FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Master Project

Implementation of an SDN-based network in CORE with OpenFlow

Submitted by: Riyad – Ul – Islam
First examiner: Prof. Dr. Armin Lehmann
Date of start: 28.06.2021
Date of submission: 29.11.2021

Statement

Content

Abstract

Software Defined Networking (SDN) has been defined as a core component for new era in modern computer networking. SDN is an evolving networking concept which differentiates the network control plane from the data forwarding plane that significantly improves network resource utilization, simplify network management, decreases operating costs, and supports innovation and evolution. This project is focused on network implementation on CORE network emulator using SDN technology. While working on this project, several SDN controllers such as- Ryu controller, Open Network Operating System (ONOS), Floodlight have been taken into consideration for realizing different kinds of approaches. OpenFlow protocol using Open vSwitch (OVS), REST API, Network Virtualization, Network Slicing, connecting traditional Autonomous System with SDN network using Border Gateway Protocol (BGP) technique have been played obligatory role for designing and testing SDN network approaches throughout the project.

***Keywords* - Ryu, Floodlight, ONOS, OpenVirtex, CORE, OpenFlow, Open vSwitch, SDN, BGP**

1 Introduction

Traditionally, the distributed control and transport network systems operating inside the routers and switches are the main mechanism, which transmit information enclosed in digital packets around the world. Despite their extensive implementation, traditional IP networks are complicated and difficult to manage, where each independent node in the network has to be configured separately. This becomes troublesome for maintaining and bringing change to the network, and makes a slow and expensive process [1].

Software-Defined Network (SDN) is one of the latest and popular network concept which decouples control plane decisions from data plane, in regards to make and maintain connections for high-speed forwarding of packets. Through separation, SDN brings simplification in network management and contribute to network modernization by breaking down traditional IP network, and changing the networking concept to realise programmatic idea of network [2]. In SDN concept, control-plane monitors and decides how a device reacts to the packet that comes to it as a part of a computer network device, which is done through SDN controller. On the other hand, responsibility of the data-plane is forwarding packet [3][4].

Now this era of 5G network infrastructure demands application-centric networks, that facilitates the use of new data planes attainable over a programmable compute, storage, and transport infrastructure. Network virtualization and network slicing are key factors for the deployment of 5G networks. Software Defined Networking (SDN) proposed the idea of network programmability offering both a standard protocol to program network devices (i.e., OpenFlow), and a uniform vision of network devices [9]. These concepts strongly enable the implementation of network slicing and network virtualization by introducing the configuration of the data plane in both flexible and dynamic manner [5][6].

This work introduces how SDN concept using three different SDN controller can be employed to implement through an emulated testbed called CORE (Common Open Research Emulator) [19], where a number of Open vSwitches (OVS) [20] is deployed to make the data plane. The communication with the control plane takes place over OpenFlow version 1.3 [12]. Then it proposes several approaches like network slicing, network virtualization, connection between traditional autonomous system and SDN network based autonomous system over BGP protocol. The SDN concept is obtained by using Ryu [18], Floodlight, Open Network Operating System (ONOS) [21] SDN controller.

2 Theoretical Background

Throughout the project, some of the major networking concepts have arisen to deal with for better understanding, including SDN architecture and several protocols. For instance OpenFlow, BGP. In addition, the concept of network slicing, network virtualization, virtual switches, hypervisors, emulation tools (i.e., CORE) come across, and facilitate the implementation of the project.

2.1 SDN Architecture

The core concept of SDN is to separate network control and forwarding functions from each other. The network control has to be directly programmable and the infrastructure has to be conceptualized for network services and applications [7].

Figure 2.1 summarizes the SDN architecture concepts in the form of a comprehensive, high-level representation. In such an architecture, infrastructure devices behave as simple forwarding devices, that handle packets coming according to several policies generated directly by a controller in the control layer according to pre-described program logic. The controller generally manages on a distant server, and connects over a secured connection to the forwarding elements utilizing a number of uniform instructions. SDN architecture consist of 3 key layers; i) Application layer, ii) Control layer & iii) Infrastructure layer [10].

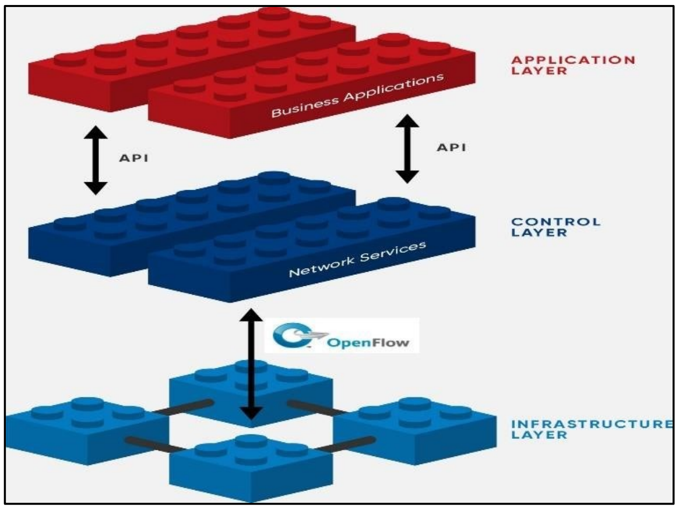


Figure 2.1: SDN Architecture

Infrastructure Layer: This layer usually known as the data plane, which takes responsibility of forwarding messages through forwarding elements, that involve physical and virtual switches, servers and routers, which can be accessed by control layer. This layer is the physical layer, that would be demonstrate under the control layer [9].

Control Layer: Control layer known as the control plane, is the logical section of the SDN. This layer is composed of a certain amount of software-based SDN controllers, that controls the network infrastructure. In SDN theory, configuration and execution of various network applications can be done centrally in the SDN controller. The role of the controller in SDN becomes very important, as the functionality of bridging network applications with the networks physical devices or data-plane. This functionality is done through the interface concept, where two main interfaces in the SDN controller, specifically the southbound interface and the northbound interface take the role. The southbound interface links the controller with network devices in the data-plane. However, the northbound interface connects the controller to the network application layer [8].

Application Layer: This layer is mainly focused on end-user applications, which use SDN communication, and network services. It is an open area to develop application as much as possible to support network topology, network state, network statistics, etc [10].

2.2 Protocol

There are two main network protocols that facilitate the project work: OpenFlow version 1.3, which connect the control plane to data plane, and BGP that designed to exchange routing and reachability information among autonomous systems.

2.2.1 OpenFlow

OpenFlow is one of the most common protocols employed on the southbound interface, which connects the SDN controller with OpenFlow switch over a secure channel on either well known Transmission Control Protocol (TCP) port 6633 or 6653. Because of using OpenFlow as the southbound protocol, the SDN controller is called OpenFlow Controller, while the data-plane is called OpenFlow Switch [11].

OpenFlow has also evolved, coming under the management of the Open Networking Foundation (ONF) founded in 2011 for the promotion and adoption of SDN through open standards development. OpenFlow has evolved to version 1.5.1, however, hardware typically supports up to v1.3 [12].

OpenFlow switches (i.e., Open vSwitch) have a flow-table and a group-table or a meter table, which operate on matching the packet and take action with forwarding. Various flow entries can be in a single or multiple Flow-table. An OpenFlow switch is linked to SDN controller via OpenFlow channel. SDN controller with the help of OpenFlow protocol, can add, update, and delete the flow-entries within the flow-table through some dynamic program. Any packet comes to switch first check for matching in the flow table according to pre-set priority. The Switch takes action if any matching entry is noticed. Otherwise, it executes in accordance with table miss entry [11].

All the communication between SDN controller and a OpenFlow switch occurs over OpenFlow messages. After establishing a TCP and Transport Layer Security (TLS) handshake, switch and controller start communication with hello message exchange. Afterward, in respond to controller's feature request, switch sends information about it's own supported capabilities. Entries in the flow table are done with modify state message from controller. If a switch sends a data packet to controller, it uses the packet in message, and controller uses packet out message back to the switch after analysing packet reached over packet in. The active connection indication can be observed through Echo request or reply [12].

2.2.2 BGP

BGP is an exterior gateway protocol which is based on a path vector approach. A BGP speaker creates a session over TCP with its directly connected neighbors, and shares routing information with each other. Then it updates the routing table based on the routing information received. BGP uses TCP for reliable transport of its packets, on well known port 179. BGP most commonly is used, when multiple autonomous system are connected [13].

In BGP functionality, BGP speakers should make neighbor relationships, which is called peers. Commonly, there are two types of BGP peers, which are iBGP peers that creates neighbor within same autonomous system, and eBGP peers that connect egress router of separate autonomous system [14].

BGP creates its peer relations over a series of messages. Initially, with an OPEN message, BGP starts the session. BGP Version, Local AS Number, BGP Router ID are some of the parameters for the OPEN message. Within every 60 seconds by default a KEEPALIVE message is sent to make sure that the remote peer is still reachable. The routes between the neighbors are shared through UPDATE message. Lastly, If any error occurs, NOTIFICATION messages are sent [14].

2.3 Slicing

The new networking technology is migrating from a discrete network system to a shared network for different application service. Particularly, network slicing is a significant feature for 5G system [5], as it allows operators to flexibly arrange the network resources, while offering a range of services to subscribers. The concept of network slicing appeared in recent times as a way of establishing a number of logical networks (i.e., network slices) to present a set of services on a single physical network. Figure 2.2 provides a visualization of slicing the network.

Functionally, the control and management plane is planned for slice commissioning, and dynamic reconfiguration. On the other hand, the data plane ensures each slice requirements. Software Defined Networking SDN provides the idea of network programmability offering both a standard protocol to program network devices (i.e., OpenFlow), and a uniform vision of network devices. These perceptions strongly enable the implementation of network slicing through establishing both flexible and dynamic way of configuration in the data plane [6].

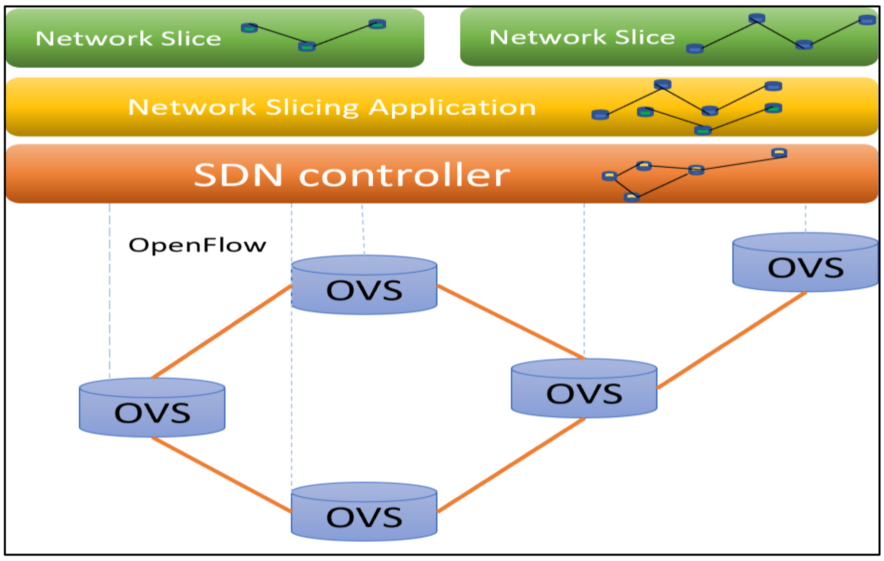


Figure 2.2: Network slicing using a SDN application

2.4 Network Virtualization

Fundamentally, network virtualization allows operating several individual networks on top of a single shared physical network infrastructure. In this way, role of the service provider of the traditional network can be divided into infrastructure based and service based. The responsibility of physical network is carried out by Infrastructure providers, on the other side providing the service with virtual network resources to customer can be managed by service providers. The key idea is to maintain a flexible, manageable and secure network architecture.

This approach tackles the complexity of deploying and accepting new improvements in the network structure of the Internet [8].

Network virtualization is performed in this project through a hypervisor called OpenVirteX, which offers multiple tenants to occupy the same network infrastructure. In this concept, every tenants illude that they are accessing the full network to their own. OpenVirteX applies network virtualization by being a proxy which stays in between the physical network and tenants' network OSes. Conception of virtualization through OpenVirtex is presented in Figure 2.3. Functionally, OpenVirteX works like a (de)multiplexer for OpenFlow messages that flow between the tenant control planes and the physical network [16].

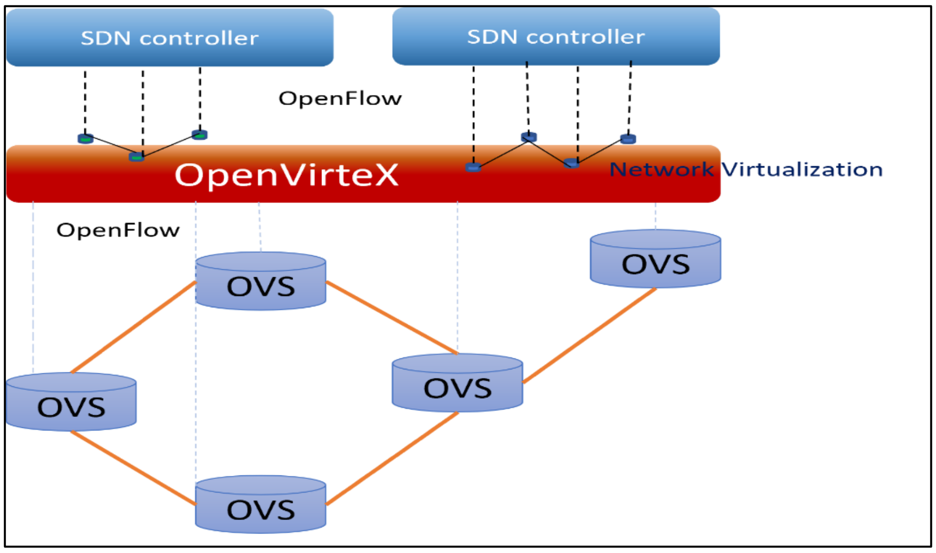


Figure 2.3: Network virtualization using OpenVirtex

2.5 Emulation Tools(CORE)

The Common Open Research Emulator, or CORE, which is a basis for emulating networks on one or more computers. Routers, PCs, Switches and other hosts can be emulated by CORE, and additionally, network links between them can be simulated. Since Core is a live emulation, real-time physical networks can also be connected with the help of RJ45 connectors to CORE network. SDN controller, Open vSwitches, wireless routers can be used in CORE emulators network. Basically, CORE is established on the open source Integrated Multi-protocol Network Emulator/Simulator (IMUNES) from the University of Zagreb. Generally, network research and demonstrations, application and platform experiment, different types of network security studies can be done utilizing CORE. Using tools offered by Linux operating system, CORE gives an environment for experimenting real application and protocols [15].

Architecturally, CORE primarily composed of core-daemon, core-gui, coresendmsg, vcmd. Emulated sessions of nodes and links of a network are formed using Linux namespace, Linux bridges and virtual ethernet peers, is managed by core-daemon. CORE-GUI and core-daemon connect using custom TLV API. CORE-GUI is a TCL/TK program that can creates nodes and links, and can initiate session with start button. coresendmsg is basically used to send TLV API messages for communicating with core-daemon. Lastly, vcmd is command line interface which sends shell command to the nodes [19].

3 Project Description

3.1 General Objectives

To implement a SDN based network, which can be employed as multitenant sliced network, accomplished using network emulation environment CORE [19]. The implemented network has to be done comprising multiple host and multiple servers, which is running on CORE emulator. The physical network devices from data plane supporting OpenFlow version 1.3 protocol are to be controlled by SDN controller.

3.2 Approachs

While implementing this project, there are three different approaches that have to be considered. Firstly, using RYU which is python based SDN controller [18]. A RYU application has been established to communicate with the RYU SDN controller in northbound interface, which then enable the controller to communicate with Open vSwitch [20] using OpenFlow version 1.3 protocol. Secondly, network virtualization is considered with the help of OpenVirteX [16] as a hypervisor, and Floodlight SDN controller. Lastly, another experiment has been done with the help of ONOS [21] SDN controller to connect several autonomous systems of different networks through BGP protocol.

3.2.1 1st Approach

In this approach, the experimental goal is to make a multitenant sliced network. In this regards, nine Open vSwitches (OVS) are used to build the SDN data plane network. Figure 3.1 presents the topology for implementing the 1st approach which is done in CORE emulator.

In the network diagram, all the Open vSwitches are connected to a simple layer 2 physical switch from interface eth0. The traditional switch is then connected with a RJ45 connector from CORE emulator to localhost of CORE hosted machine. The RYU SDN controller running on the same virtual machine connects to the CORE emulated physical devices.

The purpose of this topology is to make two slices using ICMP protocol, namely slice 300 and slice 400. Slice 300 consists of User1, User3 and Server2, while slice 400 comprises of User4, User2, Server1. Slice 300 devices can only able to reach each other through ICMP ping, which is totally independent from slice 400 devices. This is also similarly applicable for slice 400. In addition, service based slices are also implemented on this experiment, and those slices are defined as slice 600 and slice 700. Two services, one is webservice with HTTP port 80 and another SSH with port 22 are considered in this method. These services are sliced on the basis of TCP protocol. Both of the services have been running on server1, but only one service which is in a specific slice can be reached by tenant of that slice. As a result, slice 600 comprised of User1, User2 can get webservice through TCP port 80 and on the other hand, slice 600 composed of User3, User4 can get SSH service through TCP port 22.

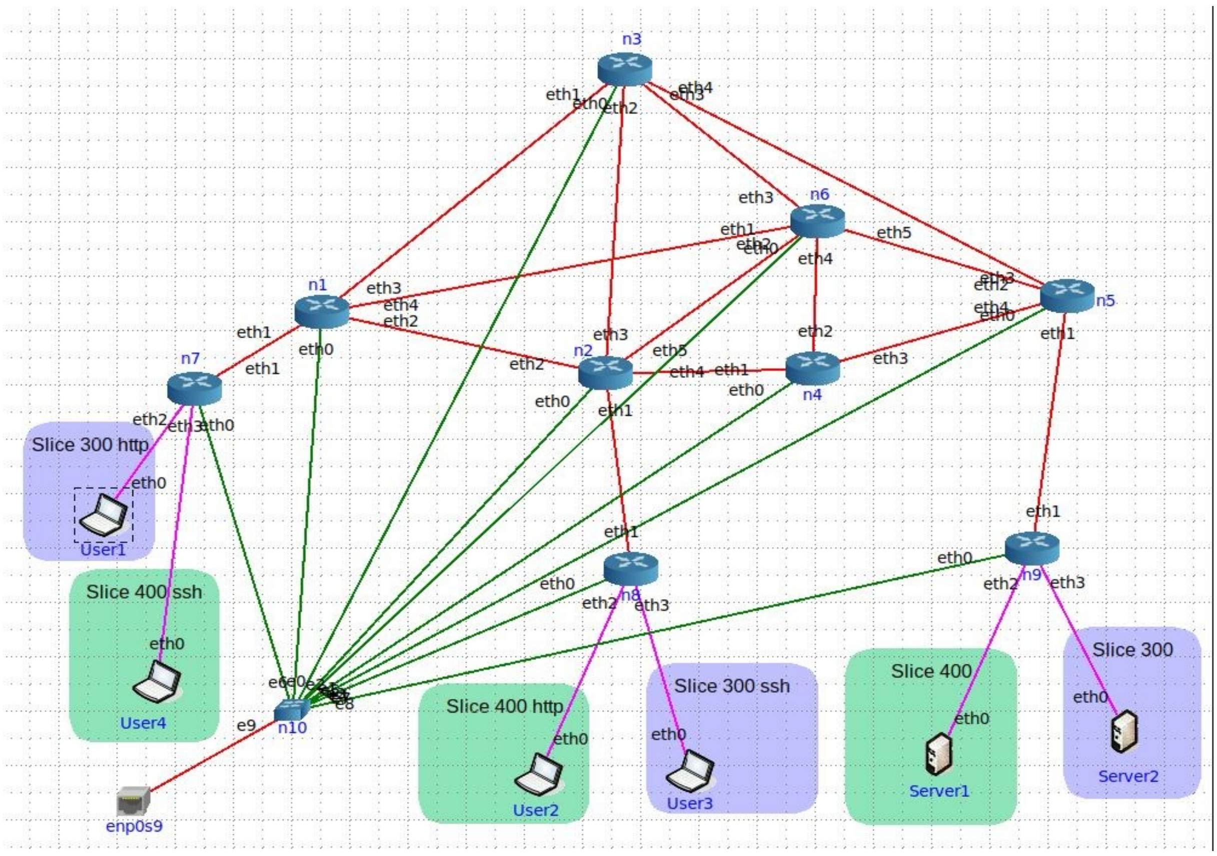


Figure 3.1: Designed Network Infrastructure for Controlling through RYU SDN controller

RYU SDN controller provides application written on python language. In this setup, RYU application take into account all OpenFlow version 1.3 messages and manages properly for establishing communication with OVS switch, which is providing service in the data plane.

Initially, Open vSwitch forms connection to RYU controller over TCP port 6633 and forwards a OpenFlow Hello message and RYU controller replies the Hello back to the switch. Then RYU controller gets feature of the switch with Feature Reply Message from switch. RYU controller sends Modify-State message for adding, removing and modifying flow to the flow table. In RYU application, all these activities are done with some event handler and custom functions. The function called switch_features_handler() used to handle the response of requested feature from OpenFlow switch, for instance DPID of the switch or OpenFlow messages. For configuration of the flow table, if there is no matching found, controller instructs the switch to add a flow of zero matching in the flow table, and to send a packet in message to controller. Controller application adds a flow to switch using add_flow() function, and sends a packet out message using send_msg() function. switch_features_handler() function for designing the application is shown in Figure 3.2.

To process the packets such as: IPv6 packet, ARP packet, IPv4 packet encapsulated in the packet in messages, _packet_in_handler() function is used in the RYU application. This function in the controller application also deals with various protocols. For example ICMP protocol, TCP protocol are processed to satisfy the project purpose. Slicing logic in the controller application is done based on ICMP and TCP protocol of ipv4 packet, and sliced the tenant over their mac address. A list is taken in the application for declaring different slices. Figure 3.3 illustrates the slicing list based on mac addresses.

```
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
  def switch_features_handler(self, ev):
    datapath = ev.msg.datapath
    dpid = datapath.id
    ofproto = datapath.ofproto parser
    = datapath.ofproto_parser

    # install table-miss flow entry
    match = parser.OFPMatch()
    actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                      ofproto.OFPCML_NO_BUFFER)]
    self.add_flow(datapath, 0,  match, actions)
    self.logger.info("switch:%s connected", dpid)
```

Figure 3.2: Switch feature handler function for designing application for the controller

```
    # Slices 300 and 400 with their respective MAC addresses for ICMP
    slices_data2 = [(300,"00:00:00:00:00:11","00:00:00:00:00:15","00:00:00:00:00:16",),
            (400,"00:00:00:00:00:14","00:00:00:00:00:12","00:00:00:00:00:13",) ]

    # Slices 600 and 700 with their respective MAC addresses and TCP port
    slices_data4 = [(600,"00:00:00:00:00:11","00:00:00:00:00:12","00:00:00:00:00:13",80),
       (700,"00:00:00:00:00:14","00:00:00:00:00:15","00:00:00:00:00:13",22)]
```

Figure 3.3: Defined slices based on the mac address of users

The main logic for slicing within the application occurs in _packet_in_handler() function. The packet is matched on the basis of ethernet source, ethernet destination, and in addition, TCP source port and TCP destination port. For this project implementation, TCP ports are 80 and 20. The key functionality of slicing is presented in Figure 3.4.

```
if protocol == in_proto.IPPROTO_TCP:   #Check TCP protocol
  _tcp = pkt.get_protocol(tcp.tcp)
  dst_port = _tcp.dst_port          # Source and dest. port capture
  src_port = _tcp.src_port
  for net_slice in slices_data4:
    slice_id = net_slice[0]     # extract the slice ID
    net_slice = net_slice[1:]  # extract Slice member from list
    if src in net_slice and dst in net_slice and (dst_port in net_slice or src_port in net_slice): # chekcing s
ource and dest. mac address  in addition with source port or dest. port are in the same list or not .
      self.logger.info("dpid %s in eth%s out eth%s", dpid, in_port, out_port)
      self.logger.info("Slice pair [%s, %s] in slice %i protocol %s dst_port %s", src, dst, slice_id,protoc
ol, dst_port)
      # Flow table matching accordence with source and dest. mac, port, protocol
      match = parser.OFPMatch(in_port=in_port, eth_dst=eth.dst, eth_src=eth.src,eth_type=ether_type
s.ETH_TYPE_IP, ip_proto=protocol,tcp_src= src_port, tcp_dst= dst_port )
      if msg.buffer_id != ofproto.OFP_NO_BUFFER:
        # Flow additon in the flow table
        self.add_flow(datapath, 1, match, actions, msg.buffer_id)
        return
      else:
        self.add_flow(datapath, 1, match, actions)
  else: # pair of MAC addresses are not in a slice so skip
    return
```

Figure 3.4: Key logic to perform the slicing procedure

3.2.2 2nd Approach

This approach is based on network virtualization. In this project, OpenVirteX hypervisor is considered to fulfil the virtualization of network, switches and links. The virtualization of switches depends on a concept of BIG switch. The topology of the physical network for this experiment is depicted in Figure 3.5. There are nine Open vSwitches used to compose this topology which are connected to the local virtual machine that runs CORE emulator. OpenVirteX runs on the other virtual machine, but is in the same network of the virtual machine runs CORE. In this way, OpenVirteX hypervisor is connected to all the Open vSwitches for controlling the data plane.

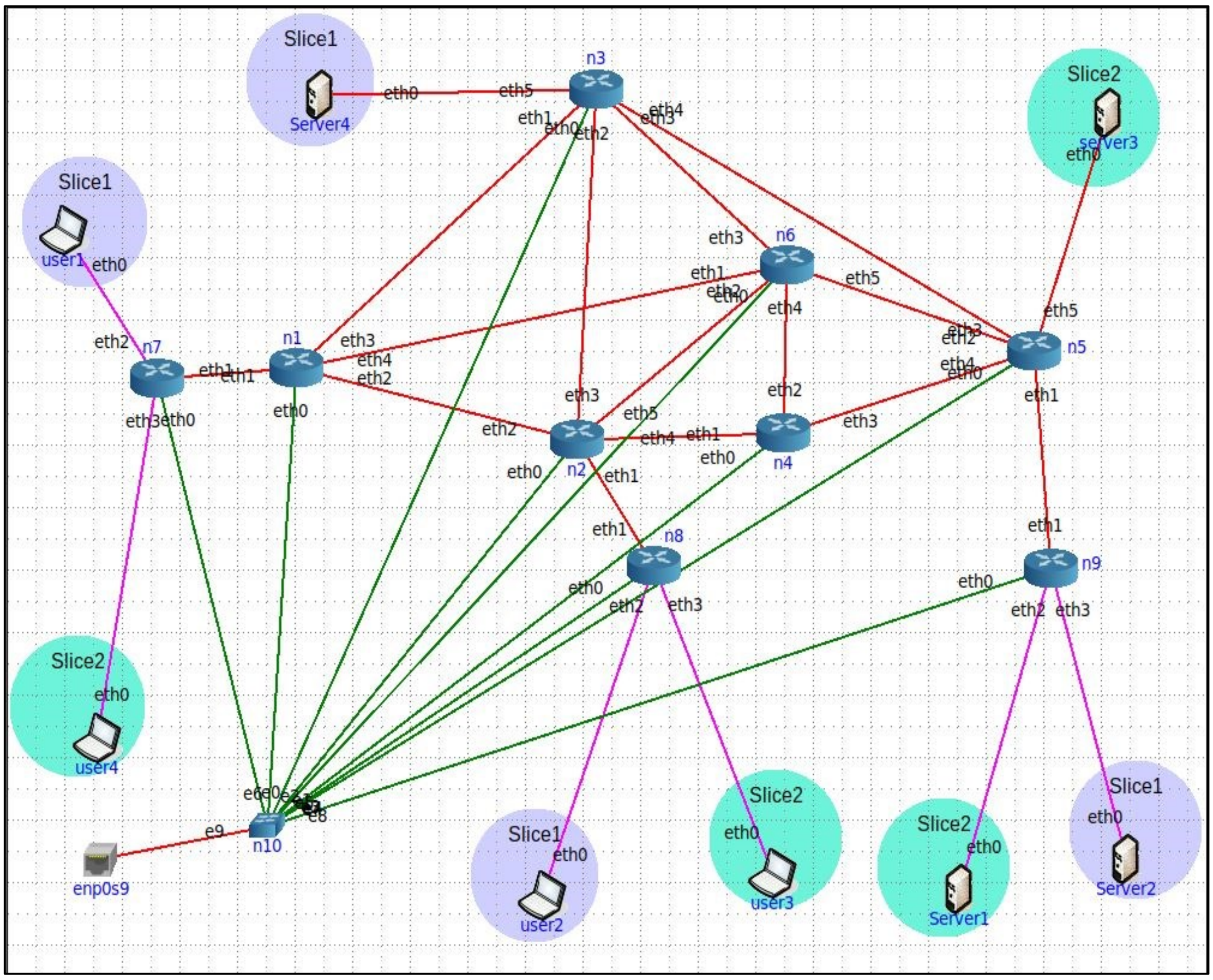


Figure 3.5: Network topology for performing the 2nd experiment

OpenVirteX that is connected to all the Open vSwitches for controlling the data plane virtually and creates a virtual network. OpenVirteX builds fully independent virtual network infrastructure, where tenant with a same tenant id thinks the overall network to their own.

In this approach, in a same slice, switches that are connected to the tenants are considered as a single big switch. OpenVirteX forms this big virtual switch with a tenant id by the instruction over REST API and handover the controlling functionality of these big switches to several SDN controllers. A number of Floodlight SDN controllers have been chosen to control different slices. The REST data is given in a json format to OpenVirteX.

In the json data, mac addresses of users in the first slice comprising User1, User2, Server2 and Server4, DPIDs of the switches and port number of user's connected Open vSwitch are given. Moreover, tenant id, which is 1 for the first slice, network subnet, routing algorithm, and the address of SDN controller are also binded in the json data.

For second slice, which has tenant id with 2, data containing mac addresses of User3, User4, Server3 and Server1, DPIDs of the switches connected to users, and port number of the respective switch connected to users is given with REST API. OpenVirteX then creates two independent virtual big switches, one virtual switch out of switch number 7,8,9,3 for slice 1 and another including switches 7,8,9,5 for slice 2. In addition, OpenVirtex initiates two Floodlight SDN controller for controlling these two different slices. Figure 3.6 summarises the virtualization technique of creating single virtual switch composed out of multiple physical switches.

With the routing algorithm information, OpenVirteX creates the route in the physical devices. If any interruptions occur in the physical path, there are a number of backup path will take responsibility that is also specified through REST API.

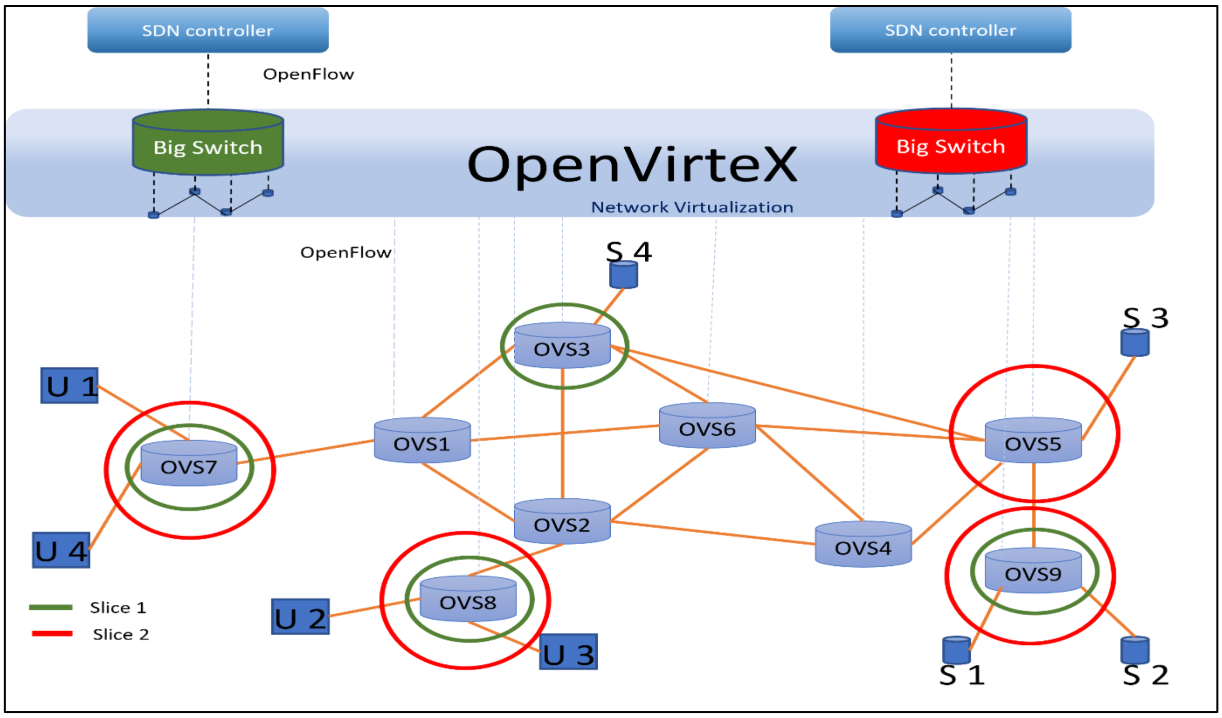


Figure 3.6: Virtual Big Switch Architecture

3.2.3 3rd Approach

Finally, another experiment is taken into account for realizing the attachment of BGP network over SDN based data plane. Network infrastructure for this experiment is described in Figure 3.6. There are five autonomous system interconnected to each other over BGP protocol and SDN technique. The entire data plane of the SDN network is considered as a single internal autonomous system, which is given as AS number of 100. Rest of the four autonomous system with AS number 200, 300, 400 and 500 is connected from their respective external router to Open vSwitches 1,6,7 and 8 respectively.

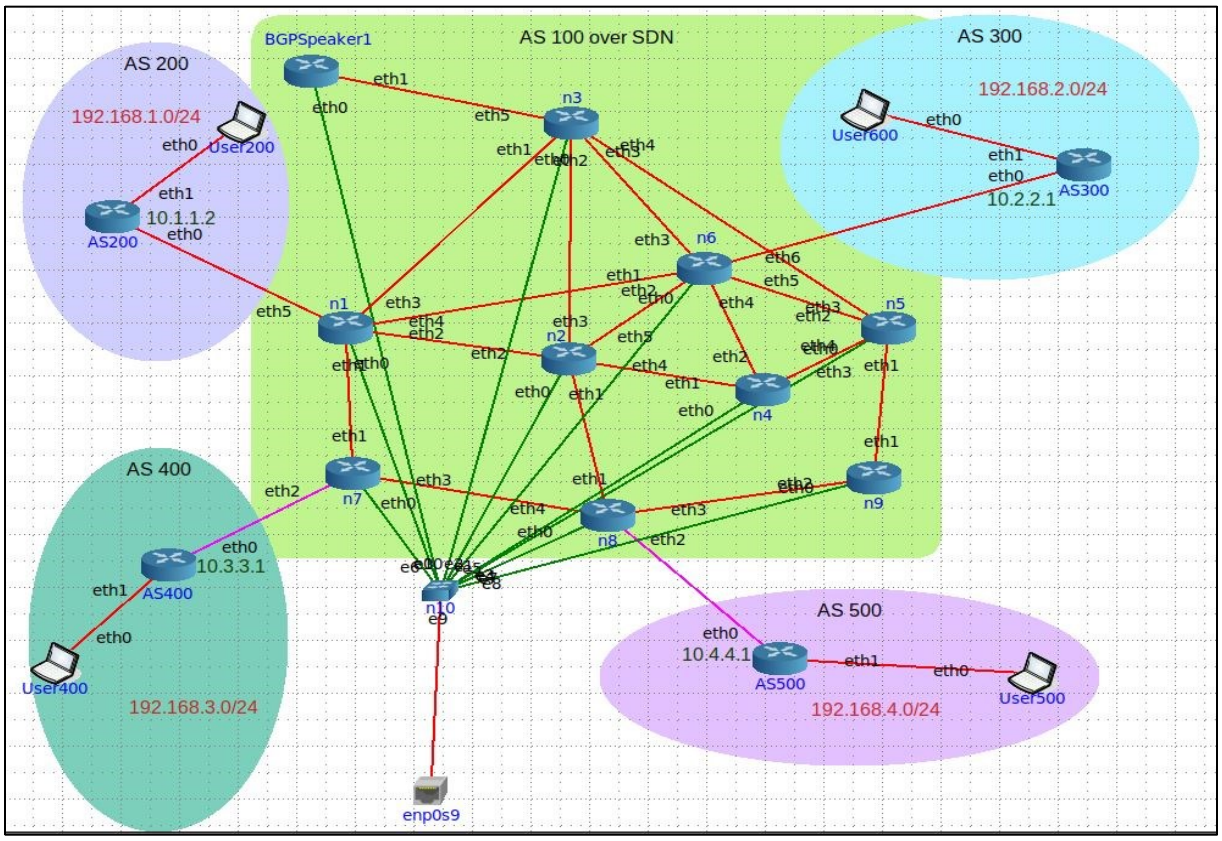


Figure 3.7: Physical Network design controlled by ONOS SDN controller

ONOS SDN controller runs on the same machine which runs CORE emulator, and all the nine Open vSwitches are connected to the ONOS controller from their eth0 interface. One extra router within the SDN data plane connected to the Open vSwitch 3 is providing the functionality of internal BGP speaker for the SDN network. The Internal BGP speaker is connected to the SDN controller through iBGP peering using the TCP port 2000, which is default for ONOS controller. ONOS controller application creates intents, which is the border switches connected to external router of other autonomous systems. These intents allow the external BGP routers to establish a eBGP peer using physical devices of the SDN data plane to the BGP speakers that reside in the data plane. The creation of those intents's network configuration is done by the REST API with a json information. Inside the json information, there is a section called port, which defines the IP addresses and mac address of the intents by mentioning the DPID of Open vSwitch and port number connected to the border router of other ASes. Another section in the json information is called app, is used to provide information of internal BGP speaker. In this section, connection point of BGP speaker to the data plane, which is switch 3 and external peering connection addresses are indicated.

The internal BGP speakers take advantage of eBGP peering to share BGP route information with the border routers of the neighboring external networks, and iBGP peering to propagate that information to ONOS application instances.

4 Project Implementation

4.1 1st Approach Implementation

To implement the experiment using RYU SDN controller, Linux Ubuntu version 18.04 LTS is considered. Installation process of CORE emulator can be found in [19]. Then RYU controller has been installed from this source [18] within the same Linux machine.

Firstly, after starting CORE emulator service in the machine, a file called core_ryu_design.imn, which can be found in 'project_ryu' folder is opened in core-gui.

The CORE emulated network is connected to the host linux machine by forming a bridge, that includes a interface of the local host. The sub-network of the bridge is 10.0.0.0/24. The bridge interface's IP address is defined as 10.0.0.254/24. All the emulated Open vSwitches are in the same subnet of the bridge network. This connection is executed with some bash commands given in Figure 4.1 from CORE session hooks by creating a runtime-hook.sh file.

```
#!/bin/sh
# session hook script; write commands here to execute on the host at the
# specified state
BRIFNAME=$(find /sys/devices/virtual/net -name 'enp0s9' | awk -F '[/:]' '{print $6}')
ip addr add 10.0.0.254/24 dev $BRIFNAME
```

Figure 4.1 Runtime hook for connecting emulated network to the hosted machine

During installation of RYU controller, the GUI interface for RYU controller is not come combined with RYU controller. So, the GUI interface known as flowmanager has to be cloned using git command depicted in Figure 4.2.

```
"git clone https://github.com/martimy/flowmanager"
```

Figure 4.2: git command to clone GUI for RYU controller

The RYU controller with custom application for controlling the designed data plane in CORE emulator starts by opening a terminal in 'project_ryu' folder, and with the executed command presented in Figure 4.3.

```
"ryu-manager ~/flowmanager/flowmanager.py core_ryu_application.py"
```

Figure 4.3: Starting RYU SDN controller

The emulation of the physical network initiates after starting it from core-gui, and the result of connected switches to the controller can be seen in the terminal where RYU is running, and also in the GUI interface of RYU. From core-gui, ping testing can be observed for different slices. To test the SSH and Webservice, a terminal is opened in one of the users, and ssh or curl command to Server1 is executed in the experiment. Afterwards to check the flow table of the Open vSwitchs from the GUI interface of RYU controller, flow manager is opened in a browser with http://localhost:8080/home/index.html.

4.2 2nd Approach Implementation

In the 2nd experiment, two linux machines are used, one is to provide the service of CORE emulator and another one is for providing the service of OpenVirteX hypervisor. After starting CORE emulator in the same linux machine of 1st approach, core_openvirtex_design.imn file is opened, that is in ‘openvirtex_project’ folder. As the emulated network and the controller are in two different machines, the connectivity is slight different than previous approach. The bridge network is created similar way of the previous experiment using 10.0.0.0/24 subnet by defining the bridge interface’s IP address 10.0.0.254/24. But, for reaching other linux machine which hosts the controller, ipv4 forwarding is enabled. Additionally, default route service is enabled in every Open vSwitch to 10.0.0.254/24 IP address. Furthermore, the interface that connects CORE hosted machine and controller hosted machine, is masqueraded, so that Open vSwitches in emulator can reach the controller. This procedure is accomplished in the runtime_hook.sh file shown in Figure 4.4 from CORE emulator.

```
#!/bin/sh
# session hook script; write commands here to execute on the host at the
# specified state
BRIFNAME=$(find /sys/devices/virtual/net -name 'enp0s9' | awk -F '[/:]' '{print $6}')
ip addr add 10.0.0.254/24 dev $BRIFNAME
echo 1 > /proc/sys/net/ipv4/ip_forward
iptables -t nat -A POSTROUTING -o enp0s8 -j MASQUERADE
```

Figure 4.4: Runtime hook for connecting emulated network to controller machine

Later on, the emulated network runs by starting the network from core-gui, but still the controller cannot reach the switches in the emulated network because of those networks are not defined in the routing table of the controller hosted machine. By executing the below bash script in the controller hosted machine, it creates the route to reach to the emulated network from the controller. The script called ‘route_core.sh’ which is described in Figure 4.5 is located in ‘openvirtex_project’ folder.

```
#!/bin/sh
route add -net 10.0.0.254 netmask 255.255.255.255 dev eth1
route add -net 10.0.0.0 netmask 255.255.255.0 gw 10.0.0.254
```

Figure 4.5: Route add script

A linux machine with pre-installed OpenVirteX and Floodlight controller is found in [17]. In the pre-installed OpenVirteX machine, by opening a terminal in ‘OpenVirtex/scripts/’ folder, command in Figure 4.6 is executed to run the hypervisior.

```
"sh ovx.sh"
```

Figure 4.6: Starting OpenVirteX hypervisor

It is seen that OpenVirtex is connected with emulated topology from the terminal log. Then by opening a terminal in 'OpenVirtex/utils/' folder of the OpenVirteX hosted machine, the first command shown in Figure 4.7 is executed for starting a embedded environment of REST API. Afterwards, file named slice1.json and slice2.json in 'openvirtex_project' of the project folder is copied to '/home/ovx' folder of the OpenVirteX hosted machine. Subsequently, a terminal is opened in '/home/ovx' folder of the OpenVirteX hosted machine. Two 'curl' commands depicted in Figure 4.7 are executed sequentially for establishing the virtual network through REST API.

```
"python embedder.py"

curl localhost:8000 -X POST -d @slice1.json

curl localhost:8000 -X POST -d @slice2.json
```

Figure 4.7: Commands for starting embedder and sending data with REST API

The reachability with ping testing is performed from core-gui within the defined slices.

The virtual network and slicing for tenants are observed from the floodlight controller's GUI, in OpenVirtex hosted machine by opening http://localhost:10001/ui/index.html and http://localhost:20001/ui/index.html in a browser for slice 1 and slice 2 respectively.

4.3 3rd Approach Implementation

The final experiment is performed on the similar linux machine which runs the first experiment. Both CORE emulator and ONOS SDN controller is hosted on this machine. ONOS SDN controller can be installed from this source [21]. In this project, ONOS version of 2.5.1 is used to control the data plane.

Firstly, ONOS controller is started by executing 'sudo /opt/onos-2.5.1/bin/onos-service start' command in the terminal. After a successful start of ONOS controller, the emulated network for this experiment is started by opening 'core_onos_design.imn' file from 'onos_project' in the project folder.

The connectivity of the emulated switches and BGP internal speaker to the controller is achieved in similar method of the first experiment, except enabling default route to 10.0.0.254/24 IP address in those switches. The reason behind enabling default route is that the ONOS controller, which runs from internal docker interface of the hosted machine, is in the different subnet from emulated network connected to host machine.

Four necessary applications in ONOS controller are activated to facilitate the controlling service. These applications can be activated either from ONOS GUI, which is found under http://localhost:8181/onos/ui/login.html or from the CLI of ONOS controller with the executed command in Figure 4.8.

The implementation of BGP peering is accomplished by activating BGP and zebra services in BGP internal speaker, and in the border router of external autonomous systems of the emulated network. BGP configuration for AS100, which is the SDN data plane, is performed in internal BGP speaker. The configuration for BGP neighboring for AS 100 is depicted in Figure 4.9.

```
onos> app activate org.onosproject.config

onos> app activate org.onosproject.proxyarp

onos> app activate org.onosproject.fwd

onos> app activate org.onosproject.sdnip
```

Figure 4.8: Activating ONOS applications

```
! BGP configuration

  router bgp 100
  bgp router-id 10.0.0.10
  redistribute connected
  timers bgp 3 9
  ! AS300
  neighbor 10.2.2.1 remote-as 300
  neighbor 10.2.2.1 ebgp-multihop
  neighbor 10.2.2.1 timers connect 5
  neighbor 10.2.2.1 advertisement-interval 5
  ! AS200
  neighbor 10.1.1.2 remote-as 200
  neighbor 10.1.1.2 ebgp-multihop
  neighbor 10.1.1.2 timers connect 5
  neighbor 10.1.1.2 advertisement-interval 5
  ! AS400
  neighbor 10.3.3.1 remote-as 400
  neighbor 10.3.3.1 ebgp-multihop
  neighbor 10.3.3.1 timers connect 5
  neighbor 10.3.3.1 advertisement-interval 5
  ! AS500
  neighbor 10.4.4.1 remote-as 500
  neighbor 10.4.4.1 ebgp-multihop
  neighbor 10.4.4.1 timers connect 5
  neighbor 10.4.4.1 advertisement-interval 5
  ! A6300
  neighbor 10.5.5.1 remote-as 600
  neighbor 10.5.5.1 ebgp-multihop
  neighbor 10.5.5.1 timers connect 5
  neighbor 10.5.5.1 advertisement-interval 5
  ! ONOS
  neighbor 10.0.0.254 remote-as 100
  neighbor 10.0.0.254 port 2000
  neighbor 10.0.0.254 timers connect 5
```

Figure 4.9: BGP configuration of Internal BGP Speaker AS100

BGP configuration in the external router of AS 200 can be defined as the configuration in Figure 4.10 for proper peering, and sharing the network information with the internal BGP speaker. All other ASes's border routers are configured in a similar way of AS 200, except changing the peering address and their own individual advertised network.

```
! BGP configuration
!
router bgp 200
  bgp router-id 10.1.1.2
  redistribute connected
  timers bgp 3 9
  neighbor 10.1.1.1 remote-as 100
  neighbor 10.1.1.1 ebgp-multihop
  neighbor 10.1.1.1 timers connect 5
  neighbor 10.1.1.1 advertisement-interval 5
  network 192.168.1.0/24
  network 10.1.1.0/24
```

Figure 4.10: BGP configuration of AS200

Finally, the file called 'onos_peer.json' is copied from the 'onos_project' project folder to the /tmp folder of the controller hosted machine, which is a network configuration file. Afterwards, the command in Figure 4.11 is used to form the network configuration within the SDN data plane using rest API. The successful reachability test within different ASes resided in different network is performed by ICMP ping from core-gui.

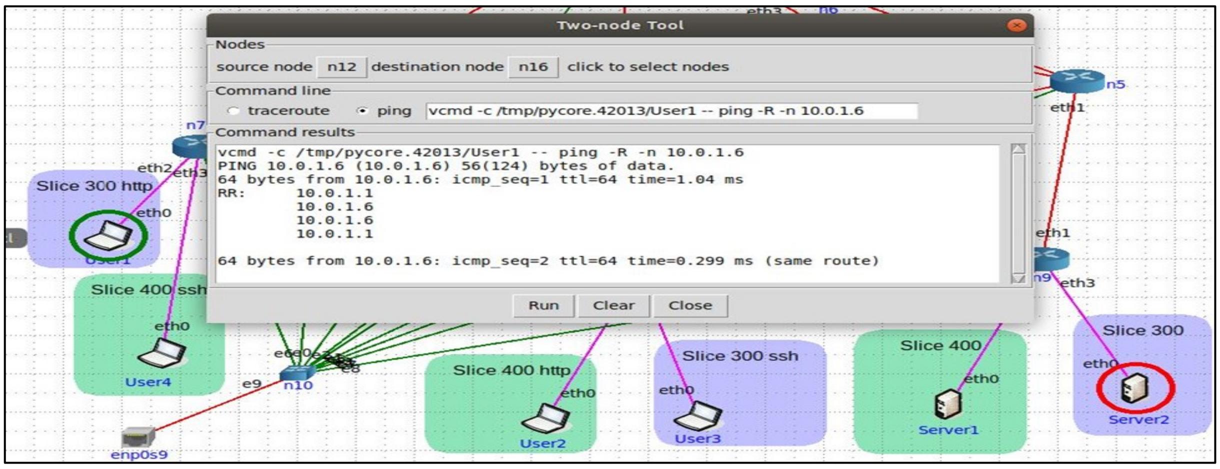
```
curl --user onos:rocks -X POST -H "Content-Type: application/json"
http://172.17.0.1:8181/onos/v1/network/configuration/ -d @/tmp/onos_peer.json
```

Figure 4.11: API command to build the network configuration

5 Project Result Analysis

5.1 1st Approach Analysis

The objective of this experiment is to slice the network on the basis of mac addresses of the users controlled by the RYU controller. As a result, User1 can reach Server2 with ICMP ping which is shown in Figure 5.1.



Nodes
source node n12 destination node n16 click to select nodes

Command line
traceroute [✓ ping] vcmd -c /tmp/pycore.42013/User1 -- ping -R -n 10.0.1.6

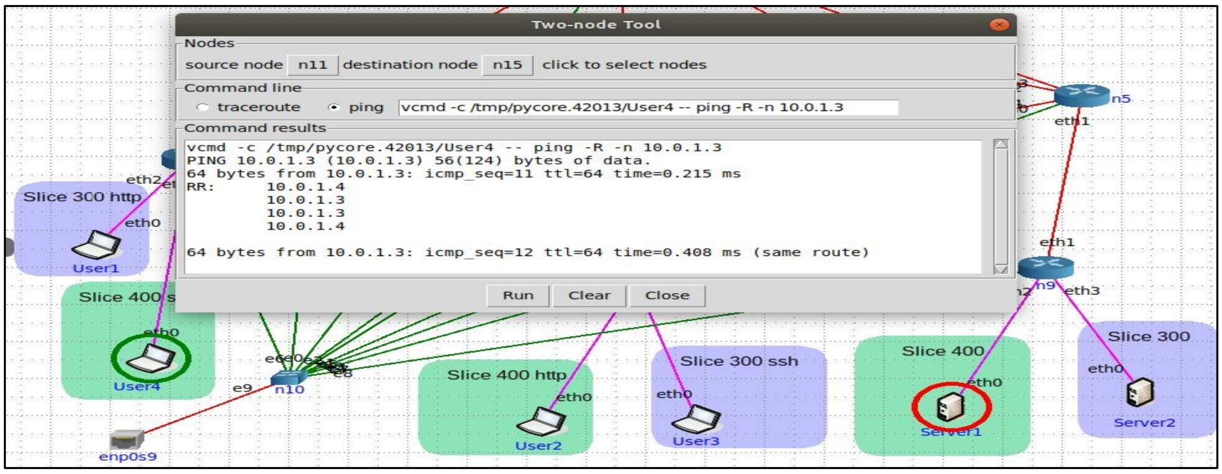
Command results

```
vcmd -c /tmp/pycore.42013/User1 -- ping -R -n 10.0.1.6
PING 10.0.1.6 (10.0.1.6) 56(124) bytes of data.
64 bytes from 10.0.1.6: icmp_seq=1 ttl=64 time=1.04 ms
RR:     10.0.1.1
        10.0.1.6
        10.0.1.6
        10.0.1.1

64 bytes from 10.0.1.6: icmp_seq=2 ttl=64 time=0.299 ms (same route)
```

Figure 5.1: Ping test between User1 and Server2

Similarly, from the other slice’s member Server1 is reachable with ICMP ping from User4. The ping testing is presented in Figure 5.2. The successful reachability test is also tested with other members from same slice. Testing is also done to check if any member of a slice can ping the other member from another slice. But it is successfully observed that this connectivity is not possible by the control logic of the controller, which is also the concept of this experiment.



Nodes
source node n11 destination node n15 click to select nodes

Command line
traceroute [✓ ping] vcmd -c /tmp/pycore.42013/User4 -- ping -R -n 10.0.1.3

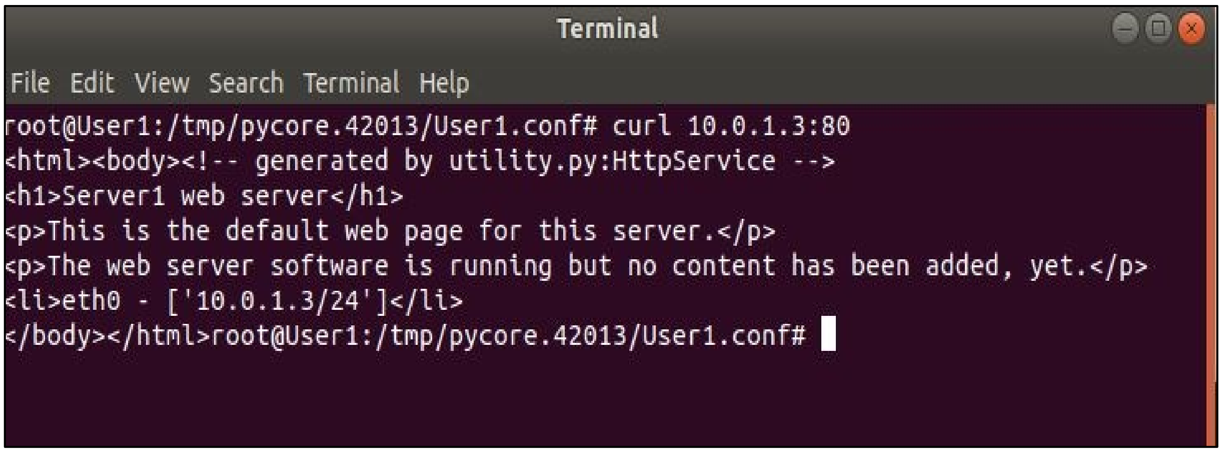
Command results

```
vcmd -c /tmp/pycore.42013/User4 -- ping -R -n 10.0.1.3
PING 10.0.1.3 (10.0.1.3) 56(124) bytes of data.
64 bytes from 10.0.1.3: icmp_seq=11 ttl=64 time=0.215 ms
RR:     10.0.1.4
        10.0.1.3
        10.0.1.3
        10.0.1.4

64 bytes from 10.0.1.3: icmp_seq=12 ttl=64 time=0.408 ms (same route)
```

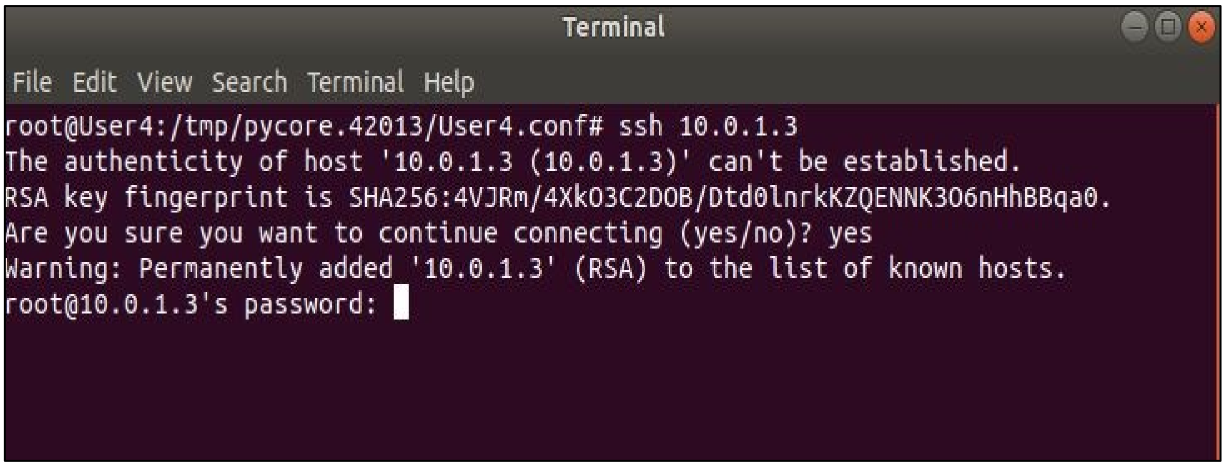
Figure 5.2: Ping test between User4 and Server1

Additionally, in this experiment, service based slicing is also performed. Server1 hosts a web server and a SSH server. According to the objective, web service can be utilized by the users in Slice 600, and users within slice 700 take advantage of SSH service. Therefore, User1 which is a member of Slice 600, can effectively get the service from web server which is demonstrated in Figure 5.3. Figure 5.4. provides the successful reachability of SSH server form User4, which is the customer in slice 700. In accordance with the concept, User3 and User4 cannot get the service from the web server, and on the other hand, User1 and User2 cannot connect with the SSH server.



```
root@User1:/tmp/pycore.42013/User1.conf# curl 10.0.1.3:80
<html><body><!-- generated by utility.py:HttpService -->
<h1>Server1 web server</h1>
<p>This is the default web page for this server.</p>
<p>The web server software is running but no content has been added, yet.</p>
<li>eth0 - ['10.0.1.3/24']</li>
</body></html>root@User1:/tmp/pycore.42013/User1.conf#
```

Figure 5.3: Web service reachability form User1

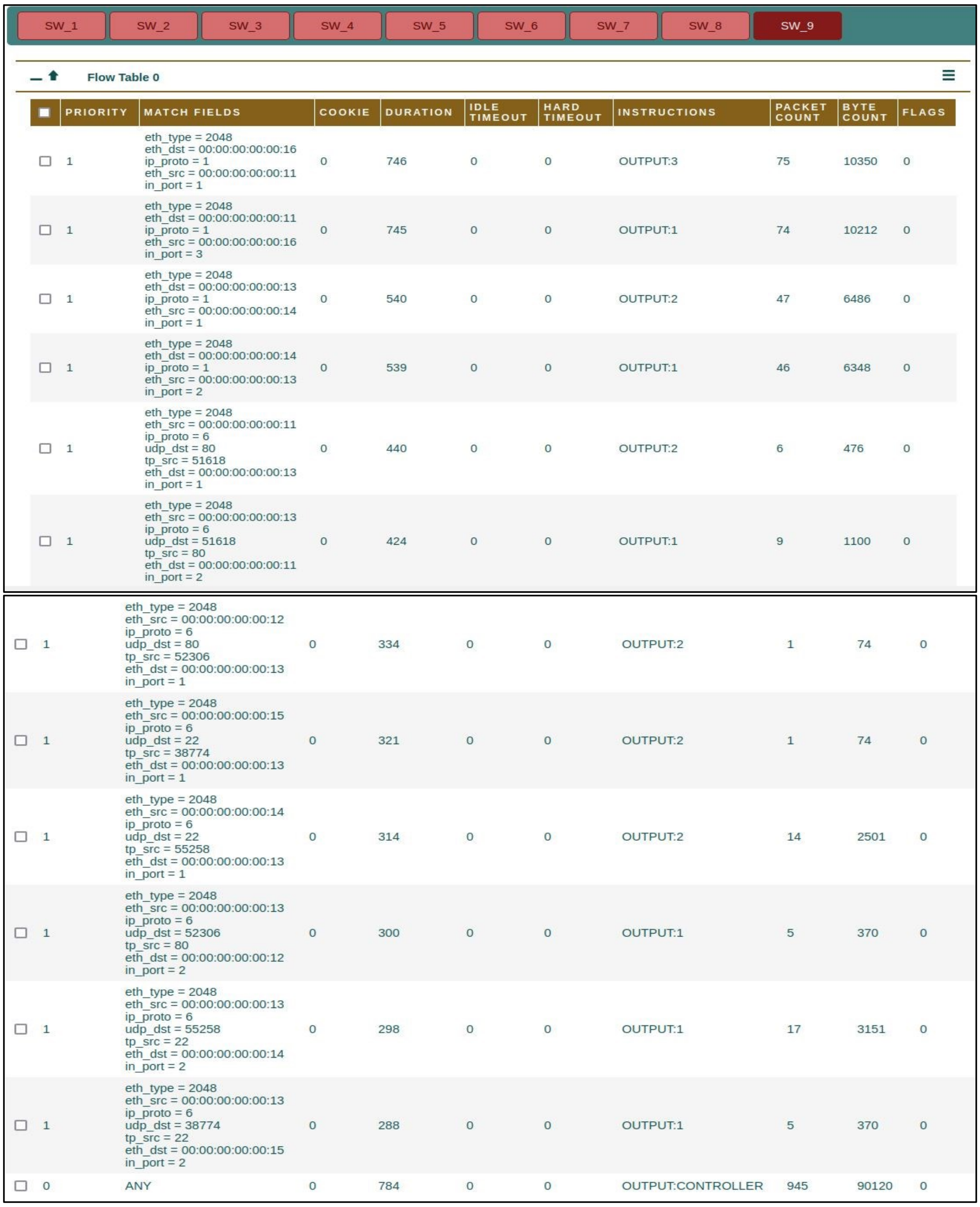


```
root@User4:/tmp/pycore.42013/User4.conf# ssh 10.0.1.3
The authenticity of host '10.0.1.3 (10.0.1.3)' can't be established.
RSA key fingerprint is SHA256:4VJRm/4XkO3C2DOB/Dtd0lnrkKZQENNK3O6nHhBBqa0.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.0.1.3' (RSA) to the list of known hosts.
root@10.0.1.3's password:
```

Figure 5.4: SSH server reachability form User4

Figure 5.5 summarizes the flow table of switch 9, which gives detail analysis of the routing of packets coming to and going from switch 9. Compliant with the concept of first slicing, mac addresses of User4 and User2 should get matched with the mac address of server1, and mac addresses of User1 and User3 should be matched with Server2. This can easily identified from the flow table of switch 9. Moreover, in the flow table, TCP port 80 which is used for providing web service is matched with the mac addresses of User1 and Server3 or User2 and Server3, which is the logic for Slice 600. But SSH service on TCP port 22 get matched with the mac addresses of

User3 and Server1 or User4 and Server1 because of the ruling of Slice 700. As a consequence of the concept behind service based slicing of this experiment, no matching is found other than the mentioned matching in the flow table.



☐	PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS
☐	1	eth_type = 2048 eth_dst = 00:00:00:00:00:16 ip_proto = 1 eth_src = 00:00:00:00:00:11 in_port = 1	0	746	0	0	OUTPUT:3	75	10350	0
☐	1	eth_type = 2048 eth_dst = 00:00:00:00:00:11 ip_proto = 1 eth_src = 00:00:00:00:00:16 in_port = 3	0	745	0	0	OUTPUT:1	74	10212	0
☐	1	eth_type = 2048 eth_dst = 00:00:00:00:00:13 ip_proto = 1 eth_src = 00:00:00:00:00:14 in_port = 1	0	540	0	0	OUTPUT:2	47	6486	0
☐	1	eth_type = 2048 eth_dst = 00:00:00:00:00:14 ip_proto = 1 eth_src = 00:00:00:00:00:13 in_port = 2	0	539	0	0	OUTPUT:1	46	6348	0
☐	1	eth_type = 2048 eth_src = 00:00:00:00:00:11 ip_proto = 6 udp_dst = 80 tp_src = 51618 eth_dst = 00:00:00:00:00:13 in_port = 1	0	440	0	0	OUTPUT:2	6	476	0
☐	1	eth_type = 2048 eth_src = 00:00:00:00:00:13 ip_proto = 6 udp_dst = 51618 tp_src = 80 eth_dst = 00:00:00:00:00:11 in_port = 2	0	424	0	0	OUTPUT:1	9	1100	0
☐	1	eth_type = 2048 eth_src = 00:00:00:00:00:12 ip_proto = 6 udp_dst = 80 tp_src = 52306 eth_dst = 00:00:00:00:00:13 in_port = 1	0	334	0	0	OUTPUT:2	1	74	0
☐	1	eth_type = 2048 eth_src = 00:00:00:00:00:15 ip_proto = 6 udp_dst = 22 tp_src = 38774 eth_dst = 00:00:00:00:00:13 in_port = 1	0	321	0	0	OUTPUT:2	1	74	0
☐	1	eth_type = 2048 eth_src = 00:00:00:00:00:14 ip_proto = 6 udp_dst = 22 tp_src = 55258 eth_dst = 00:00:00:00:00:13 in_port = 1	0	314	0	0	OUTPUT:2	14	2501	0
☐	1	eth_type = 2048 eth_src = 00:00:00:00:00:13 ip_proto = 6 udp_dst = 52306 tp_src = 80 eth_dst = 00:00:00:00:00:12 in_port = 2	0	300	0	0	OUTPUT:1	5	370	0
☐	1	eth_type = 2048 eth_src = 00:00:00:00:00:13 ip_proto = 6 udp_dst = 55258 tp_src = 22 eth_dst = 00:00:00:00:00:14 in_port = 2	0	298	0	0	OUTPUT:1	17	3151	0
☐	1	eth_type = 2048 eth_src = 00:00:00:00:00:13 ip_proto = 6 udp_dst = 38774 tp_src = 22 eth_dst = 00:00:00:00:00:15 in_port = 2	0	288	0	0	OUTPUT:1	5	370	0
☐	0	ANY	0	784	0	0	OUTPUT:CONTROLLER	945	90120	0

Figure 5.5: Flow table of switch 9

5.2 2nd Approach Analysis

For analyzing this experimental concept, the key factor is to deal with the virtual big switches. User1, User2, Server2 and Server4 share the same virtual network, so these network devices creates a single big switch which is depicted in Figure 5.6. On the contrary, another big switch is allocated to User3, User4, Server1 and Server4, that is illustrated in Figure 5.7. So, the communication within the slice is done by simple switching system.

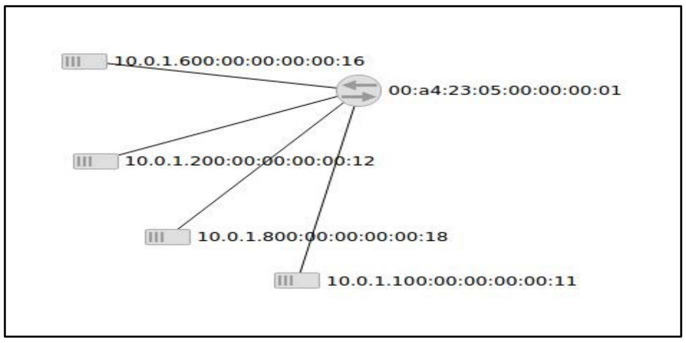


Figure 5.6: Virtual big switch for tenant ID 1

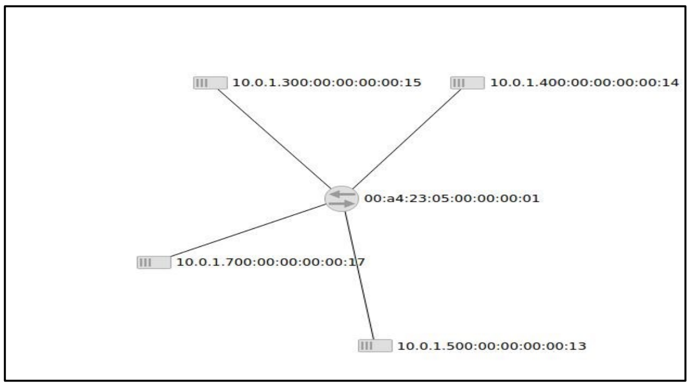
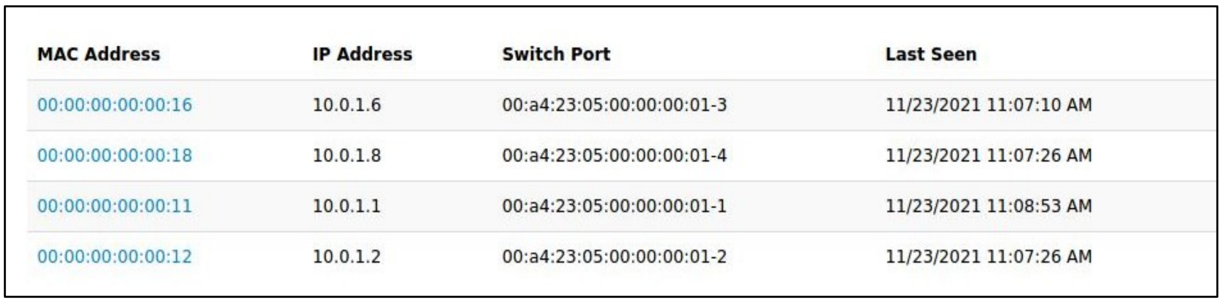


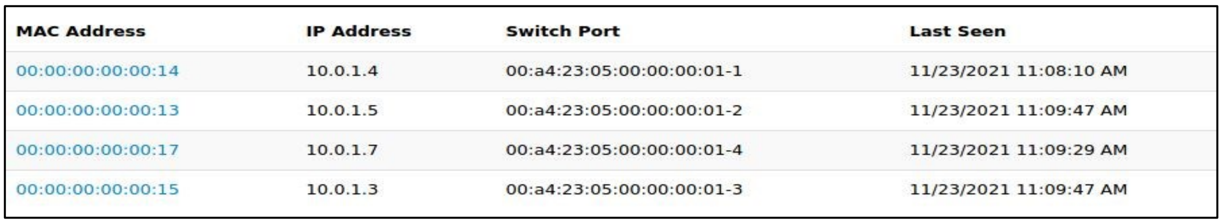
Figure 5.7: Virtual big switch for tenant ID 2

Figure 5.8 and Figure 5.9 describe the details of the hosts connected to their virtual switches for slice 1 and slice 2 respectively.



MAC Address	IP Address	Switch Port	Last Seen
00:00:00:00:00:16	10.0.1.6	00:a4:23:05:00:00:00:01-3	11/23/2021 11:07:10 AM
00:00:00:00:00:18	10.0.1.8	00:a4:23:05:00:00:00:01-4	11/23/2021 11:07:26 AM
00:00:00:00:00:11	10.0.1.1	00:a4:23:05:00:00:00:01-1	11/23/2021 11:08:53 AM
00:00:00:00:00:12	10.0.1.2	00:a4:23:05:00:00:00:01-2	11/23/2021 11:07:26 AM

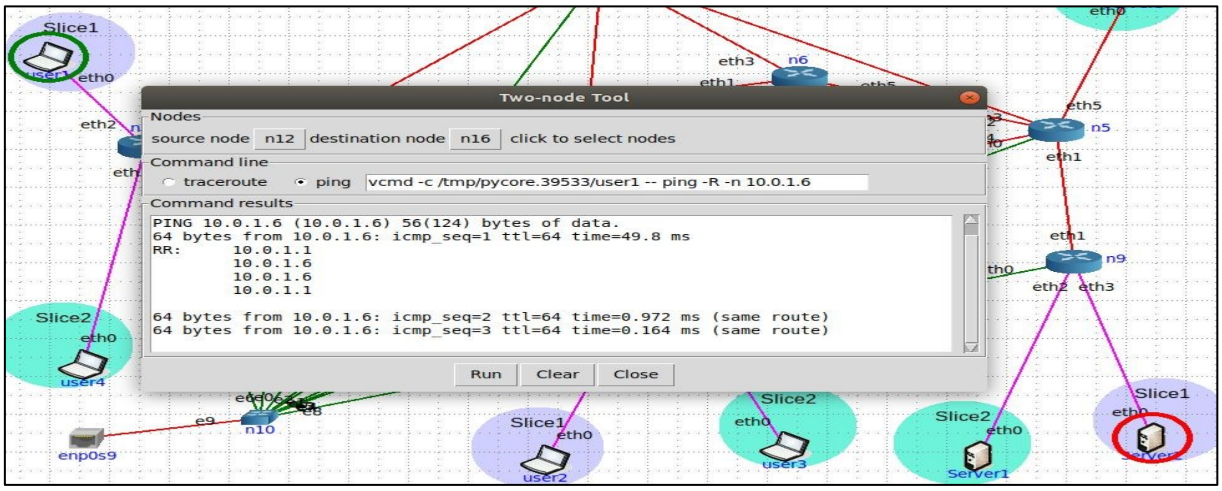
Figure 5.8: Hosts details for tenant ID 1



MAC Address	IP Address	Switch Port	Last Seen
00:00:00:00:00:14	10.0.1.4	00:a4:23:05:00:00:00:01-1	11/23/2021 11:08:10 AM
00:00:00:00:00:13	10.0.1.5	00:a4:23:05:00:00:00:01-2	11/23/2021 11:09:47 AM
00:00:00:00:00:17	10.0.1.7	00:a4:23:05:00:00:00:01-4	11/23/2021 11:09:29 AM
00:00:00:00:00:15	10.0.1.3	00:a4:23:05:00:00:00:01-3	11/23/2021 11:09:47 AM

Figure 5.9: Hosts details for tenant ID 2

All the member from the first virtual network with tenant id 1 is reachable to each other through ICMP ping. Ping test from User1 to Server2 is presented in below Figure 5.10.



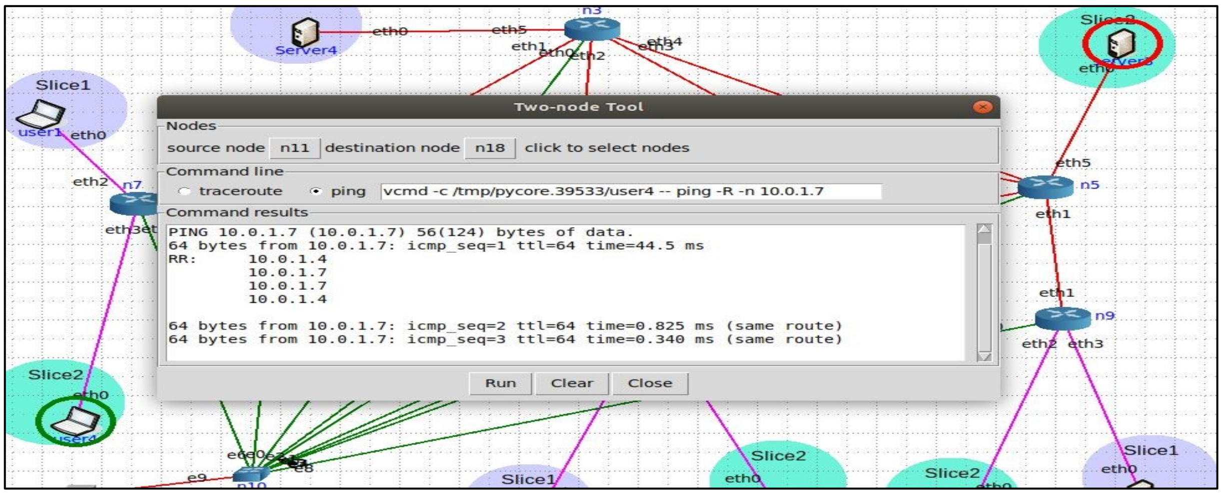
traceroute [✓ ping] vcmd -c /tmp/pycore.39533/user1 -- ping -R -n 10.0.1.6

```
PING 10.0.1.6 (10.0.1.6) 56(124) bytes of data.
64 bytes from 10.0.1.6: icmp_seq=1 ttl=64 time=49.8 ms
RR:     10.0.1.1
        10.0.1.6
        10.0.1.6
        10.0.1.1

64 bytes from 10.0.1.6: icmp_seq=2 ttl=64 time=0.972 ms (same route)
64 bytes from 10.0.1.6: icmp_seq=3 ttl=64 time=0.164 ms (same route)
```

Figure 5.10: Ping test between User1 and Server2

From the other slice, the test is performed from User4 to Server3 which is given in Figure 5.11.



traceroute [✓ ping] vcmd -c /tmp/pycore.39533/user4 -- ping -R -n 10.0.1.7

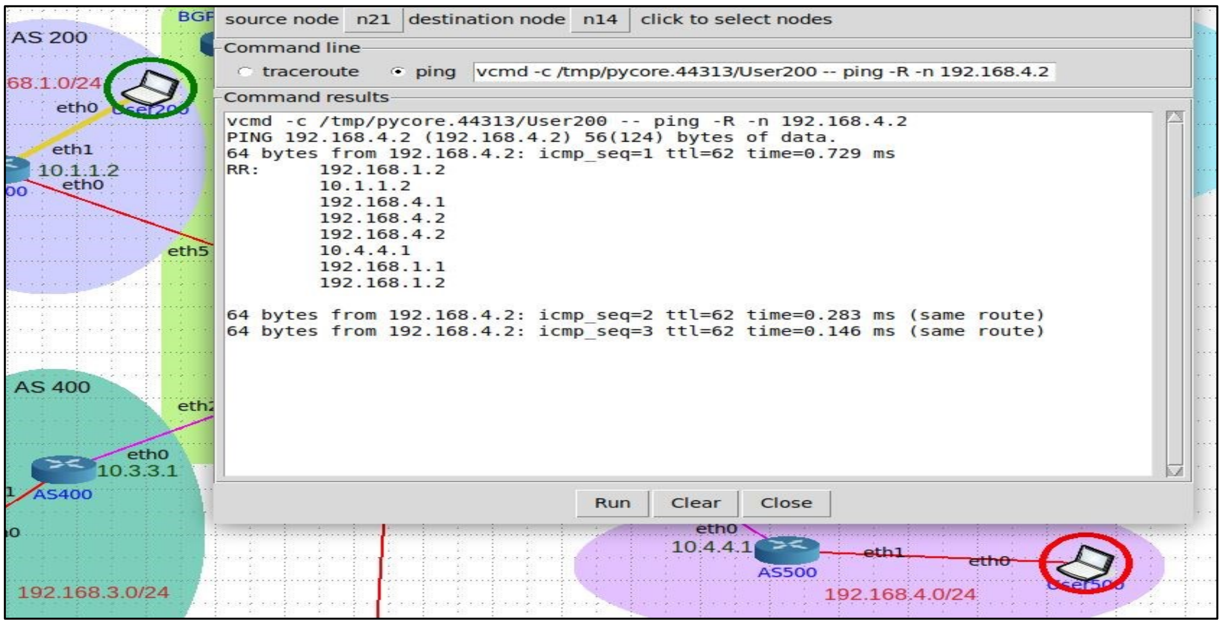
```
PING 10.0.1.7 (10.0.1.7) 56(124) bytes of data.
64 bytes from 10.0.1.7: icmp_seq=1 ttl=64 time=44.5 ms
RR:     10.0.1.4
        10.0.1.7
        10.0.1.7
        10.0.1.4

64 bytes from 10.0.1.7: icmp_seq=2 ttl=64 time=0.825 ms (same route)
64 bytes from 10.0.1.7: icmp_seq=3 ttl=64 time=0.340 ms (same route)
```

Figure 5.11: Ping test between User4 and Server3

5.3 3rd Approach Analysis

After a successful implementation of the experiment, reachability from network of one autonomous system to the network of other autonomous system is observed. The reachability test that performed between AS 200 and AS 500 is demonstrated in Figure 5.12, and also with Figure 5.13, test between AS 300 and AS 400 is checked.



source node n21 destination node n14 click to select nodes

Command line

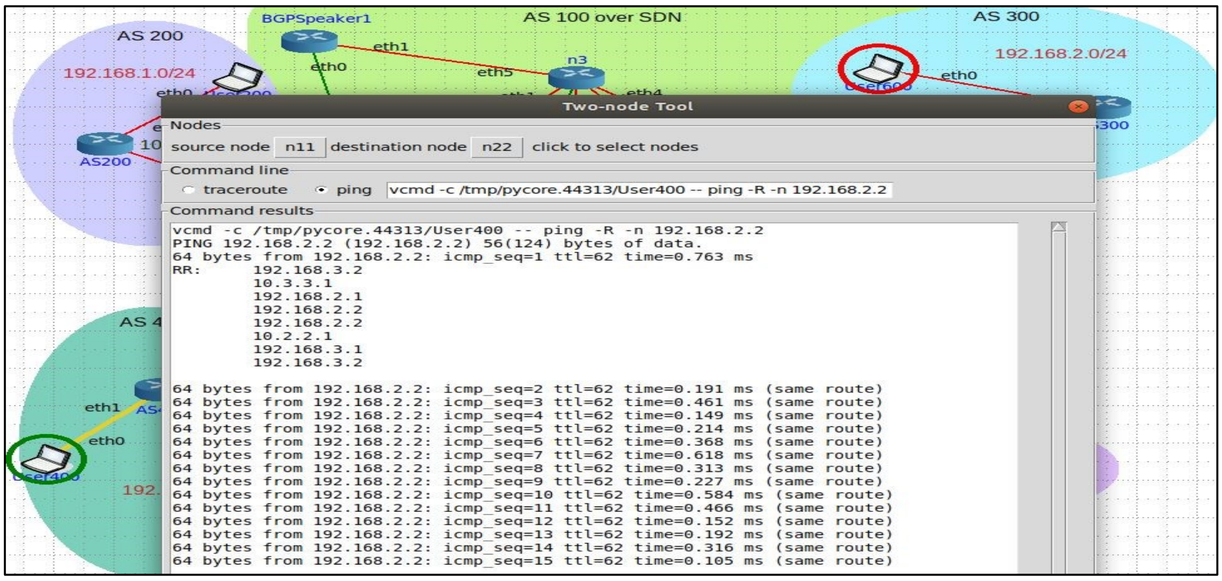
○ traceroute [✓ ping] vcmd -c /tmp/pycore.44313/User200 -- ping -R -n 192.168.4.2

Command results

```
vcmd -c /tmp/pycore.44313/User200 -- ping -R -n 192.168.4.2
PING 192.168.4.2 (192.168.4.2) 56(124) bytes of data.
64 bytes from 192.168.4.2: icmp_seq=1 ttl=62 time=0.729 ms
RR:     192.168.1.2
        10.1.1.2
        192.168.4.1
        192.168.4.2
        192.168.4.2
        10.4.4.1
        192.168.1.1
        192.168.1.2

64 bytes from 192.168.4.2: icmp_seq=2 ttl=62 time=0.283 ms (same route)
64 bytes from 192.168.4.2: icmp_seq=3 ttl=62 time=0.146 ms (same route)
```

Figure 5.12: Ping test from AS 200 to AS 500



Two-node Tool

Nodes

source node n11 destination node n22 click to select nodes

Command line

○ traceroute [✓ ping] vcmd -c /tmp/pycore.44313/User400 -- ping -R -n 192.168.2.2

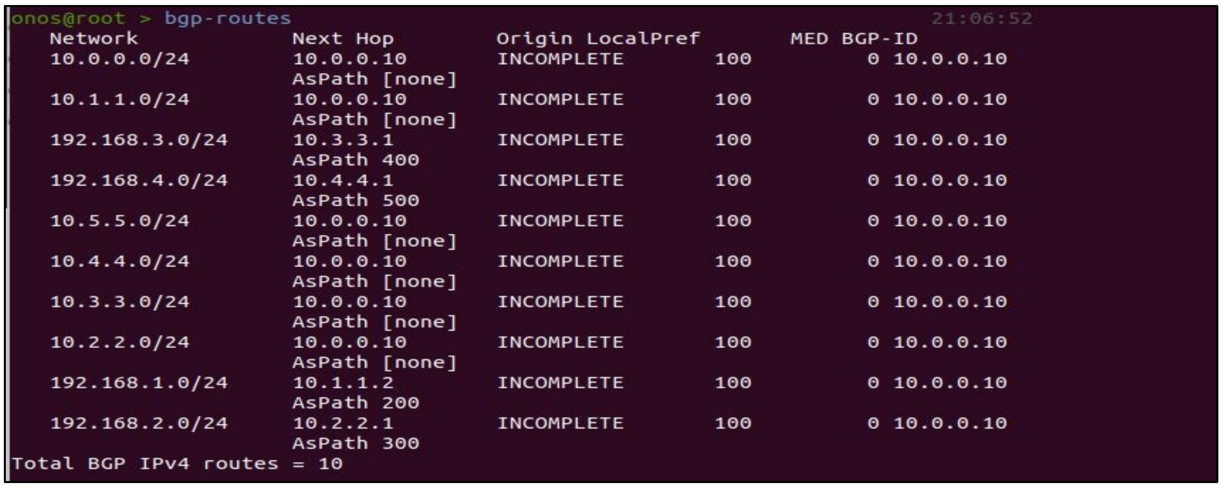
Command results

```
vcmd -c /tmp/pycore.44313/User400 -- ping -R -n 192.168.2.2
PING 192.168.2.2 (192.168.2.2) 56(124) bytes of data.
64 bytes from 192.168.2.2: icmp_seq=1 ttl=62 time=0.763 ms
RR:     192.168.3.2
        10.3.3.1
        192.168.2.1
        192.168.2.2
        192.168.2.2
        10.2.2.1
        192.168.3.1
        192.168.3.2

64 bytes from 192.168.2.2: icmp_seq=2 ttl=62 time=0.191 ms (same route)
64 bytes from 192.168.2.2: icmp_seq=3 ttl=62 time=0.461 ms (same route)
64 bytes from 192.168.2.2: icmp_seq=4 ttl=62 time=0.149 ms (same route)
64 bytes from 192.168.2.2: icmp_seq=5 ttl=62 time=0.214 ms (same route)
64 bytes from 192.168.2.2: icmp_seq=6 ttl=62 time=0.368 ms (same route)
64 bytes from 192.168.2.2: icmp_seq=7 ttl=62 time=0.618 ms (same route)
64 bytes from 192.168.2.2: icmp_seq=8 ttl=62 time=0.313 ms (same route)
64 bytes from 192.168.2.2: icmp_seq=9 ttl=62 time=0.227 ms (same route)
64 bytes from 192.168.2.2: icmp_seq=10 ttl=62 time=0.584 ms (same route)
64 bytes from 192.168.2.2: icmp_seq=11 ttl=62 time=0.466 ms (same route)
64 bytes from 192.168.2.2: icmp_seq=12 ttl=62 time=0.152 ms (same route)
64 bytes from 192.168.2.2: icmp_seq=13 ttl=62 time=0.192 ms (same route)
64 bytes from 192.168.2.2: icmp_seq=14 ttl=62 time=0.316 ms (same route)
64 bytes from 192.168.2.2: icmp_seq=15 ttl=62 time=0.105 ms (same route)
```

Figure 5.13: Ping test from AS 400 to AS 300

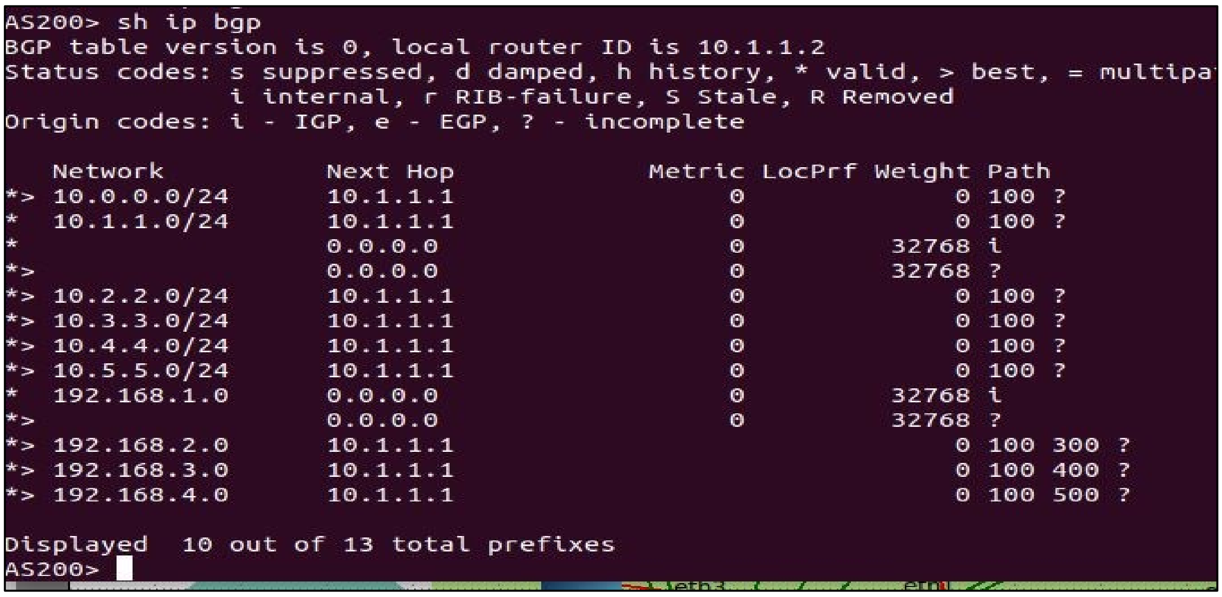
The outcome analysis can be examined by investigating the routing table from all the BGP enabled router. Firstly, the BGP routing table from ONOS controller that is described in Figure 5.14 shows that all the advertised subnets form other ASes are in the routing table of ONOS controller. Most importantly 192.168.1.0/24,192.168.2.0/24, 192.168.3.0/24 and 192.168.4.0/24 subnets from AS 200, AS 300, AS 400 and AS 500 respectively are seen in the routing table, which helps to reach those networks through SDN controller.



```
onos@root > bgp-routes                                                   21:06:52
   Network            Next Hop        Origin LocalPref       MED BGP-ID
   10.0.0.0/24        10.0.0.10       INCOMPLETE      100          0 10.0.0.10
                      AsPath [none]
   10.1.1.0/24        10.0.0.10       INCOMPLETE      100          0 10.0.0.10
                      AsPath [none]
   192.168.3.0/24     10.3.3.1        INCOMPLETE      100          0 10.0.0.10
                      AsPath 400
   192.168.4.0/24     10.4.4.1        INCOMPLETE      100          0 10.0.0.10
                      AsPath 500
   10.5.5.0/24        10.0.0.10       INCOMPLETE      100          0 10.0.0.10
                      AsPath [none]
   10.4.4.0/24        10.0.0.10       INCOMPLETE      100          0 10.0.0.10
                      AsPath [none]
   10.3.3.0/24        10.0.0.10       INCOMPLETE      100          0 10.0.0.10
                      AsPath [none]
   10.2.2.0/24        10.0.0.10       INCOMPLETE      100          0 10.0.0.10
                      AsPath [none]
   192.168.1.0/24     10.1.1.2        INCOMPLETE      100          0 10.0.0.10
                      AsPath 200
   192.168.2.0/24     10.2.2.1        INCOMPLETE      100          0 10.0.0.10
                      AsPath 300
Total BGP IPv4 routes = 10
```

Figure 5.14: Routing table of ONOS controller in AS 100

Figure 5.15 explains the routing table of the egress router of AS 200. It can be seen that the subnetworks of other ASes are reachable having the next hop as internal BGP speaker of SDN network which has the IP address 10.1.1.1/24. Similarly, in the routing table of other router, next hop is the SDN network's BGP speaker.



```
AS200> sh ip bgp
BGP table version is 0, local router ID is 10.1.1.2
Status codes: s suppressed, d damped, h history, * valid, > best, = multipa
              i internal, r RIB-failure, S Stale, R Removed
Origin codes: i - IGP, e - EGP, ? - incomplete

   Network          Next Hop            Metric LocPrf Weight Path
*> 10.0.0.0/24      10.1.1.1                 0             0 100 ?
*  10.1.1.0/24      10.1.1.1                 0             0 100 ?
*                   0.0.0.0                  0         32768 i
*>                  0.0.0.0                  0         32768 ?
*> 10.2.2.0/24      10.1.1.1                 0             0 100 ?
*> 10.3.3.0/24      10.1.1.1                 0             0 100 ?
*> 10.4.4.0/24      10.1.1.1                 0             0 100 ?
*> 10.5.5.0/24      10.1.1.1                 0             0 100 ?
*  192.168.1.0      0.0.0.0                  0         32768 i
*>                  0.0.0.0                  0         32768 ?
*> 192.168.2.0      10.1.1.1                               0 100 300 ?
*> 192.168.3.0      10.1.1.1                               0 100 400 ?
*> 192.168.4.0      10.1.1.1                               0 100 500 ?

Displayed  10 out of 13 total prefixes
AS200>
```

Figure 5.15: Routing table border router in AS 200

The flow table for switch 7 which is connected to the external router of AS 400 is given in Figure 5.16. The flow table provides the data with BGP connection matching that includes TCP port 179, source and destination IP address of other subnets, and with instruction to send the packet to proper switch port to reach other ASes over the SDN data plane.

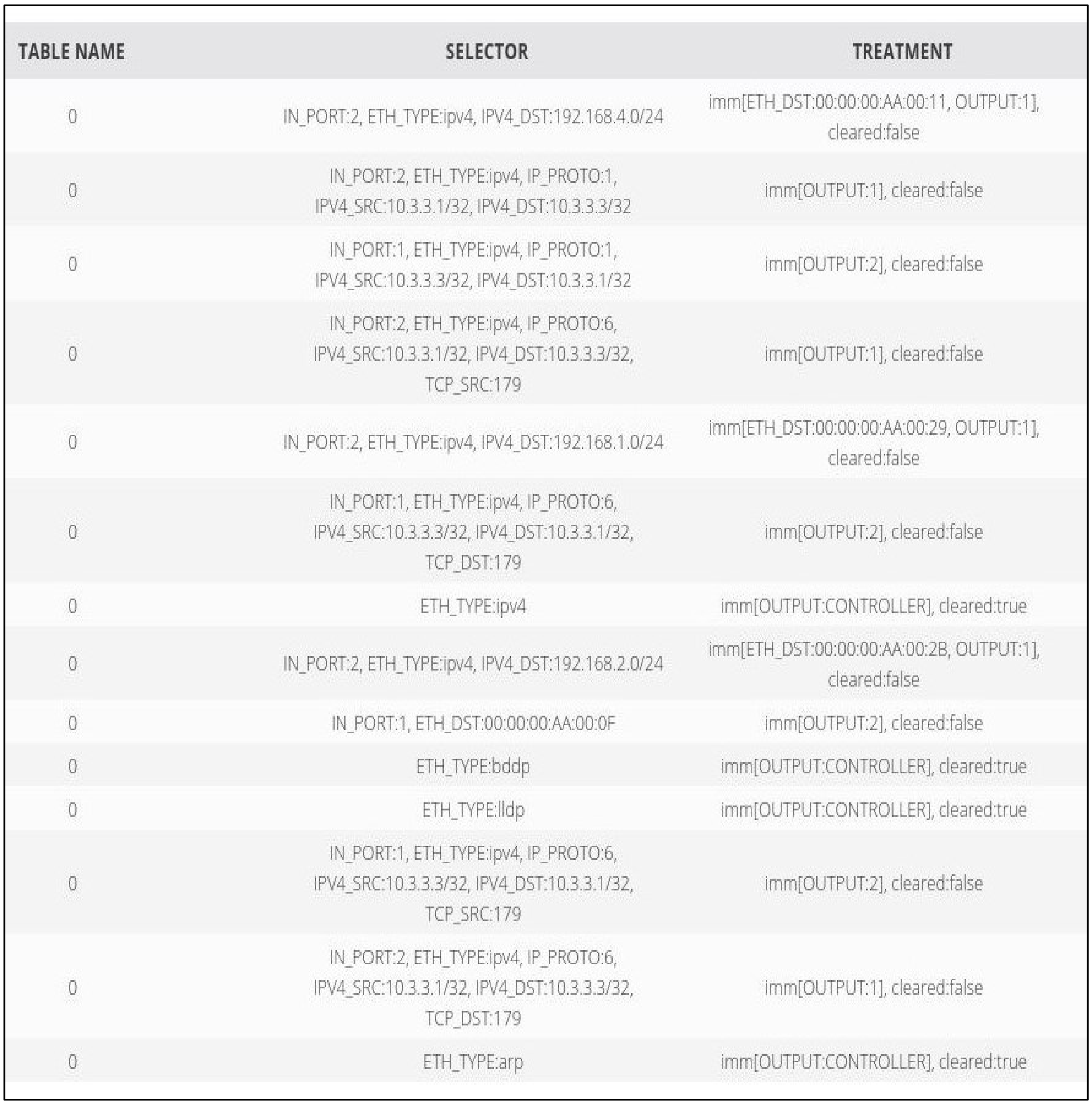


TABLE NAME	SELECTOR	TREATMENT
0	IN_PORT:2, ETH_TYPE:ipv4, IPV4_DST:192.168.4.0/24	imm[ETH_DST:00:00:00:AA:00:11, OUTPUT:1], cleared:false
0	IN_PORT:2, ETH_TYPE:ipv4, IP_PROTO:1, IPV4_SRC:10.3.3.1/32, IPV4_DST:10.3.3.3/32	imm[OUTPUT:1], cleared:false
0	IN_PORT:1, ETH_TYPE:ipv4, IP_PROTO:1, IPV4_SRC:10.3.3.3/32, IPV4_DST:10.3.3.1/32	imm[OUTPUT:2], cleared:false
0	IN_PORT:2, ETH_TYPE:ipv4, IP_PROTO:6, IPV4_SRC:10.3.3.1/32, IPV4_DST:10.3.3.3/32, TCP_SRC:179	imm[OUTPUT:1], cleared:false
0	IN_PORT:2, ETH_TYPE:ipv4, IPV4_DST:192.168.1.0/24	imm[ETH_DST:00:00:00:AA:00:29, OUTPUT:1], cleared:false
0	IN_PORT:1, ETH_TYPE:ipv4, IP_PROTO:6, IPV4_SRC:10.3.3.3/32, IPV4_DST:10.3.3.1/32, TCP_DST:179	imm[OUTPUT:2], cleared:false
0	ETH_TYPE:ipv4	imm[OUTPUT:CONTROLLER], cleared:true
0	IN_PORT:2, ETH_TYPE:ipv4, IPV4_DST:192.168.2.0/24	imm[ETH_DST:00:00:00:AA:00:2B, OUTPUT:1], cleared:false
0	IN_PORT:1, ETH_DST:00:00:00:AA:00:0F	imm[OUTPUT:2], cleared:false
0	ETH_TYPE:bddp	imm[OUTPUT:CONTROLLER], cleared:true
0	ETH_TYPE:lldp	imm[OUTPUT:CONTROLLER], cleared:true
0	IN_PORT:1, ETH_TYPE:ipv4, IP_PROTO:6, IPV4_SRC:10.3.3.3/32, IPV4_DST:10.3.3.1/32, TCP_SRC:179	imm[OUTPUT:2], cleared:false
0	IN_PORT:2, ETH_TYPE:ipv4, IP_PROTO:6, IPV4_SRC:10.3.3.1/32, IPV4_DST:10.3.3.3/32, TCP_DST:179	imm[OUTPUT:1], cleared:false
0	ETH_TYPE:arp	imm[OUTPUT:CONTROLLER], cleared:true

Figure 5.16: Flow table of switch 7

6 Conclusion and Future Work

SDN correspond to a modern, adaptable, and accessible architecture that enables the dynamic behaviour of network switches in complex and large-scale computer networks. This project work proposed SDN-based implementation of network, which is executed by utilizing various features of SDN concept. This work tries to study the SDN network implementation utilizing several SDN controller and analyse the core idea of SDN. Deployment of three different network infrastructures has broaden the learning process of SDN architecture. As SDN concept is a very large methodology, there are also lots of scope to research and deploy new thinking and innovation.

However, development of the project work can be extended by introducing Quality of Service (QoS) in a centralised manner, which is not observed throughout the project. Henceforward, taking the advantage of meter table of OpenFlow switches, and controlling by the SDN controller, QoS can be experimented in these networks. Bandwidth management is one of the crucial concern in traditional network strategy. Centralised network management by SDN can ease the managing of bandwidth by using the feature of Open vSwitches and OpenFlow protocol.

In addition, the concept of load balancing, which is a technique of enhancing network performance, delay minimization, is not examined in this work. Dynamic Load balancing can be achievable both for the data plane and for the control plane using SDN architecture. In future, by using group table's feature of OpenFlow switches, load balancing for data plane can be observed. Load balancer can increase the overall performance of the physical networks and decrease delay. Load balancing in the control plane can also become an issue if the controller has to control a very large amount of devices in the data plane. So, later on, the project can focus on taking a number of controllers in a cluster mode, which can improve the efficiency of a controller.

7 Abbreviations

0

…

5

5G	Fifth Generation

…

A

API	Application Programming Interface
ARP	Address Resolution Protocol
AS	Autonomous System

B

BGP	Border Gateway Protocol

C

CLI	Command-Line Interface
CORE	Common Open Research Emulator

D

DPID	Datapath Identifier

…

G

GUI	Graphical User Interface

H

HTTP	Hypertext Transfer Protocol

I

ICMP	Internet Control Message Protocol
IMUNES	Integrated Multi-protocol Network Emulator/Simulator
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6

…

L

LTS	Long-Term Support

…

O

ONF	Open Networking Foundation
ONOS	Open Network Operating System
OS	Operating System
OVS	Open vSwitch

P

PC	Personal Computer

…

R

REST	Representational State Transfer
RJ45	Registered Jack-45

S

SDN	Software Defined Network
SSH	Secure Shell

T

TCL	Tool Command Language
TCP	Transmission Control Protocol
TK	ToolKit
TLS	Transport Layer Security
TLV	Type-Length-Value

…

Z

…

8 References

1. Benson, T. et al (2009): Unraveling the complexity of network management, *in Proc. 6th USENIX Symp.Networked Syst. Design Implement*. pp. 335–348.

2. Kreutz, D. et al (2015): Software-Defined Networking: A Comprehensive Survey, *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14- 76.

3. Akyildiz, F. et al (2016): 'Research challenges for traffic engineering in software defined networks' *in IEEE Network*, vol. 30, no. 3, pp. 52-58, doi: 10.1109/MNET.2016.7474344.

4. Baskoro, F. et al (2019): Comparing LACP Implementation between Ryu and Opendaylight SDN Controller, *11th International Conference on Information Technology and Electrical Engineering (ICITEE)*, Pattaya, Thailand, pp. 1-4, doi: 10.1109/ICITEED.2019.8929986.

5. Ravindran, R. et al (2017): 5G-ICN: Delivering ICN services over 5G using network slicing *IEEE Communications Magazine*, vol. 55, no. 5, pp. 101-107.

6. Scano, D. et al (2020):Network Slicing in SDN Networks, *22nd International Conference on Transparent Optical Networks (ICTON)*, Bari, Italy, pp. 1-4, doi: 10.1109/ICTON51198.2020.9203184.

7. Feamster, N. et al (2014): The Road to SDN: An Intellectual History of Programmable Networks, *In: ACM Sigcomm Computer Communicattion* , pp. 87–98. issn: 01464833. doi: 10.1145/2602204.2602219.

8. Langenskiöld, T. (2017): Network Slicing using Switch Virtualization', Master's thesis, *School of Electrical Engineering*.

9. S. Denazis et al RFC 7426 (2015): *Software-Defined Networking (SDN): Layers and Architecture Terminology*, Tech. rep., doi: 10.17487/rfc7426.

10. Zavrak, S. and Iskefiyeli, M. (2017): A Feature-Based Comparison of SDN Emulation and Simulation Tools, *International Conference on Engineering Technologies (ICENTE'17)*.

11. Baskoro, F. et al (2019): LACP Experiment using Multiple Flow Table in Ryu SDN Controller, *2nd International Conference on Applied Information Technology and Innovation (ICAITI)*, Denpasar, Indonesia, pp. 51-55, doi: 10.1109/ICAITI48442.2019.8982149.

12. Open Networking Foundation (ONF) (2012): OpenFlow Switch Specifcation, Version 1.3.0 (Protocol version 0x04), *Open Networking Foundation (ONF),* 25 June.

13. Tao, W. et al (2005): A Selective Introduction to Border Gateway Protocol (BGP) Security Issues, *School of Computer Science*, Carleton University, Ottawa, Canada, 1 August.

14. Kuhn, R. et al (2017): Border Gateway Protocol Security, *Natl. Inst. Stand. Technol. Spec. Publ*. 800-54, 61 pages.

15. Ahrenholz, J. et al (2008): CORE: A real-time network emulator., *Military Communications Conference*, MILCOM, IEEE, doi: 10.1109/MILCOM.2008.4753614.

16. Operating Systems Lab., Korea University (2020a) : Libera and OpenVirteX projects, [online] https://openvirtex.com/documentation/architecture/overview/#1.1 [accessed: 25 August 2021].

17. Operating Systems Lab., Korea University (2020b) : Libera and OpenVirteX projects, [online] https://openvirtex.com/getting-started/installation/ [accessed: 25 August 2021].

18. RYU project team (2021): ryu Documentation, Ryubook documentation Release 4.34. [online] https://readthedocs.org/projects/ryu/downloads/pdf/latest/. [accessed: 20 September 2021].

19. Coreemu (2020a): CORE Documentation,[online] http://coreemu.github.io/core/ [accessed 25 July 2021].

20. Linux Foundation Collaborative Project (2016): *Open vSwitch,* [online] https://www.openvswitch.org/ [accessed: 10 September 2021].

21. Open Networking Foundation : Open Network Operating System project, [online] https://wiki.onosproject.org/display/ONOS/ONOS [accessed: 15 September 2021].

List of Figures