

elif srcip == server_2 and dstip == client:
  if client_m in self.mac_to_port[dpid]:
    out_port = self.mac_to_port[dpid][client_m]
  else:
    out_port = ofproto.OFPP_FLOOD

  match = parser.OFPMatch(eth_type=ether_types.
  ETH_TYPE_IP, ipv4_src=srcip, ipv4_dst=dstip)

  actions = [parser.OFPActionSetField(eth_src=
  server_1_m),
           parser.OFPActionSetField(ipv4_src=
  server_1),
           parser.OFPActionOutput(port=out_port)]
```

Listing 1. Actual Implementation of Redirecting

D. Issues and Problems

There are some issues during the initial designing phase, coding and debugging process:

TABLE II

Issue 1	Hosts cannot ping each other during redirection
Cause	Redirected all traffic
Solution	Redirect TCP traffic only
Issue 2	Clients and servers do not receive the correct messages during the redirection
Cause	One-way TCP traffic redirection from client to server only
Solution	Bidirectional TCP traffic redirection between client and server

V. Testing and Results

A series of tests are used to evaluate the performance of the program. The following shows the test environment, the design and implementation of the testing, and the test results.

A. Testing Environment

To ensure the reliability of the test data, all tests in this project were run on two separate physical machines. The following Table III shows the similarities and differences between the software and hardware used on these two machines.

TABLE III

ID	CPU	OS	Linux Kernel	Memory
1	AMD Ryzen 9 5900HX	Arch Linux x86_64	6.0.12	16G
2	Intel i5-7200U	Manjaro Linux x86_64	5.15.65	8G

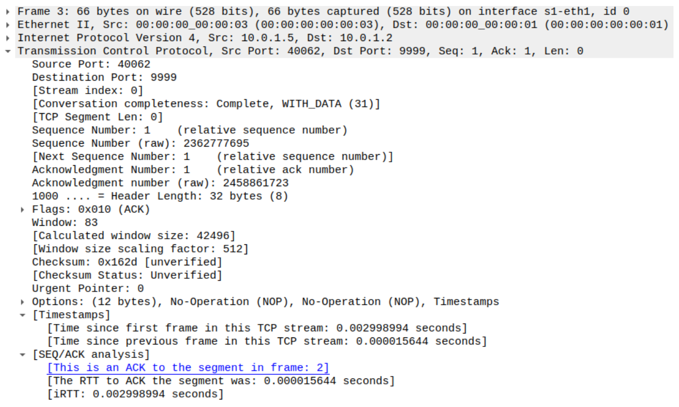
TABLE IV

Python	Ryu	Open vSwitch	Mininet	Wireshark
3.8.15	4.34	3.0.2	2.3.0	4.0.2

B. Testing Design

This project chose to use wireshark as the main testing tool for network traffic monitoring, network protocol analysis, and network latency calculation. The following are the specific preparation steps for the test.

1) Since TCP timestamps are defined in RFC 7323 [4], we can view the time elapsed since the first frame in that TCP stream for each packet arrival in **[Timestamps]** in Wireshark, and then display this time as a new column on Wireshark. You can also view the iRTT values in **[SEQ/ACK analysis]**. The following Figure 3 shows the timestamp and iRTT.



```
▸ Frame 3: 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
▾ Transmission Control Protocol, Src Port: 40062, Dst Port: 9999, Seq: 1, Ack: 1, Len: 0
    Source Port: 40062
    Destination Port: 9999
    [Stream index: 0]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 0]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 2362777695
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 2458861723
    1000 .... = Header Length: 32 bytes (8)
  ▸ Flags: 0x010 (ACK)
    Window: 83
    [Calculated window size: 42496]
    [Window size scaling factor: 512]
    Checksum: 0x162d [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  ▸ Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
  ▾ [Timestamps]
      [Time since first frame in this TCP stream: 0.002998994 seconds]
      [Time since previous frame in this TCP stream: 0.000015644 seconds]
  ▾ [SEQ/ACK analysis]
      [This is an ACK to the segment in frame: 2]
      [The RTT to ACK the segment was: 0.000015644 seconds]
      [iRTT: 0.002998994 seconds]
```

Fig. 3. Preparation Steps for the Test

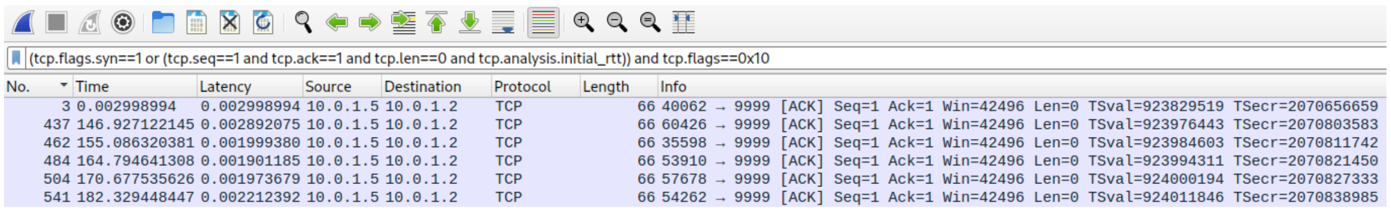
2) In order to get the desired packets, a Wireshark filter was designed for this project:

```
(tcp.flags.syn==1 or
(tcp.seq==1 and tcp.ack==1 and
 tcp.len==0 and tcp.analysis.initial_rtt)) and
 tcp.flags==0x10
```

Listing 2. Wireshark Filter Code

This filter can directly get the last ACK packet in TCP 3-ways handshake, the following Figure 4 shows the filter and its result.

3) This project designed a bash script for the server and client respectively to implement constant TCP connections and disconnections to get enough TCP handshake data more quickly. The following shows the core code of the two bash scripts
4) Use the above steps to get enough data and then export all valid data to a .csv file.
5) Write python scripts to organize and analyze the obtained data.



```
(tcp.flags.syn==1 or (tcp.seq==1 and tcp.ack==1 and tcp.len==0 and tcp.analysis.initial_rtt)) and tcp.flags==0x10
```

```
No.   Time          Latency     Source    Destination Protocol Length Info
  3 0.002998994   0.002998994 10.0.1.5 10.0.1.2    TCP     66 40062 → 9999 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=923829519 TSecr=2070656659
437 146.927122145 0.002892075 10.0.1.5 10.0.1.2    TCP     66 60426 → 9999 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=923976443 TSecr=2070803583
462 155.086320381 0.001999380 10.0.1.5 10.0.1.2    TCP     66 35598 → 9999 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=923984603 TSecr=2070811742
484 164.794641308 0.001901185 10.0.1.5 10.0.1.2    TCP     66 53910 → 9999 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=923994311 TSecr=2070821450
504 170.677535626 0.001973679 10.0.1.5 10.0.1.2    TCP     66 57678 → 9999 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=924000194 TSecr=2070827333
541 182.329448447 0.002212392 10.0.1.5 10.0.1.2    TCP     66 54262 → 9999 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=924011846 TSecr=2070838985
```

Fig. 4. Filter's Results in Wireshark

```bash
# Client Side
for i in {1..100};
do
  echo $i
  nohup python client.py >> cmd.out 2>&1 & echo $!
   > cmd.pid
  sleep 1
  kill -9 `cat cmd.pid`
  sleep 1
done
# Server Side
flag=`ps -aux | [file path to server.py]/server.py
   | grep -v "grep" | wc -l`
while true
do
  if [ $flag -eq 0 ]
  then
    nohup python server.py
  fi
done
```

Listing 3. Two Bash Scripts for the tests

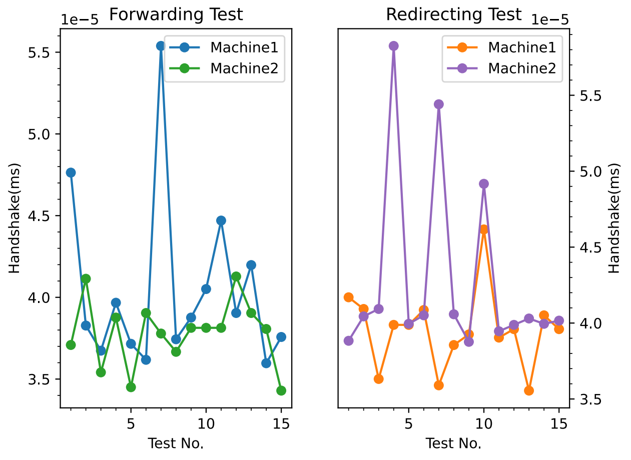


Fig. 5. Results of Testing

C. *Testing Plan and Result*

The following graph and Table V show the fluctuations of the results and the average values for multiple tests on each machine, respectively.

TABLE V

Machine	Forwarding Average Time(ms)	Redirecting Average Time(ms)
1	0.00005911313	0.00007210474
2	0.00005879671	0.00006867921

VI. Conclusion

In summary, the team was able to forward and redirect client traffic in a Linux environment by building an SDN network topology with Mininet and using the Ryu framework to make one client and two servers reachable to each other. The results of the client message capture showed that the traffic forwarding and redirection was generally fast and stable.

In the further work, this project implements forwarding and redirection of packets. Future work is planned to implement restricted access and proxy access. The first one can implement restricted access for specific users, which will effectively block useless information from specific sources and reduce the network load. The second one can use a certain server as a proxy server and be transparent to the user, so that access to the proxy server can be restricted to normal users.

References

[1] S. Wang, L. Nie, G. Li, Y. Wu and Z. Ning, "A Multitask Learning-Based Network Traffic Prediction Approach for SDN-Enabled Industrial Internet of Things," in IEEE Transactions on Industrial Informatics, vol. 18, no. 11, pp. 7475-7483, Nov. 2022, doi: 10.1109/TII.2022.3141743.

[2] M. Hasan, H. Dahshan, E. Abdelwanees and A. Elmoghazy, "SDN Mininet Emulator Benchmarking and Result Analysis," 2020 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES), 2020, pp. 355-360, doi: 10.1109/NILES50944.2020.9257913.

[3] P. -T. Tivig, E. Borcoci, A. Brumaru and A. -I. -E. Ciobanu, "Layer 3 Forwarder Application - Implementation Experiments Based on Ryu SDN Controller," 2021 International Symposium on Networks, Computers and Communications (ISNCC), 2021, pp. 1-6, doi: 10.1109/ISNCC52172.2021.9615685.

[4] David Borman and Robert T. Braden and Van Jacobson and Richard Scheffenegger, doi: 10.17487/RFC7323