Xi'an Jiaotong-Liverpool University

School of Advanced Technology

Networking Project on Traffic Control with SDN Network Topology

Authors:	*Student IDs*
Zheyuan Cao	2141805
Yanqing Fei	2143430
Ruiqi Chen	2141794
Fei Du	2141378

A Report Submitted For

CAN201 Introduction to Networking

December 10 2023

Networking Project on Traffic Control with SDN Network Topology

Zheyuan Cao
School of Advanced Technology
Xi'an Jiaotong-Liverpool University
Suzhou, China
Zheyuan.Cao21@student.xjtlu.edu.cn

Ruiqi Chen
School of Advanced Technology
Xi'an Jiaotong-Liverpool University
Suzhou, China
Ruiqi.Chen21@student.xjtlu.edu.cn

Yanqing Fei
School of Advanced Technology
Xi'an Jiaotong-Liverpool University
Suzhou, China
Yanqing.Fei21@student.xjtlu.edu.cn

Fei Du
School of Advanced Technology
Xi'an Jiaotong-Liverpool University
Suzhou, China
Fei.Du21@student.xjtlu.edu.cn

***Abstract*—Software Defined Network (SDN), a pivotal network architecture, significantly influencing the network management paradigms. This project integrates the Mininet with the Ryu framework to develop an SDN topology, aiming to achieve the client-transparent traffic control while guaranteeing reliable and efficient data transmission.**

***Index Terms*—Mininet, Switch, OpenFlow, SDN, Ryu, Traffic Control**

I. INTRODUCTION

Traditional per-router networking suffers from decentralized control, proprietary implementation and static configurations, which limits the flexibility and scalability and results in a departure from dynamic and evolving network demands. Thus, Software-Defined Networking (SDN), a logically centralized architecture that separates the control plane from the data plane, is applied to balance such deficiency in this research. Switch and controller constitute two principle components in SDN. The former is responsible for packets forwarding while the latter undertakes the implementation of management, adhering to specific protocols [1]. Specifically, the project can be divided into three main parts:

- Utilize the Mininet library to construct an SDN network topology with respective IP and MAC addresses while ensuring the accessibility of each node.
- Apply the Ryu framework to engineer an SDN controller that facilitates traffic forwarding from Client to Server 1, while capable of redirecting said traffic from Client to Server 1 towards Server 2.
- Employ Wireshark for packet capture in the network communication and subsequently compute the associated network latency.

Two resultant primary concerns, namely SDN topology establishment and Ryu task-specific implementation, are focal points of the study. Concrete to contributions, the project employed Mininet module to produce predefined topology structure with custom IP and MAC configurations, subsequently extended the Ryu framework to execute the traffic manipulation, i.e., forwarding and redirection, and deployed sockets for network transmission. The eventual product enhances the network performance and management. Specifically, relevant application range may involve following potential domains:

- **Load Balancing:** SDN controller allows to redirect client requests among multiple servers, avoiding the load imbalance and facilitating the congestion control, which is especially significant for data center management for example.
- **Secure Traffic Control:** Through origin-based traffic analysis, system can intelligently route data to appropriate destinations, thereby enhancing security. For instance, SDN controller can redirect and intercept suspicious traffic to protect network against attacks.
- **Disaster Recovery:** Upon encountering server failures, rapid dynamic redirection enables traffic to be automatically transmitted to backup devices to main-

tain the service continuity.

- **Quality of Service:** SDN traffic control provides a prioritized allocation of network resources and services for critical applications, optimizing QoS management.

II. Related Work

Owing to the limitations inherent in traditional network architecture, the pursuit of innovative patterns for network management is indispensable, which thus constitutes a primary focus within the realm of academic research. Deriving form those developmental outcomes, SDN, a prevalent centralized architecture that diverges from the conventional per-router control plane model, forms the foundational basis of the project. Specifically in this task, OpenFlow, a protocol permits the incorporation of an optional encryption mechanism namely Transport Layer Security (TLS) [2], is applied to define a specialized TCP connection channel [3] between controllers and switches. Florance et al. [4] instructed to construct a programmable SDN and OpenFlow based network topology model with Mininet emulator, which facilitates the design of this work. Furthermore, building upon a foundation of critical metrics including throughput, packet loss rate, and bandwidth utilization, Islam et al. [5] conducted an empirical investigation into the performance characteristics of the Ryu controller. Their research, which aligns so closely with the topical focus of this study that enlightens the detailed construction, involved a network topology analogous to the one examined herein, comprising a single Ryu controller, an OpenFlow-based switch, and three host nodes.

All academic research above significantly contributed to the realization of entire project, enriching various aspects, for example, the comprehension of SDN concepts and OpenFlow protocol, and methodologies involved in constructing network topology with Mininet and Ryu controller. This comprehensive scholarly foundation has been instrumental in enhancing the project's overall analysis, design and implementation phases.

III. Design

This section will focus on the project design, including an SDN network topology architecture, flow entry processing sequences and kernel algorithms of SDN controller.

A. *SDN Network Architecture*

The SDN topology is composed of a central SDN controller, a switch unit and three host of which one is client and two are servers. The detailed network structure is illustrated in figure 1.

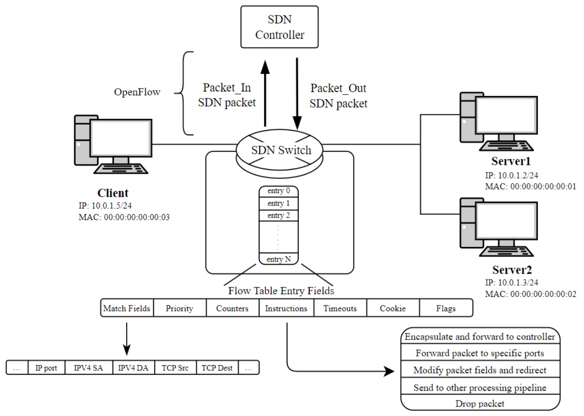


Fig. 1: SDN Network Architecture

Specifically, the main diagram elements are explained as follows:

- **SDN Controller:** The 3-layered [6] controller, programmed and extended within the open-source Ryu framework, is equipped with a variety of network control functionalities via the northbound API to conduct the traffic control. For instance, it possesses the capability to alter the flow entry match criteria in switch, thereby facilitating the forwarding or redirection of network packets. Furthermore, OpenFlow-based interactions between controller and switch encapsulates a series of events, notably the *Packet-In* and *Packet-Out*. Upon receiving *Packet-In* messages from switch, the controller responds with *Packet-Out* messages that encompasses instructions on handling certain packets, thus to dictate the network traffic flow and guarantee the performance.
- **SDN Switch:** The SDN switch maintains a flow table defined by the OpenFlow protocol, comprising entries with rules, actions and statistics. Specifically, 7 main components constitute a flow entry: Match Fields, Priority, Counters, Instructions, Timeouts, Cookie and Flags. In this project, the match field can be set to IPv4 address, TCP ports, etc. Upon receiving packets from specific ports, the switch processes them according to matched rules and predefined actions such as forwarding, redirecting and dropping in these entries, exemplifying the OpenFlow implementation in directing packet flow and enabling advanced network management. While for non-matching packets, the switch initiates a transfer to the controller via

Packet-In messages, subsequently waiting for *Packet-Out* responses that contain prescriptive directives on processing such packet flows.

- **Hosts:** All hosts are equipped with distinct IP and MAC addresses, assigned to their respective interfaces, to support for TCP-based connection and packets transmission. Additionally, socket programming enables bidirectional traffic transfer between the client and server.

B. *Workflow of System*

Critical application procedures is summarized as follows while the entire process is declared in the figure 2 below:

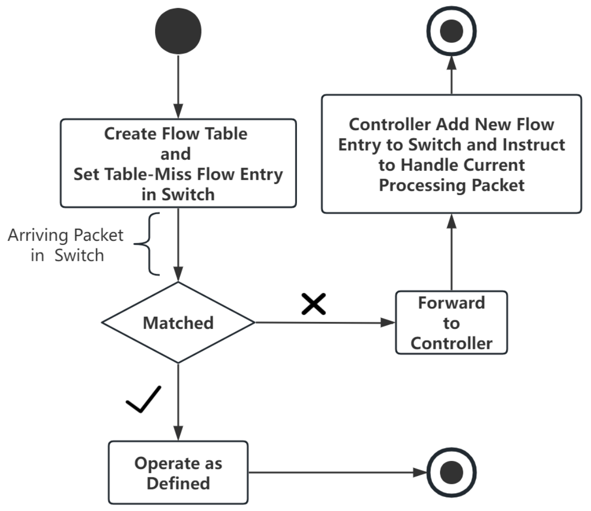

Create Flow Table and Set Table-Miss Flow Entry in Switch
Arriving Packet in Switch
Matched
✕
Forward to Controller
Controller Add New Flow Entry to Switch and Instruct to Handle Current Processing Packet
✓
Operate as Defined

Fig. 2: SDN Workflow

- Upon initialization, the SDN controller forces connected switches to instantiate a flow table with a persistent default entry, i.e., the table-miss flow entry, to guarantee that packets matching no pre-existing entries will be transferred to the controller.
- For arriving non-matching packets, the controller executes a parse to extract necessary fields such as MAC sources and destinations. The forwarding strategy hinges on the MAC destination address. Specifically, known MAC destinations prompt direct packet forwarding to the certain port otherwise flooding.
- In handling destination-specific flows, distinct strategies are employed contingent upon the protocol type of the traffic, such as ARP, ICMP, UDP, and TCP. These strategies involve modifications to matching fields and processing actions, encompassing standard forwarding and address modification for redirection.
- Various flow entries thus are created by controller and added to the switch for autonomously managing similar flows in the future.

C. *Algorithm*

The kernel redirection processing logic of controller is exhibited below:

Algorithm 1 SDN Controller Flow Redirection

```
FUNCTION ryu_redirection()
   SET table-miss flow to forward to controller
   WAIT for new flow packet
while new flow packet received do
        PARSE the packet
  if packet is ARP or ICMP then
             PERFORM simple forwarding
  else if packet is IP then
    if from client to server 1 then
               REDIRECT to server 2
    else if from server 2 to client then
               MODIFY source to appear as from server 1
    else
               FORWARD based on destination
    end if
  end if
        UPDATE flow table with new rule
end while
END FUNCTION
```

IV. IMPLEMENTATION

A. *Host Environment*

The development environment for SDN implementation is listed in table I below:

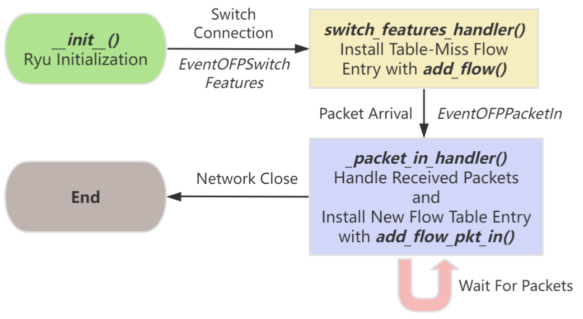
TABLE I: Host Implementation Environment

CPU	Intel(R) Core(TM) i7-1165G7
RAM	16.0 GB
OS	Windows 11 x64
IDE	PyCharm (Python 3.9)
SDN Controller	Ryu

B. *Programming Skills*

- **Object-Oriented Programming (OOP):** The *SimpleSwitch13* class, through inheritance from its superclass *RyuApp*, acquires pertinent attributes and methods in Ryu manager, constituting the foundational architecture of the program. Each instantiation of the class represents a distinct object in OOP, which guarantees the intrinsic class cohesion while enhancing the reusability and scalability.
- **Modular Programming:** Modularization subdivides the entire project into diverse methods that ded-

icated to specific functionalities, thereby facilitating the readability and maintainability. For instance, *add_flow_pkt_in* aims to add new flow entry, excluding the table-miss, to the switch while *_packet_in_handler* mainly for received packets management in the controller. Figure 3 below shows the corresponding program flow chart.



__init__()
Ryu Initialization
Switch Connection
EventOFPSwitchFeatures
switch_features_handler()
Install Table-Miss Flow Entry with add_flow()
Packet Arrival
EventOFPPacketIn
_packet_in_handler()
Handle Received Packets and Install New Flow Table Entry with add_flow_pkt_in()
Network Close
End
Wait For Packets

Fig. 3: SDN Program Implementation Flow

- **Event-Driven Programming:** Event-driven paradigm implies that program execution routines rely on external events. This mechanism is implemented by the decorator @*set_ev_cls* in Python, which serves as an instrumental construct for linking event occurrences with their corresponding event handling functions in Ryu SDN controller.
- **Encapsulation:** This work incorporates various kinds of encapsulation. For example, distinct attributes, such as the *mac_to_port* dictionary, along with functions like *add_flow*, are encapsulated within the class, which therefore maintains the security and robustness of program.

C. Actual Implementation

Following codes displays the core logic of traffic redirection, including modification to destination and source to accomplish the network communication between the client and server 2. Moreover, to avoid the Denial-of-Service (DoS) that mainly attributed to the flooding attack, *ip_src* and *tcp_src* are removed from the normal matching fields.

Listing 1: Redirection Implementation

```
# Modify destination to redirect traffic to
    server 2
if ip_src == client_ip and ip_dst == server_1_ip:
    if server_2_mac in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][
            server_2_mac]
    else:
        out_port = ofproto.OFPP_FLOOD

    match = parser.OFPMatch(eth_type=ether_types.
        ETH_TYPE_IP, ipv4_src=ip_src, ipv4_dst=
        ip_dst)

    actions = [parser.OFPActionSetField(eth_dst=
        server_2_mac),
               parser.OFPActionSetField(ipv4_dst=
                   server_2_ip),
               parser.OFPActionOutput(port=
                   out_port)]

# Modify source to disguise as server 1 caz
    client is unaware of server 2
elif ip_src == server_2_ip and ip_dst ==
    client_ip:
    if client_mac in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][
            client_mac]
    else:
        out_port = ofproto.OFPP_FLOOD

    match = parser.OFPMatch(eth_type=ether_types.
        ETH_TYPE_IP, ipv4_src=ip_src, ipv4_dst=
        ip_dst)

    actions = [parser.OFPActionSetField(eth_src=
        server_1_mac),
               parser.OFPActionSetField(ipv4_src=
                   server_1_ip),
               parser.OFPActionOutput(port=
                   out_port)]
```

D. Difficulties

During the implementation course there encountered two main challenges: redirection failure and port unavailability. The former problem referred to the inability to reroute Client traffic to Server 2, attributable to syntax errors in variables for IP definition, which resolved by removing the subnet mask '*/24*'. The latter issue was characterized by the occurrence of port binding conflicts, indicating that the intended port was already in use. This was settled by terminating the process currently occupying the port to allow for successful rebinding to the desired port.

V. Testing and Results

A. Testing Environment

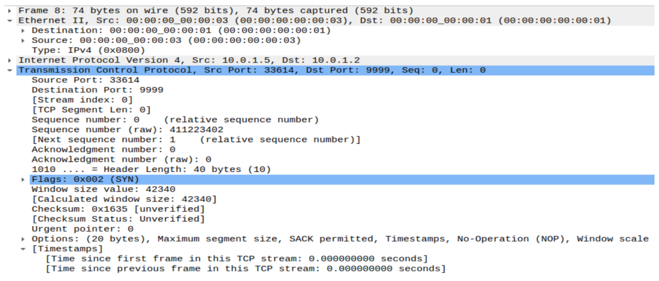
TABLE II: Test Environment

CPU	Intel(R) Core(TM) i7-1165G7
RAM	4096MB
OS	Ubuntu (64-bit)
TYPE	Linux
Interpreter	Python 3.8.10

Upon the development phrase, practical tests were carried out on a virtual machine whose configuration details are listed in table II above.

B. *Measurement Techniques*

Wireshrak, an instrumental tool for the acquisition of network traffic details such as source, destination and packet category, is applied to facilitate the packet capture and performance analysis in this project. A key metric, *Time*, is critical for assessing network performance, denoting the precise moment of packet capture. This is further detailed by *Timestamps* that contains the elapsed time from the initial TCP stream frame (SYN). An example of traffic information is presented in figure 4 below.

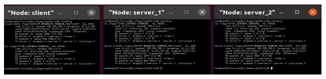


```
▸ Frame 8: 74 bytes on wire (592 bits), 74 bytes captured (592 bits)
▾ Ethernet II, Src: 00:00:00_00:00:03 (00:00:00:00:00:03), Dst: 00:00:00_00:00:01 (00:00:00:00:00:01)
  ▸ Destination: 00:00:00_00:00:01 (00:00:00:00:00:01)
  ▸ Source: 00:00:00_00:00:03 (00:00:00:00:00:03)
    Type: IPv4 (0x0800)
▸ Internet Protocol Version 4, Src: 10.0.1.5, Dst: 10.0.1.2
▾ Transmission Control Protocol, Src Port: 33614, Dst Port: 9999, Seq: 0, Len: 0
    Source Port: 33614
    Destination Port: 9999
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 0    (relative sequence number)
    Sequence number (raw): 411223402
    [Next sequence number: 1    (relative sequence number)]
    Acknowledgment number: 0
    Acknowledgment number (raw): 0
    1010 .... = Header Length: 40 bytes (10)
  ▸ Flags: 0x002 (SYN)
    Window size value: 42340
    [Calculated window size: 42340]
    Checksum: 0x1635 [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
  ▸ Options: (20 bytes), Maximum segment size, SACK permitted, Timestamps, No-Operation (NOP), Window scale
  ▾ [Timestamps]
      [Time since first frame in this TCP stream: 0.000000000 seconds]
      [Time since previous frame in this TCP stream: 0.000000000 seconds]
```

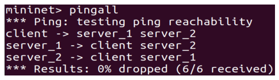
Fig. 4: Packet Information

C. *Testing Steps and Results*

- After executing the *networkTopo.py* script, an SDN topology is constructed with predefined address configurations, presented in figure 5(a). Subsequent to this initialization, figure 5(b) shows that *ping* operations are available for all nodes within the network, demonstrating full network connectivity.



(a) SDN Topology



```
mininet> pingall
*** Ping: testing ping reachability
client -> server_1 server_2
server_1 -> client server_2
server_2 -> client server_1
*** Results: 0% dropped (6/6 received)
```

(b) Ping Reachability

Fig. 5: Topology Construction

- For forwarding, the controller transfers the traffic from Client to Server 1 with *Packet-In* and *Packet-Out* message displayed in terminal. Concurrently, the switch maintains a flow table which encompasses a permanent table-miss flow entry alongside temporary normal entries adhering to certain rules, ensuring the efficient and dynamic routing. Relevant results are revealed in figure 6.
- For redirection, the SDN controller is reprogrammed to reroute Client traffic to Server 2. Given that service on Server 2 operates in a client-transparent manner, address modifications are thus indispensable. This involves redefining the destination for Client and altering the client-destined traffic source to pose Server 2 as Server 1. Figure 7 illustrates the redirected communication process.

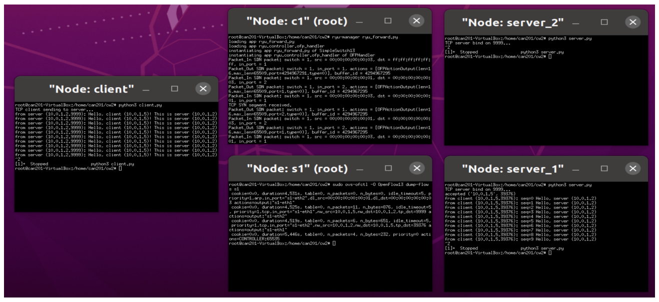


Fig. 6: Packet Forwarding

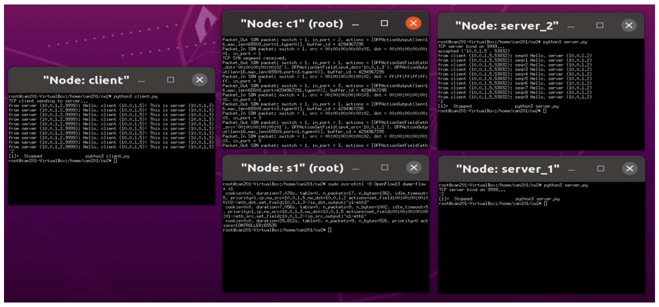


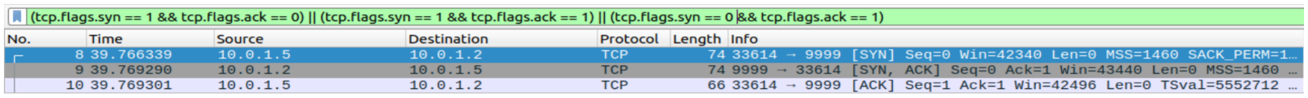
Fig. 7: Packet Redirection

- Throughout the packet forwarding and redirection procedure, Wireshark was employed to capture involved packets and extract essential information, particularly pertaining to the TCP 3-way handshake sequence: the initial SYN (synchronization) message, followed by the SYN-ACK (synchronization-acknowledgement) response and the final ACK (acknowledgement). Specifically, the following filter codes can obtain the desired packets, which are exhibited in figure 8.

```
(tcp.flags.syn==1 && tcp.flags.ack==0) ||
(tcp.flags.syn==1 && tcp.flags.ack==1) ||
(tcp.flags.syn==0 && tcp.flags.ack==1)
```

D. *Performance*

Figure 9 depicts a comparison of the average network latency for traffic forwarding and redirection, as calculated from the aforementioned timestamps intervals. The line chat sees a fluctuation which potentially attributed to network conditions or hardware performance constraints. Moreover, it is noted that the average network delay associated with forwarding is inferior to that of redirection, a disparity likely stemming from the increased complexity



```
(tcp.flags.syn == 1 && tcp.flags.ack == 0) || (tcp.flags.syn == 1 && tcp.flags.ack == 1) || (tcp.flags.syn == 0 && tcp.flags.ack == 1)
No.   Time        Source      Destination  Protocol Length Info
   8 39.766339    10.0.1.5    10.0.1.2     TCP          74 33614 → 9999 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM=1…
   9 39.769290    10.0.1.2    10.0.1.5     TCP          74 9999 → 33614 [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460 …
  10 39.769301    10.0.1.5    10.0.1.2     TCP          66 33614 → 9999 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=5552712 …
```

Fig. 8: TCP 3-Way Handshake

in redirection operations such as routing decisions and address modifications.

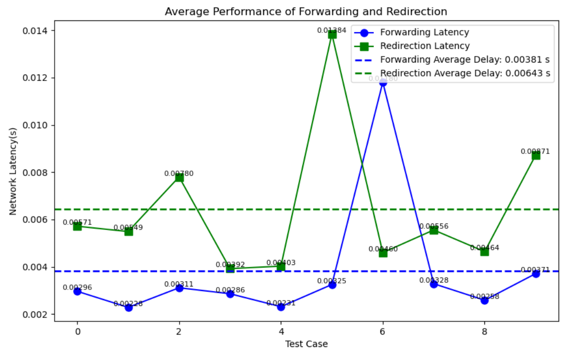


Fig. 9: Forwarding and Redirection Average Performance

VI. CONCLUSION

In conclusion, this project achieved the efficient and stable traffic forwarding and redirection within an SDN network topology, constructed via the utilization of Mininet library and Ryu framework.

Future research will extend its scope to include the secure access and fault tolerance mechanisms. The first focuses on mitigating network vulnerabilities and filter redundant information from the server through redirection of source-unidentified traffic. For the second proposal, it aims to reroute traffic to a backup server to sustain the network communication in case of failures in the primary link.

REFERENCES

[1] A. Yazdinejad, R. M. Parizi, A. Dehghantanha, Q. Zhang, and K. R. Choo, "An energy-efficient sdn controller architecture for iot networks with blockchain-based security," *IEEE Transactions on Services Computing*, vol. 13, pp. 625–638, 2020. I

[2] S. Midha and K. Triptahi, "Extended tls security and defensive algorithm in openflow sdn," *2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, pp. 141–146, 2019. II

[3] R. Wazirali, R. A. Ahmad, and S. Alhiyari, "Sdn-openflow topology discovery: An overview of performance issues," *Applied Sciences*, 2021. II

[4] G. Florance and R. J. Anandhi, "Study on sdn with security issues using mininet," *Recent Trends in Intensive Computing*, 2021. II

[5] M. T. Islam, N. Islam, and M. A. Refat, "Node to node performance evaluation through ryu sdn controller," *Wireless Personal Communications*, vol. 112, pp. 555–570, 2020. II

[6] S. Bhardwaj and S. N. Panda, "Performance evaluation using ryu sdn controller in software-defined networking environment," *Wireless Personal Communications*, vol. 122, pp. 701–723, 2021. III-A