Mobile Computing

Title of Project: Text-based Chats - SDN

WS 2020/2021

Submitted by:
Riyad Ul Islam
Md Mahbub Alam

Content

Abstract

Software Defined Networking (SDN) is a new that open a new era in modern computer networking. SDN make computer networking highly programable, more efficient and added significant structure in cloud. This project focused on text base communication between end-to-end clients using SDN technology. OpenFlow protocol, Ryu controller, Floodlight controller, Network Virtualization, are basic properties of SDN has been used in this project. Maintenance and designing a network have become more interesting. SDN help to implement innovative ideas in networking field.

Key word: Ryu, Floodlight, OpenVirtex, Containernet, Docker, OpenFlow, Open vSwitch, SDN

1 Introduction

In the computer network research field Software-Defined Network is one of the latest and popular technology. SDN provides separation of control-plane and data-plane on computer network devices. Control-plane monitor how a device reacts to the flow/packet/ frame that comes to it as a part of a computer network device. On the other hand, Responsibilities of the data-plane is forwarding flow/packet/ frame.[1][2]

Among all protocols the OpenFlow protocol is the most popular between control-plane and data-plane on the SDN architecture [2]. It is an SDN key enabler. Flow-tables are used by OpenFlow to handle the mechanism for flow of packets that come to them. OpenFlow supports Multiple Flow Tables (MFT).[7][8]

Now this era the network infrastructure is changing. A single common network is shared by different applications and services instead of using different networks for different applications. For example, in 5G infrastructure network slicing is a key property as it offers operators to arrange network resources flexibly while providing various services to consumers and third-party customers.[9][10]

Among academics Ryu is a highly regarded SDN controller.[8] Ryu Controller is an open, software-defined networking (SDN) controller that increases network resilience by making traffic management and adaptation simple.[11]

2 SDN Architecture

Now this day's most of the internet network using distributed system extensively with a plenty numbers of network elements like router switch etc. Generally, all devices from a network configured and managed through various vendor-specific proprietary interfaces that may differ a lot. Managing or maintaining this type of network is very difficult. Updating the network or making any in network system change is a slow process and huge task. [3] Software defined networks (SDNs) have recently brought an effective solution of this problems. [4, 5]. Through software applications Software defined networks able to manage behavior of network dynamically. [6][3].

SDN architecture consist of 3 layers; i) Application layer, ii) Control layer & iii) Infrastructure layer.

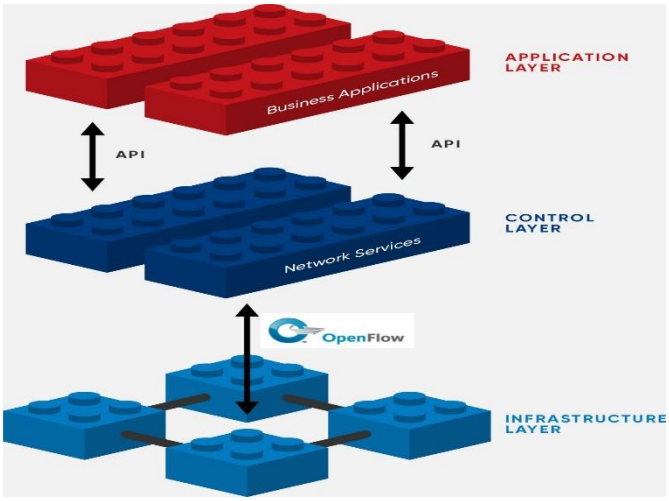


Figure 1: SDN Architecture

Infrastructure layer is the collection of different type network equipment like switches, routers, servers that pass network traffic. This layer is the physical layer that would be demonstrate under the control layer. [12]

Control layer is the logical section of the SDN which controls the network infrastructure. All network vendors work in this layer set their own products for SDN controller and framework. Control layer is also called the land of control plane. Since SDN controller layer managing networks, this layer is in the middle between all layers. Control layer can expose into two different types of interface. One is Northbound interface which defined as the connection to Application layer. Through REST APIs of SDN controllers it can generally understood. On the other hand, Southbound interface another interface. Southbound interface meant for communication with lower layer. It general realized through southbound protocols. [12]

Most wide and innovative area of SDN is Application layer. This is pace for research deploy new ideas etc. It is an open area to develop application as much as possible by support network topology, network state, network statistics, etc. Application layer uses controller layer to manage the behavior of the network also can give solution for both data center networks and real-world enterprise.[12]

3 Protocol

OpenFlow: The most popular protocol in SDN is OpenFlow. OpenFlow use Southbound interface. OpenFlow is invented by the ON.LAB, Open Networking Laboratory at Stanford University. Open Networking Foundation (ONF), a nonprofit organization manage the OpenFlow specification development. ONOS (Open Network Operating System), an SDN controller also developed by ONF specifying OpenFlow.[2]

3.1 OpenFlow Switch

Flow-tables and group-tables are used by OpenFlow Switch to perform packet matching and forwarding. Flow-table has many types of flow-entries. OpenFlow channel establish connection between An OpenFlow switch and OpenFlow controller. OpenFlow controller maintain switch using OpenFlow protocol. The controller can make

change of flow-entities int flow tables using OpenFlow flow protocol. In an OpenFlow network, OpenFlow switch holds flow tables and set of flow-entities.[8] When the controller dynamically locate devices are in the topology and must update the flow tables on those devices, Reactive Flow Entries are generated. On the other hand, a Proactive Flow Entries are programmed before traffic arrives if it's already known the communication status of two devices. [13]

When a packet comes to flow table it starts matching. The matching may continue from one flow table to another additional flow table if the network has multiple flow tables. Priority order uses to perform matching operation. Instruction is executed when any matching entity found in flow table otherwise the table-miss entry executed.[8] Packets forward to the next table using pipeline processing instructions. When the instruction set identified with a matching flow entry does not define the next table, pipeline processing comes to a halt; the packet is normally changed and transferred at this stage.[8] Logically OpenFlow switches are connected to one another via their OpenFlow ports. The network interfaces that use for transferring data between OpenFlow processing and the rest of the system are known as OpenFlow ports. [13]

4 Slicing

The new communication technology is migrating from a separate network system to a common network for different application service. A large range of new usage cases must be accommodated for the forthcoming 5G networks. Network slicing, in addition, is an integral component for 5G device implementations. It is near impossible to achieve good performance from across all use cases with a single network design meant to include all at once. Network virtualization is being researched as a way to divide physical network infrastructure resources into internally isolated, independently managed, and operated end-to-end logical networks or "slices."[3]

4.1 Network Virtualization

Network virtualizations support multiple tenants to use the same network infrastructure while each having the illusion of having their own network. This OVX mechanisms create arranged by giving access to a virtual network topology and a full network header space to each tenant. [14] There are different types of hypervisor in Network virtualizations.

FlowVisor - A hypervisor/proxy between a switch and multiple controllers that functions as an OpenFlow controller. Can perform parallel slice between multiple switches, successfully slicing a network. [15]

OpenVirteX- On top of a single physical infrastructure, OpenVirtex is a network hypervisor that can build several virtual and programmable networks. [15]

4.2 OpenVirteX

for the physical network (infrastructure) and for all tenant's virtual network OVX represent logical representations. Using topology discovery builds its view of the infrastructure in its physical network. Furthermore, OVX create the representations of virtual networks with the help of configuration data available through its API. [14]

5 Project Description

5.1 Objectives

To develop a SDN based network which can be implemented as multitenant sliced network, realised using network emulation environment Containernet **[a]**. The implemented network has to be done comprising multiple host and multiple data centre which is running on docker container and the physical network devices supporting OpenFlow protocol. Host machines must be worked as a text-based chat client and all the data centre used as server for text-based chat. Each slice must be communicated through chatting independently.

5.2 Approaches

To implement this project, there are two different approaches has been taken care of. Firstly, a ryu application has been developed to communicate with the ryu sdn controller in northbound interface which then enable the controller to communicate with Open vSwitch using OpenFlow version 3 protocol. Secondly, network virtualization with the help of OpenVirteX as a hypervisor and Floodlight as a controller has been used.

5.3.1 1st Approach

In this approach six Open vSwitch (OVS) is used to build the core network and three other switches used for providing access network to the data centre and to host machines. The given figure: **1st topology** is the topology for implementing the 1st approach.

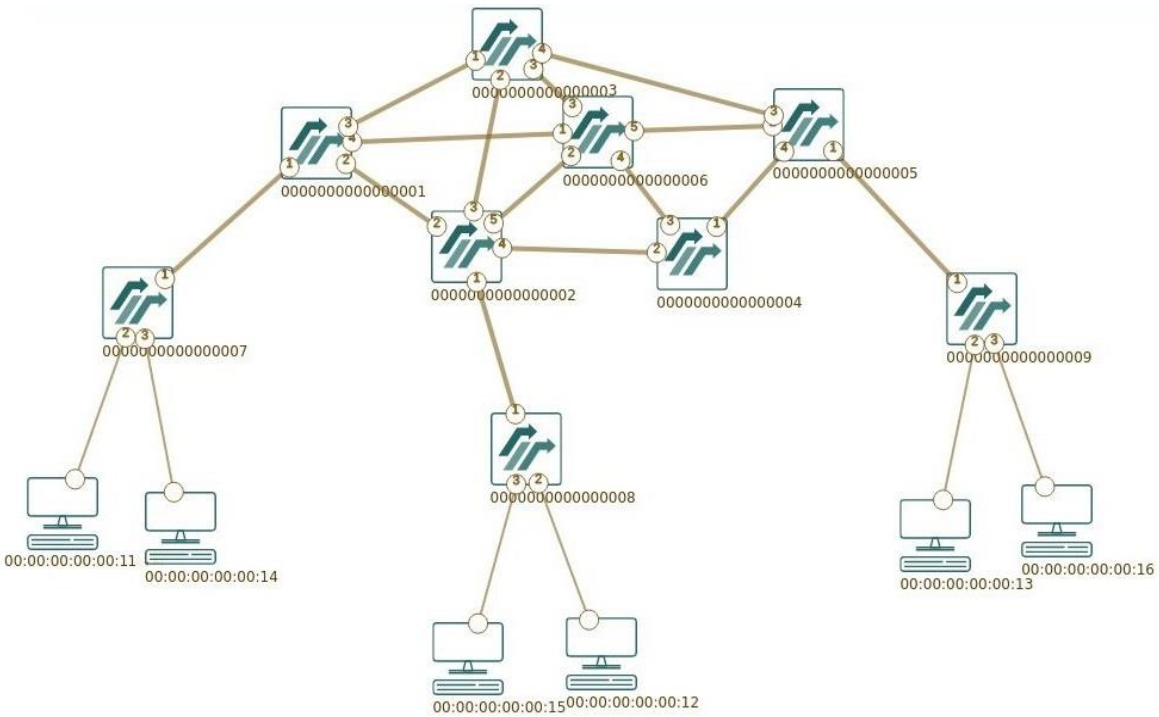


Figure: 1st topology

DPID with 7, 8 is being used to provided access service to the different host machines and DPID with 9 is for Data centre. The core switches DPID with 1 and 2 is connect with switches DPID with 7 and 8 respectively. In

the Data centre area core switch with DPID 5 is connected to the switch DPID 9. All the core switches are connected as per the figure**[a]**.

The motive of this topology is to make two slices. In slice 1 consists of h1 (Host 1 where mac ends with 11), h2 (Host 2 where mac ends with 12) and d1(Data centre service 1 where mac ends with 13) and in slice 2 consists of h3, h4, h5. Slice 1 devices or services can only communicate each other which is totally independent from slice 2 devices or services. This method is also applicable for slice 2. All the host machines are in docker container with pre-installed client chat service and data centre machines which is also in docker container pre-installed text-based chat server for providing chat service to the clients.

The transport layer for providing this chat service is provided by SDN technology which separates the data plane and the control plane. RYU **[b]** as a controller takes care of the control plane.

5.3.2 Designing Application of Controller

As it is discussed earlier that RYU is a SDN controller which provides application written on python language. For this scenario, RYU applications consider all OpenFlow version3 messages and handles properly to establish communication with OVS switch which is providing service in the data plane.

Firstly, OVS switch establish connection with RYU with TCP and sends a OpenFlow Hello message and RYU reply the Hello back to the switch. Then RYU controller gets feature of the switch the reply with Feature Reply Message from switch. Then RUY controller sends Modify-State message to switch which has been done by the event handler called CONFIG_DISPATCHER the implemented RYU application. Where controller instructs switch to add a flow with zero matching sends packet in message to controller in add_flow() function by a packet out message . For creating multiple table switches are instructed which is done by add_default_table() and add_filter_table_id (), apply_filter_table_id_rules() functios.

```
@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, 0, match, actions)
      self.logger.info("switch:%s connected", dpid)

      self.add_default_table(datapath)
      self.add_filter_table_id(datapath)
      self.apply_filter_table_id_rules(datapath)
```

To handle all the packet in message RYU application uses a event with MAIN_DISPATCHER. Where all the packet in massage is processed by the controller such as ipv6 packet, arp packet, ipv4 packet.

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
   def _packet_in_handler(self, ev):
```

In this project application the slicing is done on TCP protocol and based on all the tenants' mac address. In figure **[]** the slice is shown. Host 1, Host 2, and Data centre service 1 is in slice 100. And other tenants are in slice 200.

```
    # Slices 100 and 200 with their respective MAC addresses
slices_data = [(100,"00:00:00:00:00:11","00:00:00:00:00:12","00:00:00:00:00:13",),
          (200,"00:00:00:00:00:14","00:00:00:00:00:15","00:00:00:00:00:16",),]
```

In application its is defined that if protocol is ipv4 and TCP then add a flow to the switch table with a proper match and send packet to the defined port of the switch. The physical switch is then taken care of the multi-tenancy for these two different slices.

```
if protocol == in_proto.IPPROTO_TCP:

  for net_slice in slices_data:
        slice_id = net_slice[0]     # extract the slice ID
        net_slice = net_slice[1:]
        if src in net_slice and dst in net_slice:
        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", src, dst, slice_id,protocol)
        match = parser.OFPMatch(in_port=in_port, eth_dst=eth.dst,
eth_src=eth.src,eth_type=ether_types.ETH_TYPE_IP,
ip_proto=protocol)
        self.add_flow(datapath, 0, 1, match, actions)
        self.send_packet_out(datapath, msg.buffer_id, in_port,
        out_port, msg.data)
  else: # pair of MAC addresses are not in a slice so skip
      return
```

5.4.1 2nd Approach

This approach is based on network virtualization. In this project, OpenVirteX hypervisor is considered to fulfil the network virtualization task.

The Containernet topology implementation for this approach is slightly different from the 1st approach. In the given figure [] Host 1(h1) and Host 3 (h3) is connected to access OVS DPID ends with 1000 (which is Switch 1 Containernet topology) and Host 2(h2) and Host 4 (h4) is connected to access OVS DPID ends with 2000 (which is Switch 2 Containernet topology). Both access switch is connected to core switch with DPID 1 (which is Switch 10 Containernet topology) and switch with DPID 2 (which is Switch 11 Containernet topology) respectively. All the Data centres are connected to the core switch. Here nine core switches are considered to provide the transport layer functionality for the communication among Hosts and Data centres.

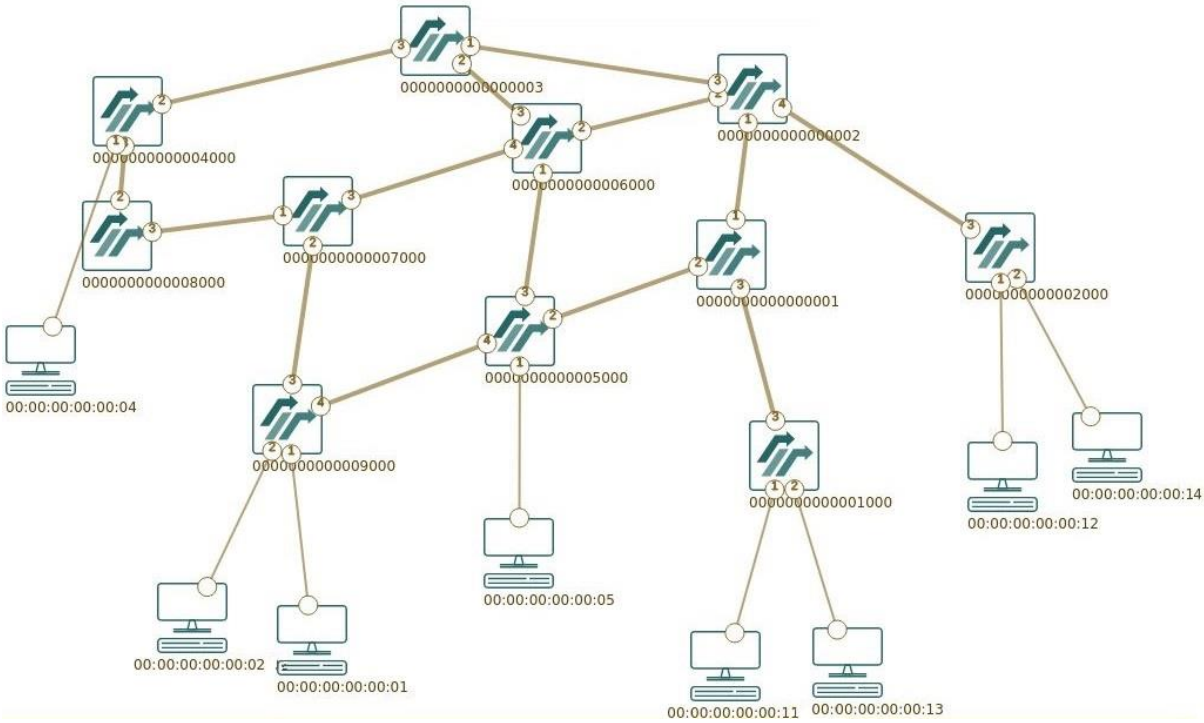


Figure: 2nd topology

5.4.2 Network Virtualization and Slicing

To implement this approach a hypervisor which is connected to all the OVS switches for controlling the data plane virtually by creating virtual network. OpenVirteX which is the used hypervisor for implementing the approach built fully independent virtual network infrastructure where tenant with a same tenant id thinks the full network to their own.

In this approach in a same slice, tenants connected to their OVS switches has considered as a single big switch. OpenVirteX constructs this big virtual switch with a tenant id by the instruction with REST API and hand it over to a controller which is connected to OpenVirteX for controlling this virtual network. In this approach Floodlight as a SDN controller has been chosen. The REST data has been given in a json format to OpenVirteX.

In the json data for the first slice comprising Host 1 (h1), Host 2 (h2), Data centre service 1 (D1) and Data centre service 5 (D5), mac address of these machines and DPID and connected port number of their respective OVS switch has been given. Moreover, tenant id, which is 1, network and routing algorithm, the specific SDN controller also bind in the json data.

For second slice, tenant id with 2 comprising Host 3 (h3), Host 3 (h3), Data centre service 4 (D4) and Data centre service 2 (D2), data with mac address of these machines and DPID and connected port number of their respective OVS switch has been given with REST API. OpenVirteX then creates two independent virtual switches controlled by two SDN controller for two different slices. With the routing algorithm information OpenVirteX creates the route. It is also mentioned there should be number backup paths if any interruptions happen in the physical path.

6 Project Implementation

6.1 1st Approach

To implement this approach first a Containernet hosted system machine (preferable is Linux Ubuntu version 18.04 LTS) is considered. Installation process of Containernet has been found in [17]. Then ryu controller has been installed from this source **[16]** in the same Linux machine.

Afterwards opening a terminal in 'project_ryu' folder, command with

```
"sudo python3 topo_with_ryu.py"
```

has been run. In another terminal in the same folder command with

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

has been run to get graphical interface of RYU controller. Then in the same terminal for starting RYU controller

```
"ryu-manager --observe-links  ~/flowmanager/flowmanager.py slicing_ryu_controller.py"
```

has been run. In the it has seen that all the switch has been connected. Afterwards in the containernet terminal command with

```
"xterm h1"
"xterm d1"
"xterm h5"
"xterm d2"
"xterm h2"
"xterm h4"
```

has been run sequentially. So, windows have popped up for host and data centre. In d1 and d2 terminal below given command has been run respectively.

```
"python3 server.py 10.0.1.3"
"python3 server.py 10.0.1.6"
```

It is seen that in d1 and d2 chat server is listening. In h1 and h2 terminal below given command has been run respectively.

```
"python3 chat_client.py 10.0.1.3 -p 1060"
```

In h4 and h5 terminal below given command has been run respectively.

```
"python3 chat_client.py 10.0.1.6 -p 1060"
```

In every host client's terminal individual name has been pressed to enter the chat room. Because there is no display installed in host container, client has got error for opening tkinter. So any key has been pressed and found the chat room in cli mode. After wards it has seen that host 1 and host 2 can chat and host 4 and host 5 only can chat in their own individual chat room.

Afterwards to check the flow table of the OVS switch, in a browser flow manager has been opened with below address.

```
http://localhost:8080/home/index.html
```

6.2 2nd Approach

For running this approach, in the same Linux machine of 1st approach below given command has been run after opening a terminal in 'openvirtex_project' folder.

```
"sudo python3 topo_openvirtex.py"
```

Before running the above command, it is necessary to be reassured that in topo_openvirtex.py file the remote ip address of the controller should be OpenVirteX hosted machine.

A machine with pre-installed OpenVirteX and Floodlight Linux machine has been found in [18].

In the pre-installed OpenVirteX machine, by opening a terminal in 'OpenVirtex/scripts/' folder given command is run.

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

It has seen that OpenVirtex is connected with containernet topology from the log. Then by opening a terminal in 'OpenVirtex/utils/' folder given command is run

```
"python embedder.py"
```

This is for the REST communication with OpenVirteX. Afterwards file named slice1.json and slice2.json in 'openvirtex_project' folder of Containernet hosted machine has been copied to

in '/home/ovx' folder of OpenVirteX hosted machine. In '/home/ovx' folder of OpenVirteX hosted machine by opening terminal below commands has been run sequentially.

```
curl localhost:8000 -X POST -d @slice1.json
curl localhost:8000 -X POST -d @slice2.json
```

Afterwards in the Containernet terminal command with

```
“xterm h1”
“xterm d1”
“xterm h3”
“xterm d2”
“xterm h2”
“xterm h4”
```

has been run sequentially. So, windows have popped up for host and data centre. In d1 and d2 terminal below given command has been run respectively.

```
“python3 server.py 10.1.0.7”
“python3 server.py 10.2.0.7”
```

It is seen that in d1 and d2 chat server is listening. In h1 and h2 terminal below given command has been run respectively.

```
“python3 chat_client.py 10.1.0.7 -p 1060”
```

In h3 and h4 terminal below given command has been run respectively.

```
“python3 chat_client.py 10.2.0.7 -p 1060”
```

In every host client’s terminal individual name has been pressed to enter the chat room. Because there is no display installed in host container, client has got error for opening tkinter. So, any key has been pressed and found the chat room in cli mode. After wards it has seen that host 1 and host 2 can chat and host 3 and host 4 only can chat in their own individual chat room. For checking the virtual switch and ip address for clients opened in below given two address in browser for slice 1 and slice 2 respectively.

http://localhost:10001/ui/index.html

http://localhost:10001/ui/index.html

7 Project Outcome Analysis

7.1 1st Approach

After successful chatting hosts of both individual slices the below figure is the flow table for the switch 7.

Flow Table 0

PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS
65535	eth_type = 35020 eth_dst = 01:80:c2:00:00:0e	0	779	0	0	OUTPUT:CONTROLLER	435	26100	0
32768	ANY	0	779	0	0	GOTO_TABLE:1	144	11596	0

Flow Table 1

PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS
10	eth_type = 2048 ip_proto = 17	0	779	0	0	DROP	0	0	0
1	ANY	0	779	0	0	GOTO_TABLE:2	144	11596	0
1	eth_type = 2054 eth_dst = 00:00:00:00:00:12 in_port = 3	0	775	0	0	OUTPUT:1	1	42	0
1	eth_type = 2054 eth_dst = 00:00:00:00:00:13 in_port = 3	0	771	0	0	OUTPUT:1	1	42	0
1	eth_type = 2054 eth_dst = 00:00:00:00:00:12 in_port = 2	0	770	0	0	OUTPUT:1	0	0	0
1	eth_type = 2054 eth_dst = 00:00:00:00:00:14 in_port = 1	0	769	0	0	OUTPUT:3	7	294	0
1	eth_type = 2054 eth_dst = 00:00:00:00:00:13 in_port = 2	0	767	0	0	OUTPUT:1	2	84	0
1	eth_type = 2054 eth_dst = 00:00:00:00:00:14 in_port = 2	0	764	0	0	OUTPUT:3	0	0	0
1	eth_type = 2054 eth_dst = 00:00:00:00:00:15 in_port = 3	0	759	0	0	OUTPUT:1	0	0	0
1	eth_type = 34525	0	779	0	0	DROP	36	4880	0

☐	1	eth_type = 2048 eth_dst = 00:00:00:00:00:11 ip_proto = 1 eth_src = 00:00:00:00:00:15 in_port = 1	0	776	0	0	OUTPUT:2	1	98	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 = 1	0	776	0	0	OUTPUT:2	1	98	0
☐	1	eth_type = 2048 eth_dst = 00:00:00:00:00:14 ip_proto = 1 eth_src = 00:00:00:00:00:12 in_port = 1	0	775	0	0	OUTPUT:3	1	98	0
☐	1	eth_type = 2048 eth_dst = 00:00:00:00:00:12 ip_proto = 1 eth_src = 00:00:00:00:00:14 in_port = 3	0	774	0	0	OUTPUT:1	1	98	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 = 1	0	771	0	0	OUTPUT:3	1	98	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 = 3	0	771	0	0	OUTPUT:1	1	98	0
☐	1	eth_type = 2048 eth_dst = 00:00:00:00:00:14 ip_proto = 6 eth_src = 00:00:00:00:00:16 in_port = 1	0	137	0	0	OUTPUT:3	2	132	0
☐	1	eth_type = 2048 eth_dst = 00:00:00:00:00:13 ip_proto = 6 eth_src = 00:00:00:00:00:11 in_port = 2	0	50	0	0	OUTPUT:1	3	243	0
☐	1	eth_type = 2048 eth_dst = 00:00:00:00:00:11 ip_proto = 6 eth_src = 00:00:00:00:00:13 in_port = 1	0	50	0	0	OUTPUT:2	2	132	0
☐	0	ANY	0	779	0	0	OUTPUT:CONTROLLER	73	4697	0

It has been seen that in table 2 for TCP protocol communication the both slice is to instruct slice 1 to go to output port 2 and for slice 2 to go to output port 3 to maintain the multitenancy. In switch 9 it is also instructed instruct slice 1 to go to output port 2 and for slice 2 to go to output port 3 to maintain the multitenancy from the below illustrated flow table of switch 9.

Flow Table 0

PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS
65535	eth_type = 35020 eth_dst = 01:80:c2:00:00:0e	0	779	0	0	OUTPUT:CONTROLLER	435	26100	0
32768	ANY	0	779	0	0	GOTO_TABLE:1	131	10754	0

Flow Table 1

PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS
10	eth_type = 2048 ip_proto = 17	0	779	0	0	DROP	0	0	0
1	ANY	0	779	0	0	GOTO_TABLE:2	131	10754	0

Flow Table 2

PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS
1	eth_type = 34525	0	779	0	0	DROP	36	4880	0
1	eth_type = 2054 eth_dst = 00:00:00:00:00:11 in_port = 2	0	778	0	0	OUTPUT:1	3	126	0
1	eth_type = 2054 eth_dst = 00:00:00:00:00:11 in_port = 3	0	776	0	0	OUTPUT:1	2	84	0
1	eth_type = 2054 eth_dst = 00:00:00:00:00:12 in_port = 2	0	775	0	0	OUTPUT:1	1	42	0
1	eth_type = 2054 eth_dst = 00:00:00:00:00:12 in_port = 3	0	773	0	0	OUTPUT:1	1	42	0
1	eth_type = 2054 eth_dst = 00:00:00:00:00:13 in_port = 1	0	771	0	0	OUTPUT:2	7	294	0
1	eth_type = 2054 eth_dst = 00:00:00:00:00:16 in_port = 1	0	770	0	0	OUTPUT:3	6	252	0
1	eth_type = 2054 eth_dst = 00:00:00:00:00:13 in_port = 3	0	770	0	0	OUTPUT:2	1	42	0
1	eth_type = 2054 eth_dst = 00:00:00:00:00:14 in_port = 3	0	767	0	0	OUTPUT:1	3	126	0
1	eth_type = 2054 eth_dst = 00:00:00:00:00:14 in_port = 2	0	766	0	0	OUTPUT:1	0	0	0

1	eth_type = 2054 eth_dst = 00:00:00:00:00:15 in_port = 3	0	763	0	0	OUTPUT:1	1	42	0
1	eth_type = 2054 eth_dst = 00:00:00:00:00:15 in_port = 2	0	760	0	0	OUTPUT:1	0	0	0
1	eth_type = 2054 eth_dst = 00:00:00:00:00:16 in_port = 2	0	757	0	0	OUTPUT:3	0	0	0
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	776	0	0	OUTPUT:3	1	98	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	776	0	0	OUTPUT:1	1	98	0
1	eth_type = 2048 eth_dst = 00:00:00:00:00:13 ip_proto = 1 eth_src = 00:00:00:00:00:12 in_port = 1	0	775	0	0	OUTPUT:2	1	98	0
1	eth_type = 2048 eth_dst = 00:00:00:00:00:12 ip_proto = 1 eth_src = 00:00:00:00:00:13 in_port = 2	0	775	0	0	OUTPUT:1	1	98	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	771	0	0	OUTPUT:1	1	98	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	771	0	0	OUTPUT:2	1	98	0
1	eth_type = 2048 eth_dst = 00:00:00:00:00:16 ip_proto = 1 eth_src = 00:00:00:00:00:15 in_port = 1	0	763	0	0	OUTPUT:3	1	98	0
1	eth_type = 2048 eth_dst = 00:00:00:00:00:15 ip_proto = 1 eth_src = 00:00:00:00:00:16 in_port = 3	0	763	0	0	OUTPUT:1	1	98	0
1	eth_type = 2048 eth_dst = 00:00:00:00:00:16 ip_proto = 6 eth_src = 00:00:00:00:00:14 in_port = 1	0	137	0	0	OUTPUT:3	3	249	0
1	eth_type = 2048 eth_dst = 00:00:00:00:00:14 ip_proto = 6 eth_src = 00:00:00:00:00:16 in_port = 3	0	137	0	0	OUTPUT:1	2	132	0
1	eth_type = 2048 eth_dst = 00:00:00:00:00:13 ip_proto = 6 eth_src = 00:00:00:00:00:11 in_port = 1	0	50	0	0	OUTPUT:2	3	243	0
1	eth_type = 2048 eth_dst = 00:00:00:00:00:11 ip_proto = 6 eth_src = 00:00:00:00:00:13 in_port = 2	0	50	0	0	OUTPUT:1	2	132	0
0	ANY	0	779	0	0	OUTPUT:CONTROLLER	52	3284	0

Captured hello packet is given below while controller communicate with the OVS.

```
▸ Frame 142: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 6653, Dst Port: 54524, Seq: 1, Ack: 1, Len: 8
▾ OpenFlow 1.3
    Version: 1.3 (0x04)
    Type: OFPT_HELLO (0)
    Length: 8
    Transaction ID: 4090629382
```

Captured Feature Request packet is given below while controller communicate with the OVS.

```
▸ Frame 146: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 6653, Dst Port: 54524, Seq: 9, Ack: 9, Len: 8
▾ OpenFlow 1.3
    Version: 1.3 (0x04)
    Type: OFPT_FEATURES_REQUEST (5)
    Length: 8
    Transaction ID: 4090629383
```

Captured packet in for TCP protocol packet is given below while controller communicate with the OVS.

```
▸ Frame 14267: 180 bytes on wire (1440 bits), 180 bytes captured (1440 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 6653, Dst Port: 54536, Seq: 34585, Ack: 16987, Len: 114
▾ OpenFlow 1.3
    Version: 1.3 (0x04)
    Type: OFPT_PACKET_OUT (13)
    Length: 114
    Transaction ID: 4087117120
    Buffer ID: OFP_NO_BUFFER (4294967295)
    In port: 2
    Actions length: 16
    Pad: 000000000000
  ▸ Action
  ▾ Data
    ▸ Ethernet II, Src: 00:00:00_00:00:11 (00:00:00:00:00:11), Dst: 00:00:00_00:00:13 (00:00:00:00:00:13)
    ▸ Internet Protocol Version 4, Src: 10.0.1.1, Dst: 10.0.1.3
    ▸ Transmission Control Protocol, Src Port: 37594, Dst Port: 1060, Seq: 0, Len: 0
```

Captured flow mod packet is given below while controller communicate with the OVS.

Captured packet out packet is given below while controller communicate with the OVS.

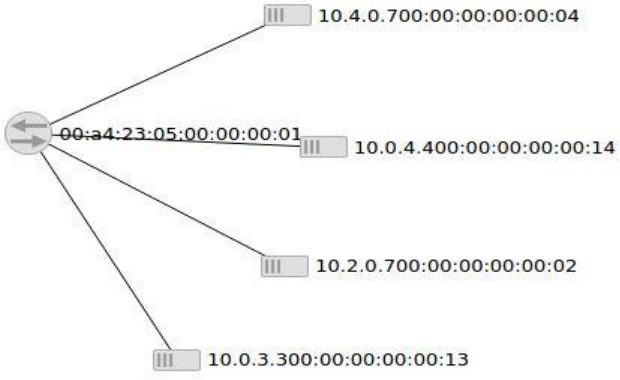
```
▾ OpenFlow 1.3
    Version: 1.3 (0x04)
    Type: OFPT_FLOW_MOD (14)
    Length: 120
    Transaction ID: 4090629819
    Cookie: 0x0000000000000000
    Cookie mask: 0x0000000000000000
    Table ID: 2
    Command: OFPFC_ADD (0)
    Idle timeout: 0
    Hard timeout: 0
    Priority: 1
    Buffer ID: OFP_NO_BUFFER (4294967295)
    Out port: 0
    Out group: 0
  ▸ Flags: 0x0000
    Pad: 0000
  ▸ Match
  ▾ Instruction
      Type: OFPIT_APPLY_ACTIONS (4)
      Length: 24
      Pad: 00000000
    ▾ Action
        Type: OFPAT_OUTPUT (0)
        Length: 16
        Port: 4
        Max length: 65509
        Pad: 000000000000
```

```
▸ Frame 14273: 180 bytes on wire (1440 bits), 180 bytes captured (1440 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 6653, Dst Port: 54524, Seq: 43583, Ack: 49233, Len: 114
▾ OpenFlow 1.3
    Version: 1.3 (0x04)
    Type: OFPT_PACKET_OUT (13)
    Length: 114
    Transaction ID: 4090629820
    Buffer ID: OFP_NO_BUFFER (4294967295)
    In port: 1
    Actions length: 16
    Pad: 000000000000
  ▾ Action
      Type: OFPAT_OUTPUT (0)
      Length: 16
      Port: 4
      Max length: 65509
      Pad: 000000000000
  ▾ Data
    ▸ Ethernet II, Src: 00:00:00_00:00:11 (00:00:00:00:00:11), Dst: 00:00:00_00:00:13 (00:00:00:00:00:13)
    ▸ Internet Protocol Version 4, Src: 10.0.1.1, Dst: 10.0.1.3
    ▸ Transmission Control Protocol, Src Port: 37594, Dst Port: 1060, Seq: 0, Len: 0
```

7.2 2nd Approach

The virtual switch for slice 1 is given below where h1, h2, d1 and d5 communicates in a single virtual switch even they are separated by distance.

The virtual switch for slice 2 is given below where h3, h4, d2 and d4 communicates in a single virtual switch even they are separated by distance.



The virtual IP address which has been created for slice 1 in is show below figure

MAC Address	IP Address	Switch Port
00:00:00:00:00:11	10.0.1.1	00:a4:23:05:00:00:00:01-1
00:00:00:00:00:01	10.1.0.7	00:a4:23:05:00:00:00:01-4
00:00:00:00:00:12	10.0.2.2	00:a4:23:05:00:00:00:01-2
00:00:00:00:00:05	10.5.0.7	00:a4:23:05:00:00:00:01-3

The virtual IP address which has been created for slice 2 in is show below figure

MAC Address	IP Address	Switch Port
00:00:00:00:00:13	10.0.3.3	00:a4:23:05:00:00:00:01-1
00:00:00:00:00:02	10.2.0.7	00:a4:23:05:00:00:00:01-4
00:00:00:00:00:14	10.0.4.4	00:a4:23:05:00:00:00:01-2
00:00:00:00:00:04	10.4.0.7	00:a4:23:05:00:00:00:01-3

The flow table of without any route for switch 10 has been shown in below figure

Flow Table 0

PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS
0	eth_type = 35020 eth_dst = 01:80:c2:00:00:0e vlan_vid = 0x0000 eth_src = ee:85:07:bb:fd:d0 in_port = 2	0	0	0	1	DROP	1	60	4
0	eth_type = 35020 eth_dst = 01:80:c2:00:00:0e vlan_vid = 0x0000 eth_src = ba:50:5b:41:ee:fe in_port = 3	0	0	0	1	DROP	1	60	4
0	eth_type = 35020 eth_dst = 01:80:c2:00:00:0e vlan_vid = 0x0000 eth_src = 1e:90:5f:14:f9:88 in_port = 1	0	0	0	1	DROP	1	60	4

When h4 communicates with d2 only below flow table has been found for route switch 2 is given below:

Flow Table 0

PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS
0	eth_dst = 00:00:00:00:00:02 vlan_vid = 0x0000 eth_src = 00:00:00:00:00:14 in_port = 2	8589934598	90	5	0	SET_FIELD: ipv4_dst:2.0.0.6 SET_FIELD: ipv4_src:2.0.0.4 OUTPUT:3	93	8834	5
0	eth_dst = 00:00:00:00:00:14 vlan_vid = 0x0000 eth_src = 00:00:00:00:00:02 in_port = 3	8589934599	90	5	0	SET_FIELD: ipv4_src:10.2.0.7 SET_FIELD: ipv4_dst:10.0.4.4 OUTPUT:2	93	8834	4
0	eth_type = 35020 eth_dst = 01:80:c2:00:00:0e vlan_vid = 0x0000 eth_src = 06:63:b0:2a:48:0d in_port = 3	0	0	0	1	DROP	1	60	4

When h4 communicates with d2 only below flow table has been found for route switch 9 is given below:

Flow Table 0

PRIORITY	MATCH FIELDS	COOKIE	DURATION	IDLE TIMEOUT	HARD TIMEOUT	INSTRUCTIONS	PACKET COUNT	BYTE COUNT	FLAGS
0	eth_dst = 00:00:00:00:00:02 vlan_vid = 0x0000 eth_src = 00:00:00:00:00:14 in_port = 4	8589934598	145	5	0	SET_FIELD: ipv4_src:10.0.4.4 SET_FIELD: ipv4_dst:10.2.0.7 OUTPUT:2	150	14196	4
0	eth_dst = 00:00:00:00:00:14 vlan_vid = 0x0000 eth_src = 00:00:00:00:00:02 in_port = 2	8589934599	145	5	0	SET_FIELD: ipv4_dst:2.0.0.4 SET_FIELD: ipv4_src:2.0.0.6 OUTPUT:4	150	14196	5
0	eth_type = 35020 eth_dst = 01:80:c2:00:00:0e vlan_vid = 0x0000 eth_src = d2:37:ca:3f:17:79 in_port = 4	0	0	0	1	DROP	1	60	4

After reviewing all the flow tables it is seen that OpenVirteX creates virtual for separating the whole data plane for individual tenants. All other switches are instructing to drop all packet rather than the tenants in a same virtual switch.

8 Future Work

After learning from this project to extends this project two future work can be considered BGP Routing In control layer and data plane it should be a possible work with BGP protocol with the help of RYU controller. Hypervisor Developing hypervisor for virtualization the network can extends the project.

Conclusion

This project has implemented using a small portion on SDN technology. The field of SDN has borderless flexibility. SND centralize the network and provide more easer management. A network can control individually by slicing with SDN. SDN has taken the network technology one step ahead. There is also lots of scope to research and deploy new thinking.

1. F. Akyildiz, A. Lee, P. Wang, M. Luo and W. Chou, "Research challenges for traffic engineering in software defined networks," in IEEE Network, vol. 30, no. 3, pp. 52-58, May-June 2016, doi: 10.1109/MNET.2016.7474344.
2. F. Baskoro, R. Hidayat and S. B. Wibowo, "Comparing LACP Implementation between Ryu and Opendaylight SDN Controller," 2019 11th International Conference on Information Technology and Electrical Engineering (ICITEE), Pattaya, Thailand, 2019, pp. 1-4, doi: 10.1109/ICITEED.2019.8929986.
3. Thomas Langenskiöld, "Network Slicing using Switch Virtualization", Master's thesis, School of Electrical Engineering, 19th November 2017, Available at: https://aaltodoc.aalto.fi/bitstream/handle/123456789/29159/master_Langenski%C3%B6ld_Thomas_2017.pdf?isAllowed=y&sequence=1
4. Diego Kreutz et al. "Software-Defined Networking: A Comprehensive Survey". In: Proceedings of the IEEE 103.1 (2015), pp. 14–76. issn: 0018-9219. doi: 10.1109/JPROC.2014.2371999.
5. Nick Feamster, Jennifer Rexford, and Ellen Zegura. "The Road to SDN: An Intellectual History of Programmable Networks". In: *ACM Sigcomm Computer Communication* 44.2 (2014), pp. 87–98. issn: 01464833. doi: 10.1145/2602204.2602219. url: http://dl.acm.org/citation.cfm? id = 2602204 . 2602219{\ & }coll = DL{\ & }dl = ACM{\ & }CFID = 429855848{\ & }CFTOKEN=24281772.
6. S. Denazis et al. *RFC 7426 - Software-Defined Networking (SDN): Layers and Architecture Terminology*. Tech. rep. 2015. doi: 10.17487/rfc7426. url: https://www.rfc-editor.org/info/rfc7426
7. M. J. Hayes, "Scalability and Performance Considerations for Traffic Classification in Software-Defined Networks," 2016, Available at: https://researcharchive.vuw.ac.nz/xmlui/bitstream/handle/10063/5660/thesis.pdf?sequence=1
8. F. Baskoro, R. Hidayat and S. B. Wibowo, "LACP Experiment using Multiple Flow Table in Ryu SDN Controller," 2019 2nd International Conference on Applied Information Technology and Innovation (ICAITI), Denpasar, Indonesia, 2019, pp. 51-55, doi: 10.1109/ICAITI48442.2019.8982149.
9. R. Ravindran, A. Chakraborti, S. O. Amin, A. Azgin, and G. Wang, "5G-ICN: Delivering ICN services over 5G using network slicing," *IEEE Communications Magazine*, vol. 55, no. 5, pp. 101-107, May 2017.
10. D. Scano, L. Valcarenghi, K. Kondepu, P. Castoldi and A. Giorgetti, "Network Slicing in SDN Networks," 2020 22nd International Conference on Transparent Optical Networks (ICTON), Bari, Italy, 2020, pp. 1-4, doi: 10.1109/ICTON51198.2020.9203184.
11. https://www.sdxcentral.com/networking/sdn/definitions/what-is-ryu-controller/
12. https://www.howtoforge.com/tutorial/software-defined-networking-sdn-architecture-and-role-of-openflow/
13. https://overlaid.net/2017/02/15/openflow-basic-concepts-and-theory/
14. https://openvirtex.com/documentation/architecture/overview/#1.1
15. https://github.com/sdnds-tw/awesome-sdn#network-virtualization
16. https://ryu.readthedocs.io/en/latest/getting_started.html
17. https://github.com/containernet/containernet
18. https://openvirtex.com/getting-started/installation/