**Project Description**

**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 has to be worked as a text-based chat client and all the data centre used as server for text-based chat. Each slice has to be communicated through chatting independently.

**Approaches**

To implement this project, there are two different approaches has been taken care of. Firstly, an 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.

**1st Approach**

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

Figure\*

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 communicates 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.

**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 considers 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 this two different slice.

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

**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 slight 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 \*

**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. OpenVirtext then creates two independent virtual switches controlled by two SDN controller for two different slice. With the routing algorithm information OpenVirtex creates the route. It is also mentioned there should be number backup paths if any interruptions happens in the physical path.

**Project Implementation**

**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 []. Then ryu controller has been installed from this source [] 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>

**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 [].

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>

**Project Outcome Analysis**

**1st Approach**

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

Figure \*

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.

Figure \*

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

![Text

Description automatically generated]()

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

![Text

Description automatically generated]()

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

![Graphical user interface, text, application, email

Description automatically generated]()

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

![A close-up of a document

Description automatically generated with low confidence]()

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

![Graphical user interface, text, application, email

Description automatically generated]()

**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.

![Diagram

Description automatically generated]()

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.

![Diagram

Description automatically generated]()

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

![Table

Description automatically generated]()

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

![Table

Description automatically generated]()

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

![Table

Description automatically generated]()

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

![Table

Description automatically generated]()

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

![Table

Description automatically generated]()

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 instructs to drop all packet rather than the tenants in a same virtual switch.

**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.