Network Slicing in an Emulated Network Environment using NSH

Mobile Computing Winter Semester 20/21

Submitted by:	Maria Majid 1324374	Sidra Hussain 1318131
Examiner:	Prof. Ulrich Trick	
Date of submission:	8th March, 2021	
Program:	M.E in Information Technology	

Abstract

Network slicing plays an important role for enabling the 5G technology. It primarily provides means to use network as a service for the use cases. This technology permits network operators to build multiple virtual networks on a physical single network infrastructure. This helps service providers to deploy new services flexibly and accommodate diversified services easily which in turn partitions different services for the provision of optimum support. In this project, network slicing is achieved using Network service header encapsulation to implement Chat room service across multiple hosts. For Software Defined Network approach, OpenFlow protocol is utilized in the extensively designed emulated network environment.

Table of Contents

Table of Figures

Table of Contribution

Maria Majid 1324374	Sidra Hussain 1318131
	Section 1
	Section 2
	Section 3
	Section 4.1
Section 4.2-4.3	
	Section 5
	Section 6.1-6.4
Section 6.5-6.6	
	Section 7
Section 8	
	Section 9
	Section 10.1-10.2
Section 10.3.1-10.3.2	
	Section 10.3.3-10.3.4
Section 10.3.5-10.3.6	
Section 10.4	
Section 11.1	
	Section 11.2
Section 11.3	
Section 11.4	
Section 12.1	Section 12.1
Section 12.2	
	Section 12.3
Section 12.4	
Section 12.5	
Section 12.6	

1. Introduction of Network Slicing

On shared physical networks, a network slice is an end-to-end logical network with a collection of isolated virtual resources. These logical networks are offered as a variety of services to meet the various connectivity needs of users. Network slicing is a network-as-a-service (NaaS) model that allows users to configure network slices for various and complex 5G connectivity scenarios. Slicing networks would be a key aspect of 5G networks [1].

In comparison to conventional networks, slice-based 5G offers the following major advantages [1]:

- As compared to physical networks, network slicing can provide logical networks with better efficiency.
- As service requirements and user numbers shift, a network slice will scale up or down.
- Network slices can separate one service's network resources from those of other services. The configurations of the different slices have no effect on one another. As a result, each slice's reliability and security can be improved.
- Finally, a network slice is tailored to the needs of the service, allowing for the most efficient allocation and utilization of physical network resources. Network operators may assign the appropriate number of resources per network slice. As a result, it aids in the productive and efficient use of resources.
- It aids network operators in lowering operating expenses (OPEX) and capital expenses (CAPEX).

2. Software Defined Network SDN

With the increase in the network traffic with emerging data formats, service types and devices, there is a need to devise a solution to cope with the highly demanding user requirements [2].

For such problems Software Defined Networking (SDN) is introduced to cope with these challenges and virtualize the network functions. SDN enables networks to respond dynamically to changes in user patterns and network resource availability. Network architectures can be updated in real time to respond to application and user demands, and services can be introduced much more efficiently, conveniently, and affordably [2].

SDN uses a protocol that modifies forwarding tables in network switches to separate the control plane (controller) and data plane (switch) functions of networks. This allows networks to be optimized as needed and adapt quickly to changes in network usage without having to manually reconfigure existing infrastructure or buy new hardware [2].

SDN decouples the control of network devices from the data they transport, while separating the switching software from the network hardware. It also includes a controller that has a detailed view of the entire network and its state, as well as the ability to communicate in real time with switches (network resources) and applications (network consumers). The controller enables networks to communicate with applications and efficiently reconfigure themselves as required, enabling multiple logical network topologies to be implemented on a single common network fabric [2].

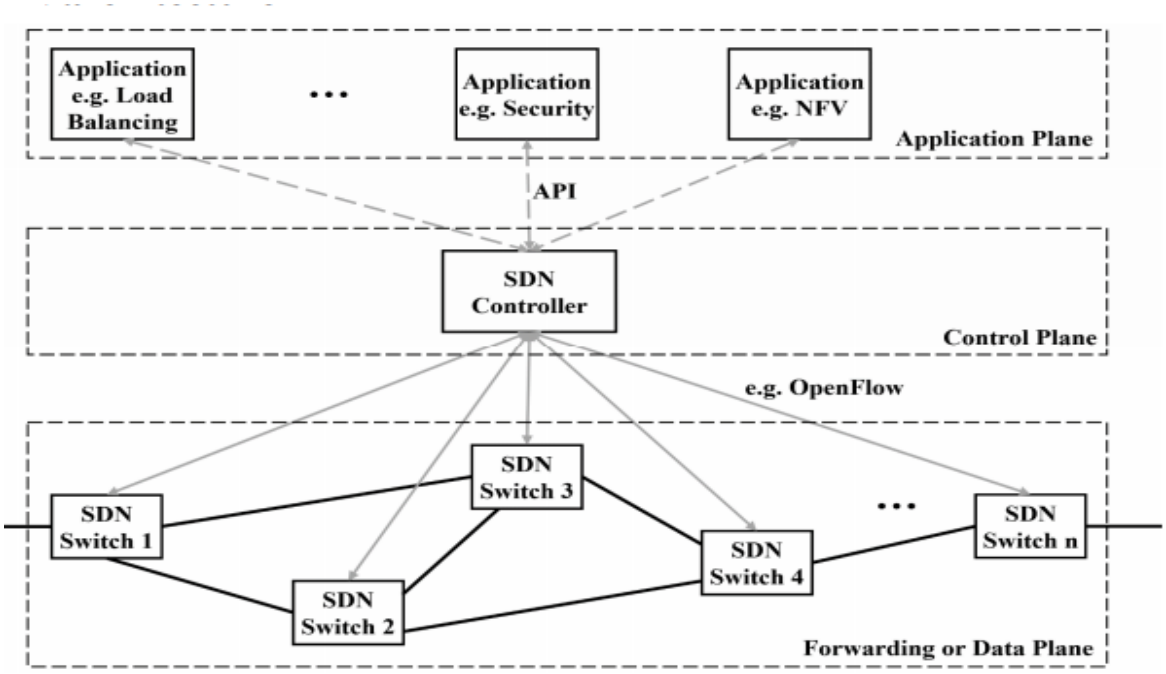

Figure 1 Software Defined Network Architecture

3. Use cases

The use cases of this project are seven hosts distributed over three access networks which will communicate with each other via a chat room service.

4. Network Service Header Protocol Description

As the networks, servers, storage and applications are changing, it is impacting the speed of operation and operational costs. To counter these problems a concept of Service Chaining or Service Function Chaining is introduced [2]. It comprises of series of service functions that a packet must traverse through. Although the service function chaining offers a solution for the expanding services, it cannot be easily implemented with the new services being physically implemented and the traffic moving through VLANs. The missing compensation introduces limitations and makes the system inefficient as the network providers on a new service have to deploy new hardware resources and reconfigure the network. These service functions are topology independent and not adaptable to virtualization [4].

These limitations made way to a new approach called the Network Service Header. It uses the transport encapsulation technique to traverse the packet through the required service function. NSH first encapsulates the original frame/packet, then this NSH packet is encapsulated on any transport header to traverse through the defined path [3].

NSH creates a dedicated service plane and is independent of the underlying transport protocol over which it is applied. Mainly it describes where the packet will be routed before reaching the destination address, this information is appended via service path information. It is deployable by many transport protocols as it is agnostic [3].

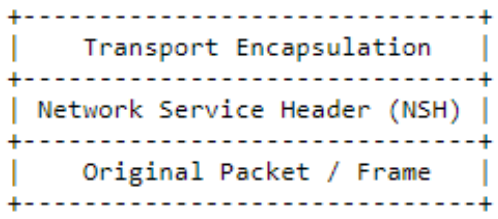

Figure 2 Network Service Header Encapsulation [2]

4.1 NSH Header

The NSH header comprises of a Base Header, Service Path header and a Context header, which are each 4 bytes long. Except context header which can vary according to the type of the context header [3].

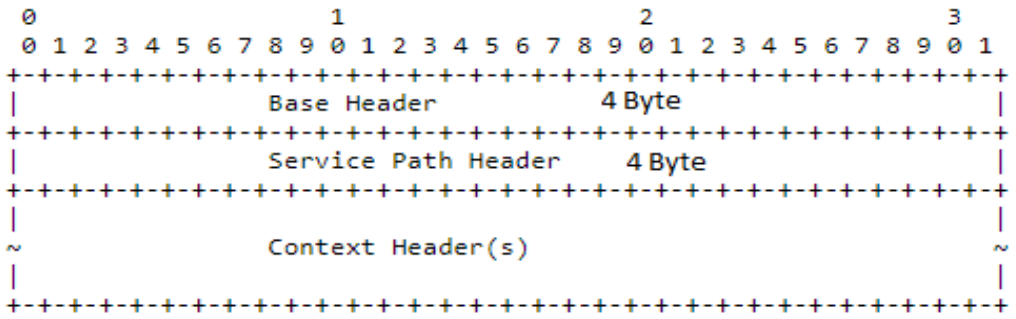

Figure 3 NSH Header format

4.1.1 NSH Base Header

The base header describes the service header and provides information about the payload protocol [3].

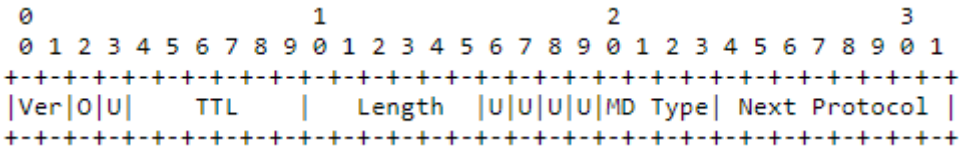

Figure 4 NSH Base Header [2].

It consists of the unique MD Type and Next protocol field. These are responsible of indicating the format of the metadata and the payload type being appended [3].

> **MD type:** In the current project MD Type 1 is used which represents a Fixed Length context header (details in section 4.1.3), using the value 0x1 for the field [3].

Next Protocol: It represents the protocol type of the encapsulated data. The current project works with payload of Ethernet, using the value 0x3 for the field [3].

4.1.2 NSH Service Path Header

The Service Path Header describes the service path helps identifying the location in the path [3].

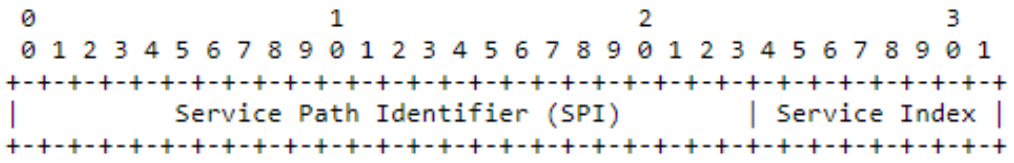

Figure 5 NSH Service Path Header

Following is the purpose of these fields in the header:

Service Path Identifier (24 bits): It is used to identify the Service Path Function for the participating nodes. In this project the specific slices have unique SPI that helps the nodes to identify the packet is valid to be processed [3].

Service Index (8 bits): The index describes the location of the packet in the service path. Every node decrements this index by one when processed [3].

4.1.3 NSH Context Header

The context header used in the scope of this project is of MD type 1. It carries 4x4 byte mandatory context headers. This particular type is used when the header carries opaque metadata between service functions. This is to offer flexibility to different use cases in the network [4].

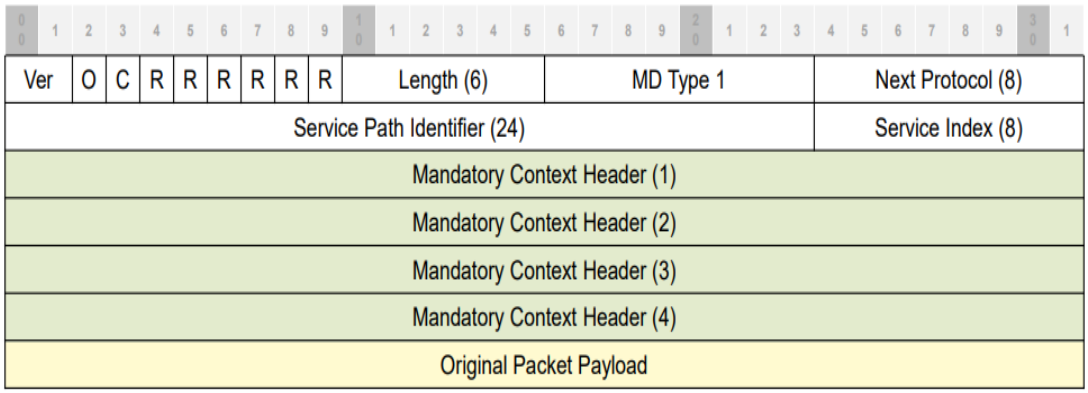

Figure 6 NSH Context Header [4]

4.2 NSH actions

The NSH encapsulation nodes called the NSH aware nodes can perform the following actions [3]:

1. Insert or remove the NSH:
   At the start and end of the service path this action needs to take place. The packets after classification if needed to require service function chaining are encapsulated. The node susceptible on this action is the Service classifier which imposes the NSH and a Service Function Forwarder which un-encapsulates the packet before delivering.

2. Select service path:
   This is used by Service Function or Service Function Forwarder to ensure the service path selection is correct.
3. Update the NSH:
   Service function decreases the service index by one after the service or else the packet is dropped by the Service forwarder if SPI and SI does not match.

   There may be an update via classifiers if the context header changes.

   SFC proxy also updates the service index when it is used in front of an NSH unaware node. As it updates the SI and un-encapsulate the packet before giving it and encapsulates the packet after receiving the packet updating the SI in both conditions.

4. Service policy selection: Service functions may permit or deny the actions according to the metadata shared in NSH.

4.3 NSH Encapsulation

NSH encapsulation serves the purpose of creating topologically independent plane. That is, packets are forward without modifying the underlying topology. This encapsulation also helps in directly forwarding the packets through transit nodes without any modification as after the NSH this transport encapsulation is outward (which is independent of the service header) before starting the service chain. The NSH is indicated via a protocol in this transport encapsulation [3].

5. Project Description

The main purpose of the project is to implement network slicing on an extensive network. This makes the basis of 5G network. As discussed in detail before, the NSH approach in OVS switches has given the opportunity to establish slices and make an environment close to a typical SDN. To show the functioning of the slices, a chat room service is deployed on the sliced network. Many approaches have been tested regarding the functioning of NSH over an emulated network. The preceding document is a detailed guide to the technologies used, approaches tried and the trials of all possible techniques in obtaining the desired results.

6. Tools and Technologies used to setup Network Emulation Environment

The implement the project the extensive network required was emulated using the following technologies:

6.1 Linux based VM

The software used was Oracle VM VirtualBox at which a virtual machine using Ubuntu version 18.04 was run. The space required for the OS installation is 16GB.

6.2 Mininet

Mininet is used to emulate the network with the functionality of all nodes. It creates virtual networks easily which are used to implement the procedures required of establishing an SDN. It supports Python 3 and installations of OVS perfectly to run the project. Mininet also allows transfer of the functionality to hardware if needed be but it is not the scope of this project [5].

6.3 Containernet

Containernet is used to add the Docker containers features in the emulated network. It is highly important while virtualizing [6]. The containernet in this project is used to create the network infrastructure of the project and deploy docker containers for various functions such as implementing the chat service on all data center servers present in the network and implementing of DHCP service on edge networks via installing it on an image in the docker and so on.

6.4 Docker

Docker is used in this project to install the packages required as images and deploy it in the network. The docker functionality is used via Containernet and the applications are run on the Linux VM. The whole process is deemed as lightweight using the Docker approach in the emulated environment which behaves as the SDN needed.

6.5 Openflow protocol

SDN is implemented using the OpenFlow Protocol for the fast configuration and flexibility nature of the network. It is a standard defined by Open Networking Foundation and the protocol allows an OpenFlow Controller to manage incoming packets on the OpenFlow switch [7].
OpenFlow is primarily a programmable protocol for SDN architecture that communicates as an interface between OpenFlow controllers and switches. The network device is programmed which benefits from quickly adapting to changes in the network while the underlying hardware is separated from the network device [7].

The OpenFlow architecture consists of an OpenFlow Switch (virtual or physical) a controller and the OpenFlow applications.

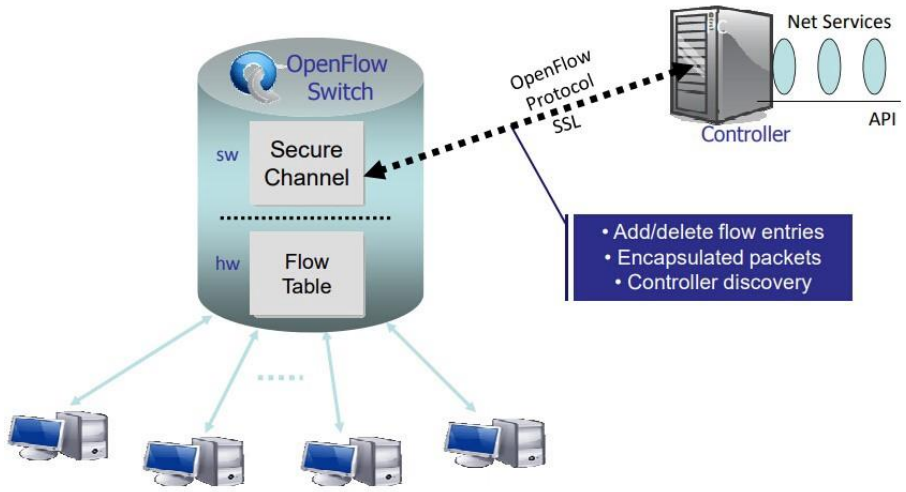

Figure 7 OpenFlow Switch [4]

The Openflow switch may be programmed to:

1. Categorize and identify the packets from the header fields.
2. Packet managing, which may include modifying the header.
3. Push or drop the packet to egress port or the controller accordingly.

The Openflow controller instructs the OpenFlow switch via flow. These flows consist of various types of information including a cookie, flow timeout, counters, flow priority, packet match field and the packet processing instructions. The controller also maintains a local state graph of the switches that communicate with the API exposed to the OpenFlow applications. This API allows a view of the network to the OpenFlow Applications so that to instruct the network of certain requirements or tasks [7].

The packets are processed by flows in multiple tables. This is a pipeline of tables situated on the Openflow Switch. The OpenFlow switch in this way separates the data plane with the control plane which is implemented in software in SDN. This coupled with the SDN controller allows high level routing decisions [7].

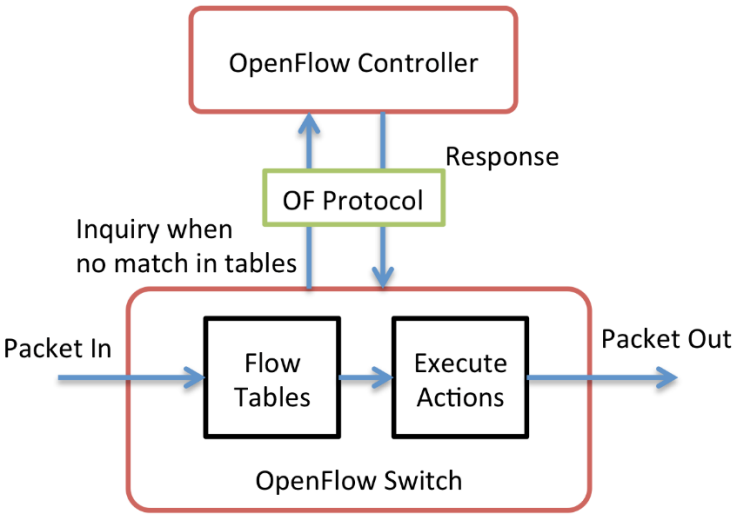

Figure 8 OpenFlow Protocol Architecture

The OpenFlow architecture works via the collaboration of the switches, the protocol maintaining the communication between the controller and switch and the controller. The switch functions by the controller imposing policies on the flow tables situated on the switch, this could be to set up optimized network paths regarding various characteristics such as speed, number of hops or latency [7].

6.6 Open virtual Switches

OpenvSwitch is an open-source application used to implement an OpenFlow switch. This is important for virtualized environments such as in SDN. An OpenSwitch is the perfect candidate to work in the scope of this Linux based virtualization [8].

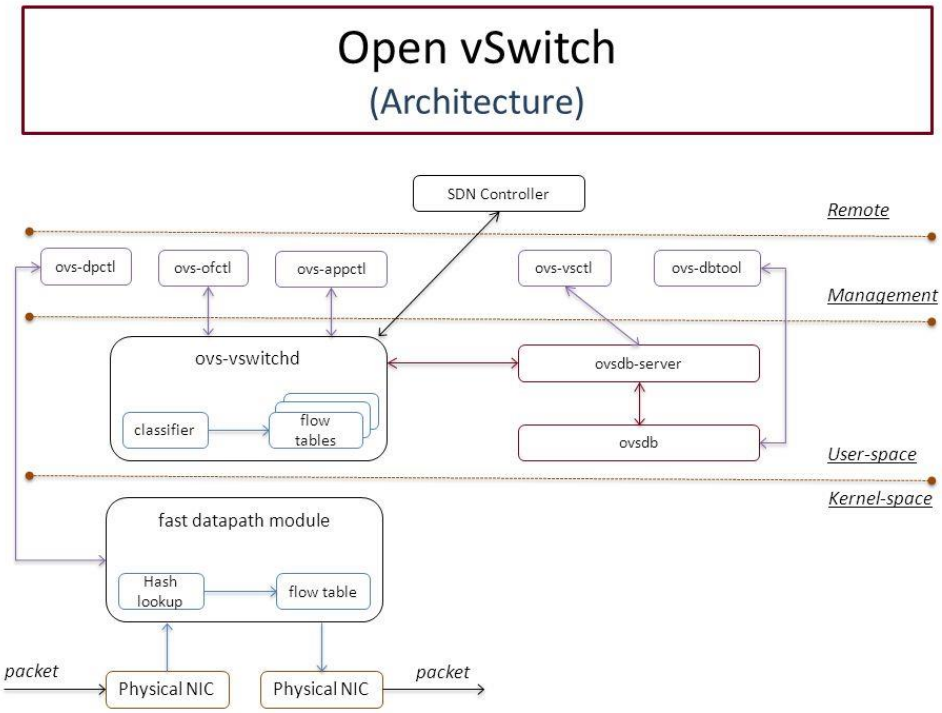

Figure 9 OpenvSwitch Architecture

OpenvSwitch consists of eight entities respectively [8]:

- Ovs-vswitchd: Daemon to implement the Openvswitch.
- Linux kernel module: For flow-based switching as in required for OpenFlow protocol.
- Ovsdb-server: A database server that is lightweight as compared to other servers.
- Ovs-dpctl: To configure switch kernel module.
- Ovs-vsctl: To query and update the configuaration of Ovs-vswitchd.
- Ovs-appctl: To send commands for running the Open vSwitch daemons.

- Ovs-ofctl: For controlling and compensation of features required for OpenFlow in OVS.

- Ovs-pki: To create and manage the public-key infrastructure.

To describe the flows, various actions and fields are used. Following is the list of these entities used in OVS regarding the flows in the project [9]:

- `Table` Number of the table could be between 0 and 254 inclusive. A name could also be described to the table.

- `in_port` The port at which the packet enters the switch

- Type: `arp` or `udp` used to discriminate the ARP or DHCP packet for their specific handling, respectively. (The detail of usage of these types will be explained in section 12.4)

- `nw_src` and `nw_dst` these fields are used to hash the source and destination address of the packet (used for ARP Broadcast handling and DHCP broadcast handling)

- `dl_dst` used to specify the destination of the packet. Used for destinations of data centers.

- `dl_type` used to identify the NSH type of the packet. For NSH the value is `0x894f`

- `nsh_spi` and `nsh_c1` metadata values for a NSH Service Path Identifier and Context header for the packets allowed in a specific slice. For our slice 1, the values are 1234 and 11223344 respectively.

- `actions` Before this field, all the fields are used for matching the flow with the packet header. At this field various actions can be defined. The actions we used are:

  - `encap()` encapsulates the header with a specific header. For e.g. ethernet or nsh header with metadata type 1 or with ethernet:

    ```
    actions=encap(nsh(md_type=1))
    ```

    ```
    actions=encap(ethernet)
    ```

  - `decap()` removes the outermost encapsulation of the packet

  - `set_field` has a metadata of 128 bits for a specific label such as `nsh_spi` and `nsh_c1`. Important for matching the flow in the table. For e.g., for slice 1 in our project

    ```
    set_field:0x1234->nsh_spi
    ```

    ```
    set_field:0x11223344->nsh_c1
    ```

In our project, the ovs switches are operating on fail-mode “secure”.
Following are some of the ovs commands that are used extensively in our project for testing purpose.

- `sh ovs-vsctl set-fail-mode standalone` (to set the fail-mode to standalone for testing and examine packets)
- `sh ovs-ofctl -Oopenflow13 add-flows bridgename`(to configure flow rules on a specific switch
- `sh ovs-ofctl -Oopenflow13 del-flows bridgename` (to delete previously added flows)
- `sh ovs-ofctl -Oopenflow13 dump-flows bridgename` (to verify the added flows)
- `sh ovs-vsctl show` (To check the interfaces/ports of all connected switches)

7. Implementation of Text based Chatroom service

The Chat room service used in the project is of existing implemented service by TomPrograms [10]. It consists of two python programs to be run on the host as client and datacenter as server. A change is made in the program. Instead of asking the datacenter IP, it is statically defined in the program server.py.

Steps in implementing the service:

1. Via command `docker run -it ubuntu : trustee /bin/bash` the docker is run and server.py file is created inside with nano server.py using the code given in the repository.
2. A client.py file is saved within the network infrastructure.
3. To run the service, xterm emulator of each host in a slice is called via command line for example by `xterm h1, xterm h2, xterm h3, xterm h4`.
4. The program server.py is run on the datacenter by first calling for xterm d1 and then running the program by `python3 server.py`.
5. The program runs and displays the datacenter IP while asking for the port number. By entering the port the server is ready to add the clients in chat room.
6. On xterms of hosts, client.py is run with command `python3 client.py`. The program asks for the host name to be connected to or the datacenter IP address it needs for communication. It the asks for username to register the user in the room.
7. The chatroom is primarily the xterm of the data center and all messages are displayed on the server window with each user's username.

The following image illustrates the simple running service with datacenter d1 where the service is deployed, and four hosts connected as nodes. The chatroom enlists all the messages with the username and the user can simultaneously see all the texts.

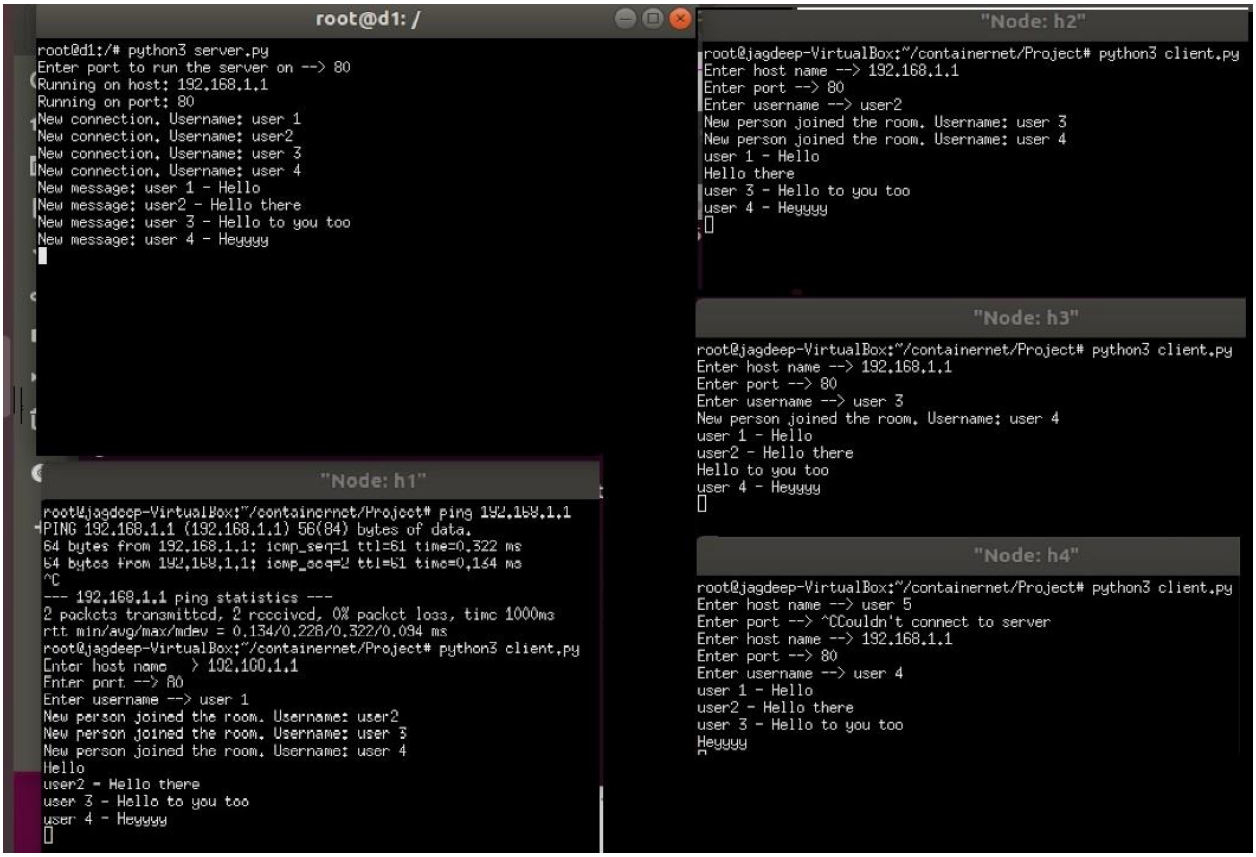

Figure 10 Implementation of Chat room service

The above image in the successful trial run program. The service running via slices will be illustrated later in the document in Implementation section 10.

8. Technologies used for Possible Extension

8.1 DHCP

DHCP service is added in all the edge networks of the network. For implementation three docker containers are created using ubunu:trusty images and a linux package “dnsmasq” is installed on each. Each image is named as dhcp1, dhcp2 and dhcp3. For every image the dnsmasq.conf file is edited. Following changes are made accordingly:

1. Individual IP address of the server is added as the listening address.
2. Interface is added at which the server is present.
3. DHCP range is statically provided accordingly to compensate all access networks.

DHCP Edge server	Listen IP address	Interface	Assigned Ranges
E1	172.10.0.20	e1-eth0	172.10.0.10,172.10.0.29
E2	172.10.0.30	e2-eth0	172.10.0.31,172.10.0.49
E3	172.10.0.50	e3-eth0	172.10.0.51,172.10.0.61

The functionality of DHCP with the running slices will be demonstrated later in the document in section 10.

8.2 Mininet WiFi

As known already that Containernet serves as a fork of mininet to enable the usage of docker containers in the emulation environment and Mininet WiFi stations. Mininet Wifi permits the utilization of WiFi stations and Access points. It provides the same functionalites as Open vSwitches but exists as access points or Wifi stations within the network emulation [11].

This possible extension is currently in testing phase and half of the results have been achieved. However, it is not added in current project emulation. Therefore, it will not be added in this document too. If this service is fully integrated in the project emulation, it will be demonstrated in the Project presentation.

9. Project Architecture/Framework

Emulated Network Description

The emulated network consists of three access networks, each with an edge network, and three datacenters or servers. Following is a descriptive map of the emulated network:

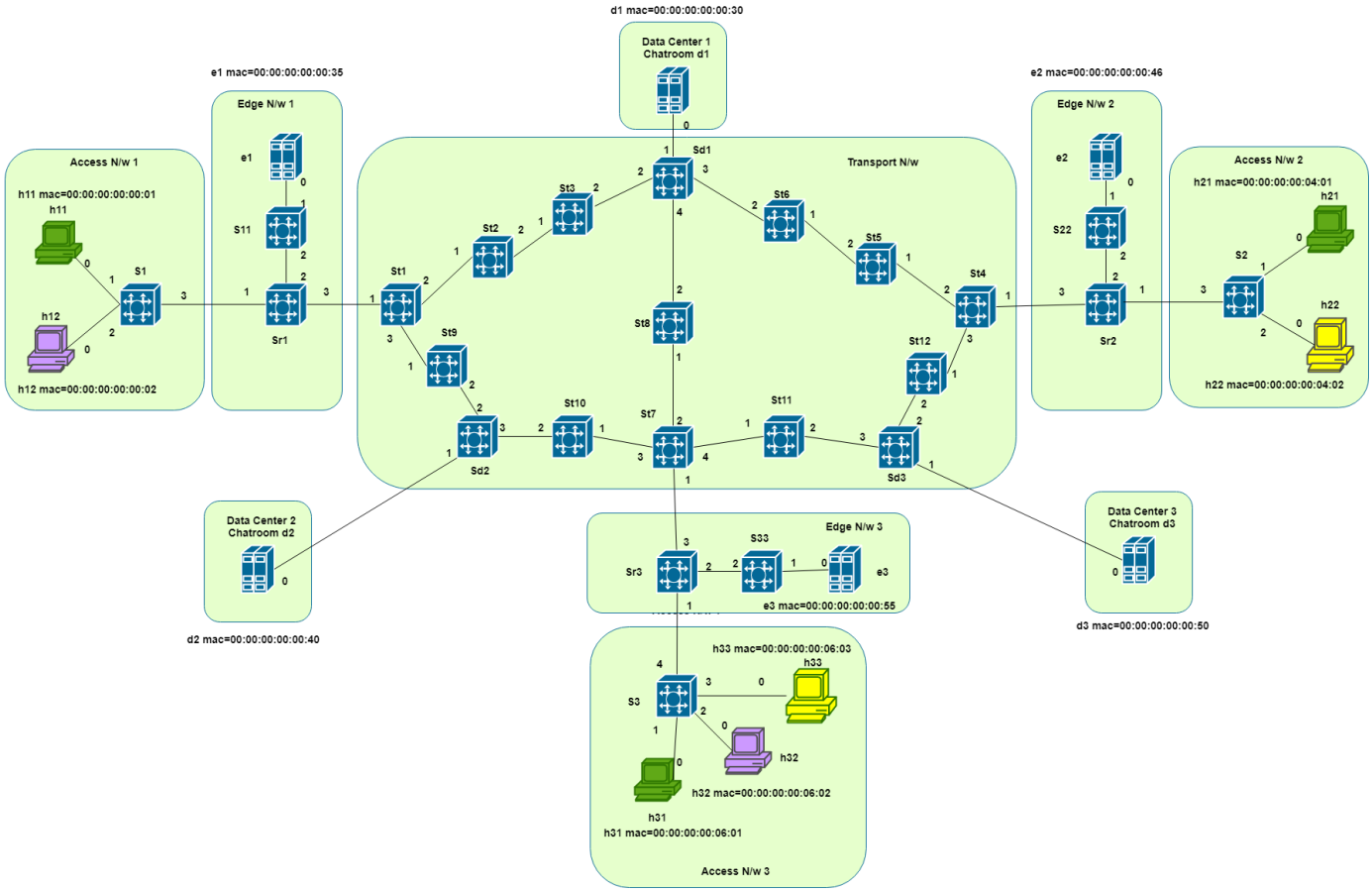

Figure 11 Network Emulation Architecture

This whole network emulation is pure OVS based network, so only one IP network is assigned to it which is "172.10.0.0/24"

Access Network 1:	Consists of hosts **h11** and **h12** connected to switch **s1**.
	s1 connects to edge network 1 by edge switch **sr1.**
	Edge network 1 consists of server **e1** via switch **s11.**
	Edge switch **sr1** connects to **st1** to enter the transport network.
Access Network 2:	Consists of hosts **h21** and **h22** connected to switch **s2**.
	s2 connects to edge network 2 by edge switch **sr2.**
	Edge network 2 consists of server **e2** via switch **s22.**
	Edge switch **sr2** connects to **st4** to enter the transport network.
Access Network 3:	Consists of hosts **h31**, **h32** and **h33** connected to switch **s3**.
	s3 connects to edge network 3 by dge switch **sr3.**
	Edge network 3 consists of server **e3** via switch **s33.**
	Edge switch **sr3** connects to **st7** to enter the transport network.
Transport network:	Consists of twelve transport switches in a ring structure with a diagonal structure.
Servers:	Three datacenter/ chatroom servers namely **d1**, **d2** and **d3** are connected via switches **sd1, sd2** and **sd3** integrated within the transport network.

The chat room service is deployed on datacenters **d1, d2** and **d3**. The DHCP service is deployed on edge servers **e1, e2** and **e3.** These services are implemented via containernet virtualization. The integrated switches are all OVS to implement OpenFlow protocol for SDN.

10. Implementation

10.1 Overview

As the core concept of Network Slicing is based on the rule of utilizing the same underlay or physical network infrastructure to create multiple different overlay/virtual network sections. The overly network contains all the required virtual/logical entities or virtual network functions to provide different services to the tenants within their network slice. On the basis of this concept, total three network slices are created from the network infrastructure presented in the section above. To achieve these network slices, the networking approach used is based of Network Service Header.

The NSH protocol explanation earlier in this document states that there are different NSH fields that can be used to achieve the goal of network slicing. However, in this project, two NSH fields are utilized, which are

Service Path Identifier “NSH_SPI” and Context header 1 “NSH_C1”. The values assigned to these fields are different for all three network slices in order to distinguish the data traffic of these slice.

10.2 General procedure of Network Slicing via NSH

To create a logical network slice using NSH based networking approach, the very first open virual Switch (OvS) of the access network, which is connected to the host/tenant directly, encapsulates the original frame coming from the host in the direction of server (either DHCP or Chatroom) in the NSH header and assign values to the nsh fields “nsh_spi” and “nsh_c1”. NSH header requires a transport header to steer the NSH encapsulated packets across the defined network path. The outer transport header used in this project is Ethernet II. Therefore, after the encapsulation of original packet into NSH, this NSH encapsulated packet is again encapsulated in EthernetII based transport header and the destination mac address is set in the “dl_dst” field. This double encapsulation takes place at the OvS which is connected to end nodes directly, and then this packet is forwarded to transport network. The OvSs in the transport network just forward the packets in the defined path towards the destination. These transport network OvSs make the decision to transport the packets across the network based on the assigned flow rules. This NSH+EthernetII encapsulated packet when reaches at the OvS which is connected to the destination entity, is decapsulated twice to remove NSH and outer transport EthernetII header and then forwarded to the end node, which can be either DHCP server at the edge network or Chatroom server at the data center. In the similar manner, the encapsulation of NSH/EthernetII and decapsulation is performed the other direction too.

10.3 Implementation of Slices in project

On the above explained approach, three network slices are created in this project. The detailed implementation of these network slices is presented in the following subsections.

10.3.1 Configuaration of Network Slice 1

In the first network slice, the tenant h11 of access network 1, tenant h21 of access network 2 and tenant h31 of access network 3 are communicating with each other via the Chatroom located at data center d1, as shown in the figure (1). The tenant h11 uses the DHCP service to get an IP from its edge server e1/DHCP1 located in its edge network. Similarly, host h21 uses DHCP service of edge server e2/DHCP2 and host h31 from e3/DHCP3. To identify the traffic (packets) within the network slice 1, the NSH fields “NSH-SPI=1234” & “NSH-C1=11223344” are assigned to all the IP packets.

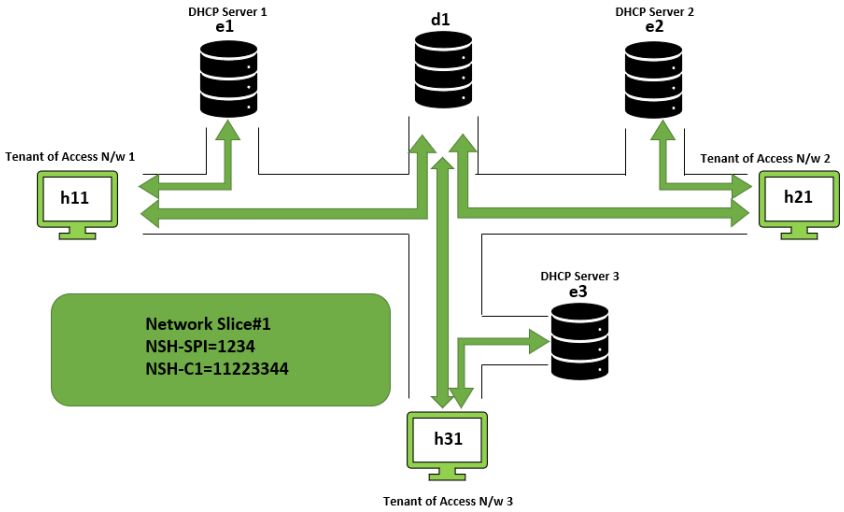

Figure 12 Pictorial Representation of Slice 1

End nodes	Ids	Network Interface	Mac Address	IP Address
Tenant of Access n/w 1	H11	H11-eth0	00:00:00:00:00:01	assigned by e1
Tenant of Access n/w 2	H21	H21-eth0	00:00:00:00:04:01	assigned by e2
Tenant of Access n/w 3	H31	H31-eth0	00:00:00:00:06:01	assigned by e3
Data Center (Chatroom server)	D1	D1-eth0	00:00:00:00:00:30	172.10.0.1/24
Edge server at Edge n/w 1	E1	E1-eth0	00:00:00:00:00:35	172.10.0.20/24
Edge server at Edge n/w 2	E2	E2-eth0	00:00:00:00:00:46	172.10.0.30/24
Edge server at Edge n/w 3	E3	E3-eth0	00:00:00:00:00:55	172.10.0.50/24

Defined Paths for Slice 1 "NSH-SPI=1234" & "NSH-C1=11223344"	
H11 to e1	h11-s1-sr1-s11-e1
H11 to d1	h11-s1-sr1-st1-st2-st3-sd1-d1
H21 to e2	h21-s2-sr2-s22-e2
H21 to d1	h21-s2-sr2-st4-st5-st6-sd1-d1
H31 to e3	h31-s3-sr3-s33-e3
H31 to d1	h31-s3-sr3-st7-st8-sd1-d1

10.3.2 OvS Rules of Slice 1

As stated earlier, OvS switches manage the data packets as per the defined flow rules. Therefore, each OvS in the defined path is configured with specific flow rules to create network slice 1. In order to explain the functionality of these configured rules on switches, the path from h11 to e1 (tenant to edge server) and h11 to d1 (tenant to datacenter1) is explained as follows:

Flows rules to manage ARP & DHCP packets (h11 to e1).

- At first, Tenant h11 of access network 1 wants to use DHCP server from Edge network e1 to get the IP address by using the command `dhclient h11-eth0`.
- As soon as h11 runs this command, the first packet received at s1-eth1 would be ARP-packet. Since, the ARP packet encapsulation in NSH was not achieved in this project, all ARP-packets will be broadcasted normally by s1 to all the nodes. The ARP-Packet with NSH encapsulation issue is discussed later in the section 12.4. The flow rule configured at s1 to manage ARP packet is as follows:

```
table=0,in_port=1,arp,nw_src=172.10.0.0/24,nw_dst=172.10.0.0/24,actions
  =local,flood,3

table=0,in_port=3,arp,nw_src=172.10.0.0/24,nw_dst=172.10.0.0/24,actions
  =local,flood,1
```

- Similarly, all OvS switches in the entire network topology are configured with these rules to handle ARP packets, which caused the looping issue, as discussed later in the section 12.4.
- After ARP, s1-eth1 would receive the DHCP Discover packet, which is again forwarded normally to the next switch from port 3. The issue related to DHCP packet encapsulation is discussed in the section 12.5. The flow rule configured at s1 to manage DHCP discover and offer packet is as follow;

```
table=0,in_port=1,udp,nw_dst=255.255.255.255,udp_dst=67,actions=output
  :3
```

- sr1 and s11 at the edge network are also configured with the above stated rule due to which sr1 will direct all the DHCP discover and request packets to s11 via the output port sr1-eth2. Similarly, s11 forward these packets to e1.
- Edge server e1 then will process the request and replies with the DHCP offer packet, which will be received at s11-eth1.
- At this interface, the DHCP offer packet will be processed as per the following configured flow rules and encapsulation of NSH+EthernetII on the received packet will take place. In the similar manner, the DHCP ack packets will be handled.

```
table=0,in_port=1,dl_dst=00:00:00:00:00:01,actions=encap(nsh(md_type=1
 )),set_field:0x1234->nsh_spi,set_field:0x11223344-
 >nsh_c1,encap(ethernet),set_field:00:00:00:00:00:01->dl_dst,2
```

This rule directs the ovs switch s11 that, when an IP packet with destination mac address 00:00:00:00:00:01 (h11's mac address) receives at interface s11-eth1, encapsulate it with NSH header, set the nsh fields "NSH-SPI=1234" & "NSH-C1=11223344" to indicate slice 1, then again encapsulate this packet with EthernetII header and set the mac address again. After all this processing, s11 would send this packet to sr1 from s11-eth2.

- The configured rule at sr1 directs that when sr1-eth2 receives a packet with "dl-type=0x894f, NSH-SPI=1234 & NSH-C1=11223344", then forward it to s1 from sr1-eth-1.

```
table=0,in_port=2,dl_type=0x894f,nsh_mdtype=1,nsh_spi=0x1234,nsh_c1=0
 x11223344,actions=output:1
```

- This NSH+EhernetII encapsulated packet when reaches at s1-eth3, s1 decapsulates the NSH and Ethernet II header and based on the matching of dl_dst (mac address field), it forwards the original DHCP offer packet to h11 from s1-eth1.

```
table=0,in_port=3,dl_dst=00:00:00:00:00:01,dl_type=0x894f,nsh_mdtype=1,ns
  h_spi=0x1234,nsh_c1=0x11223344,actions=decap(),decap(),1
```

- The DHCP packets exchange from h11 to e1 is also depicted in the following Message Sequence Chart (MSC).
- The flow rules are configured in the similar sense for tenant h21 to edge DHCP server e2 and tenant h31 to edge DHCP server e3.

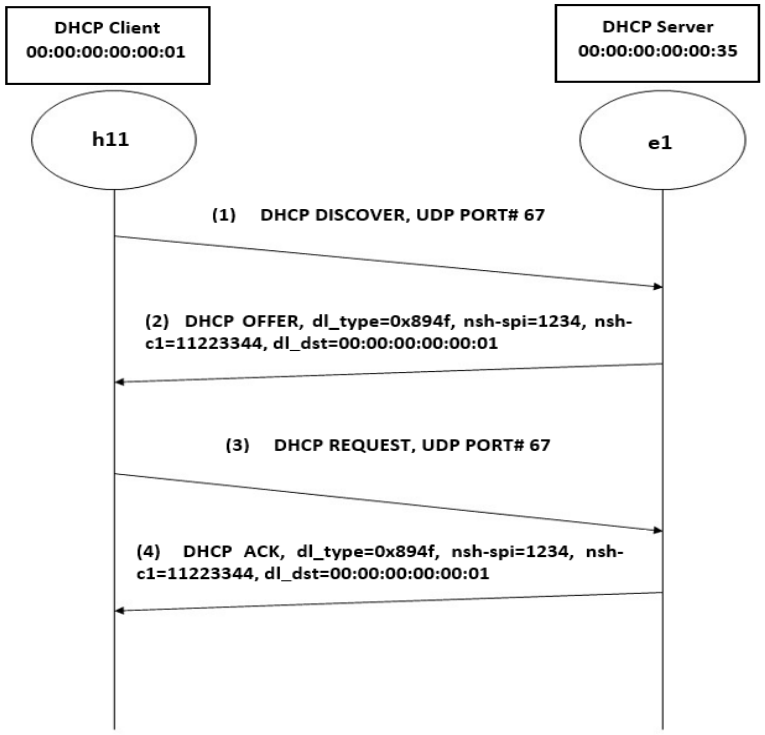

Figure 13 MSC of DHCP service of host 11 in slice 1

Flows rules to manage IP packets (h11 to d1);

- When h11 send the IP packets in the direction of chatroom server d1, it receives at s1-eth1. S1 first check the dl_dst field, which will be the mac address of d1, based on which it processes the packet with the following configured rule.

```
table=0,in_port=1,dl_dst=00:00:00:00:00:30,actions=encap(nsh(md_type=1)
  ),set_field:0x1234->nsh_spi,set_field:0x11223344-
  >nsh_c1,encap(ethernet),set_field:00:00:00:00:00:30->dl_dst,3
```

- The above rule directs the s1 to match the dl_dst field and then encapsulate it with NSH+EthernetII headers. Set the values of NSH fields of slice 1 and forward it to sr1 via s1-eth3.
- Sr1 will process the packet in the same way by matching dl_dst_dl_type and nsh fields and forward the packet to transport network from sr1-eth3.

```
table=0,in_port=1,dl_dst=00:00:00:00:00:30,dl_type=0x894f,nsh_mdtype=1,
  nsh_spi=0x1234,nsh_c1=0x11223344,actions=output:3
```

- The transport switches st1, st2 & st3 will also work on match and forward rule as following:

```
table=0,in_port=1,dl_dst=00:00:00:00:00:30,dl_type=0x894f,nsh_mdtype=1,
   nsh_spi=0x1234,nsh_c1=0x11223344,actions=output:2
```

- When this packet reaches at sd1_eth2, it checks the dl_dst and nsh fields and if these fields match, it decapsulates the outer ethernet II header, then NSH header and forwards the original packet to the datacenter d1 where the chatroom server is located.
- Sd1 deals with all the incoming packets from port 2,3 and 4 in the same way on the basis of following configured rules and forward it the d1 from sd1-eth1;

```
table=0,in_port=2,dl_type=0x894f,nsh_mdtype=1,nsh_spi=0x1234,nsh_c1=0x1
  1223344,actions=decap(),decap(),1
table=0,in_port=3,dl_type=0x894f,nsh_mdtype=1,nsh_spi=0x1234,nsh_c1=0x1
  1223344,actions=decap(),decap(),1
table=0,in_port=4,dl_type=0x894f,nsh_mdtype=1,nsh_spi=0x1234,nsh_c1=0x1
  1223344,actions=decap(),decap(),1
```

- On the way back, when packets from d1 receives at sd1-eth1, it encapsulated the original packet in NSH+EthernetII back and forward it to the ports directed towards the transport network on the basis of dl_dst (mac address of tenants).
- Again st3, st2, st1, sr1 will match and forward the packet until it receives at the s1 eth-3. Here, the decapsulation of NSH+EthernetII header is performed and the original frame coming from d1 is forwarded to h11.
- The flow rules are configured in the similar sense for tenant h21 to Chatroom server d1 and and tenant h31 to d1.

10.3.3 Configuration of Network Slice 2

In the second network slice, the tenant h12 of access network 2 and tenant h32 of access network 3 are communicating with each other via the Chatroom located at data center d2, as shown in the figure (2). The tenant h12 uses the DHCP service to get an IP from its edge server e2/DHCP2 located in its edge network. Similarly, host h32 uses DHCP service of edge server e3/DHCP3. To identify the traffic (packets) within the network slice 2, the NSH fields "NSH-SPI=5678" & "NSH-C1=55667788" are assigned to all the IP packets.

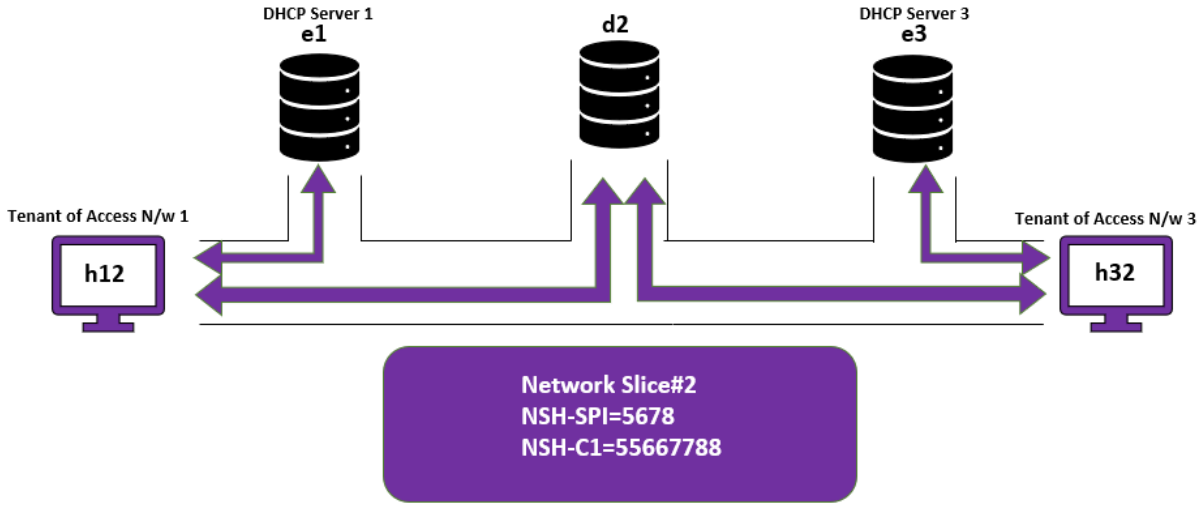

Figure 14 Pictorial Representation of Slice 2

End nodes	Ids	Network Interface	Mac Address	IP Address
Tenant of Access n/w 2	H12	H12-eth0	00:00:00:00:00:02	assigned by e1
Tenant of Access n/w 3	H32	H32-eth0	00:00:00:00:06:02	assigned by e3
Data Center (Chatroom server)	D2	D2-eth0	00:00:00:00:00:40	172.10.0.2/24
Edge server at Edge n/w 2	E2	E2-eth0	00:00:00:00:00:46	172.10.0.30/24
Edge server at Edge n/w 3	E3	E3-eth0	00:00:00:00:00:55	172.10.0.50/24

Defined Paths for Slice 2 “NSH-SPI=5678” & “NSH-C1=55667788”	
H12 to e1	h12-s1-sr1-s11-e1
H12 to d2	h12-s1-sr1-st1-st9-sd2-d2
H32 to e3	h32-s3-sr3-s33-e3
H32 to d2	h32-s3-sr3-st7-st10-sd2-d2

10.3.4 OvS Rules of Slice 2

The flow rules making the path from h12 to e1 (tenant to edge server) and h12 to d2 (tenant to datacenter2) is explained as follows:

Flows rules to manage ARP & DHCP packets (h12 to e1).

- For address resolution of host h12 is dealt in the similar manner as that in slice 1, the flow rule configured at s1 to manage ARP packet from client h12 is as follows:

```
table=0,in_port=2,arp,nw_src=172.10.0.0/24,nw_dst=172.10.0.0/24,acti
ons=local,flood,3

table=0,in_port=3,arp,nw_src=172.10.0.0/24,nw_dst=172.10.0.0/24,acti
ons=local,flood,2
```

  It is to be observed, only the port numbers are configured according to the position of host h12 at s1.

- Similarly, the DHCP discover packet from host h12 is matched with the following rule. Again only the port number is changed:

```
table=0,in_port=2,udp,nw_dst=255.255.255.255,udp_dst=67,actions=outp
ut:3
```

- sr1 and s11 at the edge network are also configured with the above stated rule to direct the packets to e1. The DHCP offer packet then will be received at s11-eth2.

- s11 will encapsulate the DHCP offer packets with NSH as in slice 1 but with the slice specific nsh_spi and nsh_c1. The DHCP ACK will be handled in similar manner.

```
table=0,in_port=1,dl_dst=00:00:00:00:00:02,actions=encap(nsh(md_type
=1)),set_field:0x5678->nsh_spi,set_field:0x55667788-
>nsh_c1,encap(ethernet),set_field:00:00:00:00:00:02->dl_dst,2
```

- The configured rule at sr1 directs that when sr1-eth2 receives a packet with “dl-type=0x894f, NSH-SPI=5678 & NSH-C1=55667788”, then forward it to s1 from sr1-eth-1.

```
table=0,in_port=2,dl_type=0x894f,nsh_mdtype=1,nsh_spi=0x5678,nsh_c1=
0x55667788,actions=output:1
```

- This NSH+EhernetII encapsulated packet when reaches at s1-eth3, s1 decapsulates the NSH and Ethernet II header and based on the matching of dl_dst (mac address field), it forwards the original DHCP offer packet to h12 from s1-eth2.

```
table=0,in_port=3,dl_dst=00:00:00:00:00:02,dl_type=0x894f,nsh_mdtype
=1,nsh_spi=0x5678,nsh_c1=0x55667788,actions=decap(),decap(),2
```

The DHCP packets exchange from h12 to e1 is also depicted in the following Message Sequence Chart (MSC). The flow rules are configured in the similar sense for tenant h32 to edge DHCP server e3.

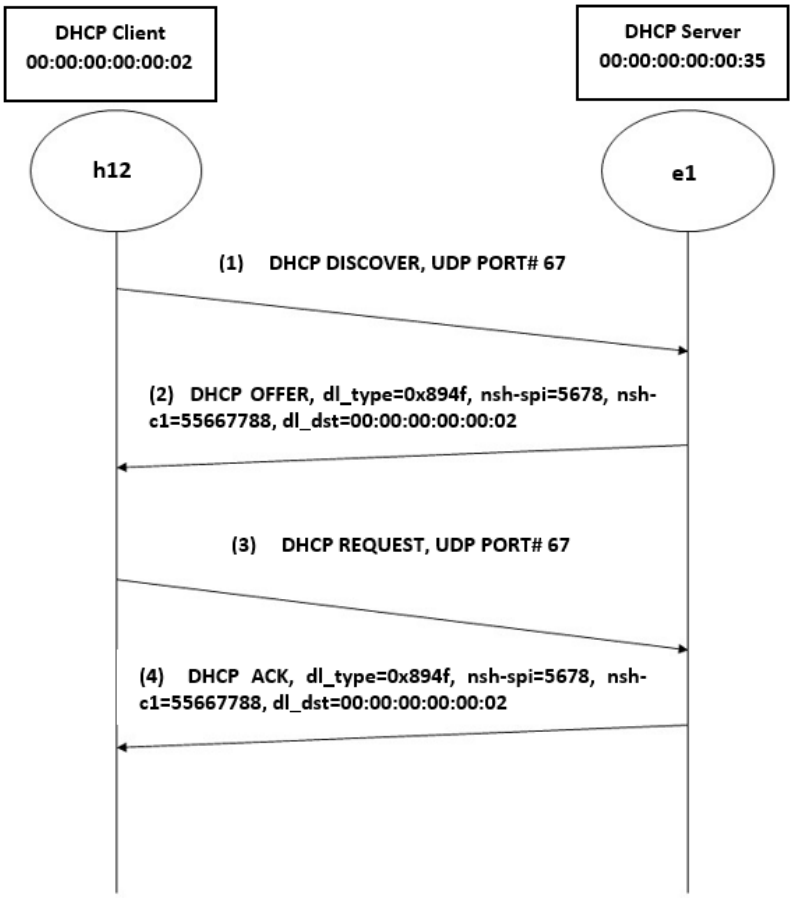

Figure 15 MSC of DHCP service of host 12 in slice 2

Flows rules to manage IP packets (h12 to d2);

At end user h12, OvS s1 connected to h12, is responsible for encapsulating the packets with NSH header that are leaving for datacenter2 with destination address 00:00:00:00:00:40 with slice 2 specific tags of nsh_spi 0x5678 and nsh_c1 0x55667788. Similarly, all packets received at s1 for h12 are decapsulated to forward to h12.

At service point datacenter d2, OvS sd2 decapsulates the NSH header coming from the host h12 and encapsulates the packet coming from d2 to move in the transport network with NSH tags of slice 2 i.e. fields nsh_spi and nsh_c1.

The details are as follows:

- When h12 send the IP packets in the direction of chatroom server d2, it receives at s1-eth2. S1 first check the dl_dst field, which will be the mac address of d2, based on which it processes the packet with the following configured rule.

```
table=0,in_port=2,dl_dst=00:00:00:00:00:40,actions=encap(nsh(md_type
=1)),set_field:0x5678->nsh_spi,set_field:0x55667788-
>nsh_c1,encap(ethernet),set_field:00:00:00:00:00:40->dl_dst,3
```

- The above rule directs the s1 to match the dl_dst field and then encapsulate it with NSH+EthernetII headers. Set the values of NSH fields of slice 2 and forward it to sr1 via s1-eth3.

- Sr1 will process the packet in the same way by matching dl_dst_dl_type and nsh fields and forward the packet to transport network from sr1-eth3.

```
table=0,in_port=1,dl_dst=00:00:00:00:00:40,dl_type=0x894f,nsh_mdtype
=1,nsh_spi=0x5678,nsh_c1=0x55667788,actions=output:3
```

- The transport switches st9 will work on match and forward rule as following;

```
table=0,in_port=1,dl_dst=00:00:00:00:00:40,dl_type=0x894f,nsh_mdtype
=1,nsh_spi=0x5678,nsh_c1=0x55667788,actions=output:2
```

- When this packet reaches at sd2_eth2, it checks the dl_dst and nsh fields and if these fields match, it decapsulates the outer ethernet II header, then NSH header and forwards the original packet to the datacenter d2 where the chatroom server is located.

- Sd2 deals with all the incoming packets from port 2 and 3 and in the same way on the basis of following configured rules and forward it to d2 from sd2-eth1;

```
table=0,in_port=2,dl_type=0x894f,nsh_mdtype=1,nsh_spi=0x5678,nsh_c1=
0x55667788,actions=decap(),decap(),1

table=0,in_port=3,dl_type=0x894f,nsh_mdtype=1,nsh_spi=0x5678,nsh_c1=
0x55667788,actions=decap(),decap(),1
```

- On the way back, when packets from d2 received at sd2-eth1, they are encapsulated in NSH+EthernetII back and forward it to the ports directed towards the transport network on the basis of dl_dst (mac address of tenants).

- Again st9 and sr1 will match and forward the packet until it receives at the s1 eth-3. Here, the decapsulation of NSH+EthernetII header is performed and the original frame coming from d2 is forwarded to h12.

- The flow rules are configured in the similar sense for tenant h32 to Chatroom server d2.

10.3.5 Configuration of Network Slice 3

The third slice consists of host h22 and h33. Similar to slices 1 and slice 2 the DHCP service and datacenter service is handled.

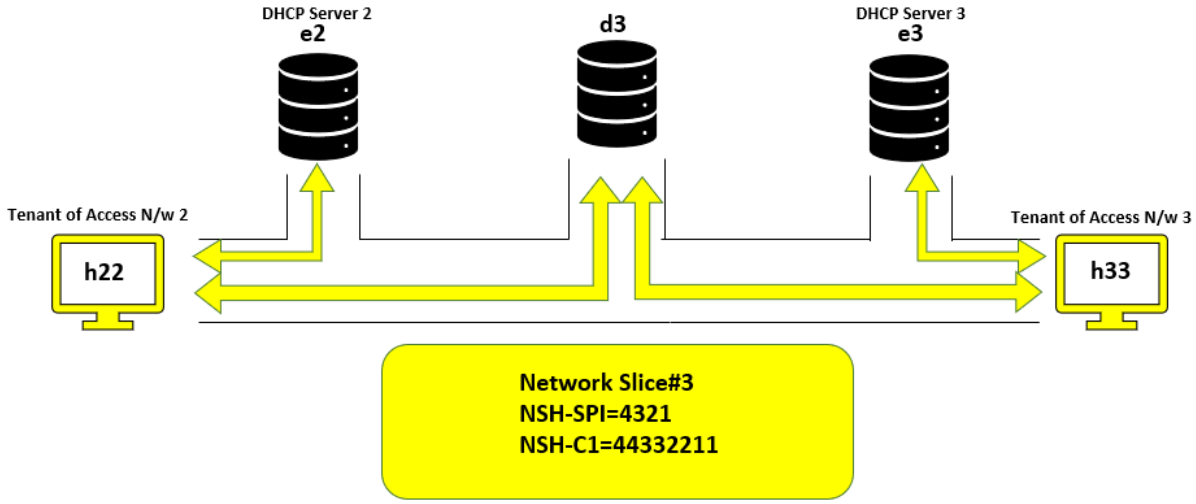

Figure 16 Pictorial Representation of Slice 3

End nodes	Ids	Network Interface	Mac Address	IP Address
Tenant of Access n/w 2	H22	H22-eth0	00:00:00:00:04:02	assigned by e2
Tenant of Access n/w 3	H33	H33-eth0	00:00:00:00:06:03	assigned by e3
Data Center (Chatroom server)	D3	D3-eth0	00:00:00:00:00:50	172.10.0.3/24
Edge server at Edge n/w 2	E2	E2-eth0	00:00:00:00:00:46	172.10.0.30/24
Edge server at Edge n/w 3	E3	E3-eth0	00:00:00:00:00:55	172.10.0.50/24

Defined Paths for Slice 2 “NSH-SPI=4321” & “NSH-C1=44332211”	
H22 to e2	H22-s2-sr2-s22-e2
H22 to d3	H22-s2-sr2-st4-st12-sd3-d3
H33 to e3	h33-s3-sr3-s33-e3
H33 to d3	h33-s3-sr3-st7-st11-sd3-d3

10.3.6 OvS Rules of Slice 3

The flow rules making the path from h33 to e3 (tenant to edge server) are explained:

Flows rules to manage ARP & DHCP packets (h33 to e3).

- For address resolution of host h33 for slice 3, the flow rule configured at s3 to manage **ARP packet** is as follows:

```
table=0,in_port=3,arp,nw_src=172.10.0.0/24,nw_dst=172.10.0.0/24,acti
ons=local,flood,4

table=0,in_port=4,arp,nw_src=172.10.0.0/24,nw_dst=172.10.0.0/24,acti
ons=local,flood,3
```

It is to be observed, only the port numbers are configured according to the position of host h33 at s3. That is at receiving an ARP from host, deliver it to edge switch sr3 and vice versa.

- Similarly, the **DHCP discover packet** from host h33 is matched with the following rule. Again only the port number is changed:

```
table=0,in_port=3,udp,nw_dst=255.255.255.255,udp_dst=67,actions=outp
ut:4
```

- Sr3 and s33 at the edge network are also configured with the above stated rule to direct the packets to e3. The **DHCP offer packet** then will be received at s33-eth2.

- S33 will encapsulate the DHCP offer packets with NSH with the slice specific nsh_spi and nsh_c1. The DHCP ACK will be handled in similar manner.

```
table=0,in_port=1,dl_dst=00:00:00:00:06:03,actions=encap(nsh(md_type
=1)),set_field:0x4321->nsh_spi,set_field:0x44332211-
>nsh_c1,encap(ethernet),set_field:00:00:00:00:06:03->dl_dst,2
```

- The configured rule at sr3 directs that when sr3-eth2 receives a packet with "dl-type=0x894f, NSH-SPI=4321 & NSH-C1=44332211", then forward it to s3 from sr3-eth-1.

```
table=0,in_port=2,dl_type=0x894f,nsh_mdtype=1,nsh_spi=0x4321,nsh_c1=
0x44332211,actions=output:1
```

- This NSH+EhernetII encapsulated packet when reaches at s3-eth4, s1 decapsulates the NSH and Ethernet II header and based on the matching of dl_dst (mac address field), it forwards the original **DHCP offer packet** to h33 from s3-eth3.

```
table=0,in_port=4,dl_dst=00:00:00:00:06:03,dl_type=0x894f,nsh_mdtype
=1,nsh_spi=0x4321,nsh_c1=0x44332211,actions=decap(),decap(),3
```

The DHCP packets exchange from h33 to e3 is also depicted in the following Message Sequence Chart (MSC). The flow rules are configured in the similar sense for tenant h22 to edge DHCP server e2.

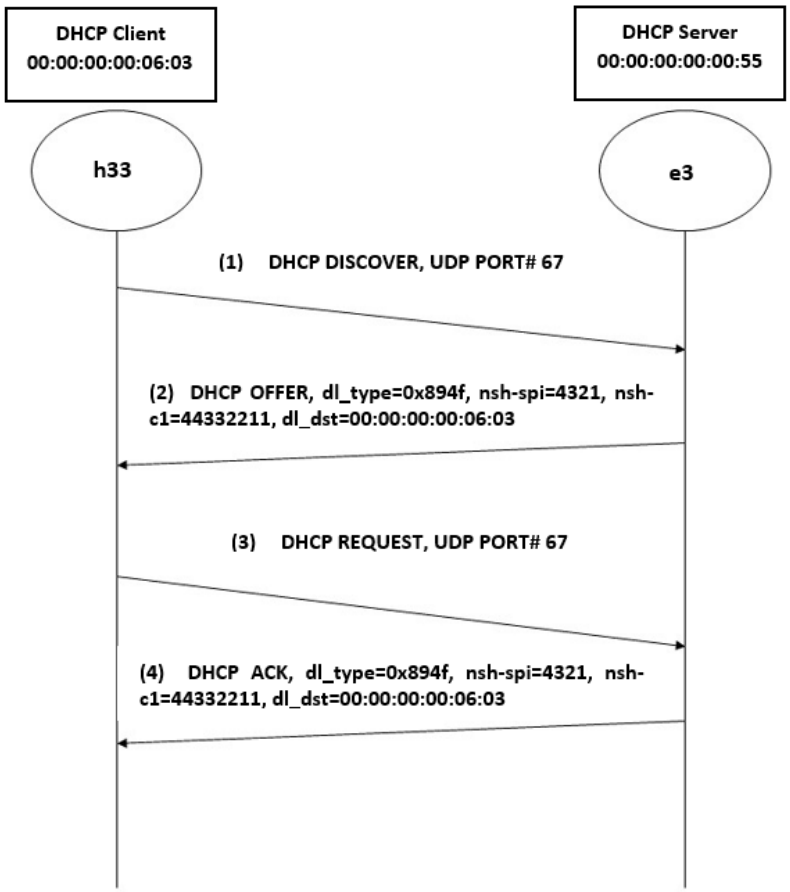

Figure 17 MSC of DHCP service of host 33 in slice 3

Flows rules to manage IP packets (h33 to d3);

At end user h33, OvS s3 is responsible for encapsulating the packets with NSH header that are leaving for datacenter3 with destination address 00:00:00:00:00:50 with slice 3 specific tags of nsh_spi 0x4321 and nsh_c1 0x44332211. Similarly, all packets received at s3 for h33 are decapsulated to forward to h33.

At service point datacenter d3, OvS sd3 decapsulates the NSH header coming from the host h33 and encapsulates the packet coming from d3 to move in the transport network with NSH tags of slice 3 i.e. fields nsh_spi and nsh_c1.

The details are as follows:

- When h33 sendS the IP packets in the direction of chatroom server d3, it receives at s3-eth3. S3 first check the dl_dst field, which will be the mac address of d3, based on which it processes the packet with the following configured rule.

  ```
  table=0,in_port=3,dl_dst=00:00:00:00:00:50,actions=encap(nsh(md_type
  =1)),set_field:0x4321->nsh_spi,set_field:0x44332211-
  >nsh_c1,encap(ethernet),set_field:00:00:00:00:00:50->dl_dst,4
  ```

  The above rule directs the s3 to match the dl_dst field and then encapsulate it with NSH+EthernetII headers. Set the values of NSH fields of slice 3 and forward it to sr3 via s3-eth4.

- Sr3 will process the packet in the same way by matching dl_dst_dl_type and nsh fields and forward the packet to transport network from sr3-eth3.

```
table=0,in_port=1,dl_dst=00:00:00:00:00:50,dl_type=0x894f,nsh_mdtype
=1,nsh_spi=0x4321,nsh_c1=0x4332211,actions=output:3
```

- The transport switches st7 and st11 will work on match and forward rule as following;

```
table=0,in_port=1,dl_dst=00:00:00:00:00:50,dl_type=0x894f,nsh_mdtype
=1,nsh_spi=0x4321,nsh_c1=0x44332211,actions=output:2
```

- When this packet reaches at sd3_eth3, it checks the dl_dst and nsh fields and if these fields match, it decapsulates the outer ethernet II header, then NSH header and forwards the original packet to the datacenter d3 where the chatroom server is located.

- Sd3 deals with all the incoming packets from port 2 and 3 and in the same way on the basis of following configured rules and forward it to d3 from sd3-eth1;

```
table=0,in_port=2,dl_type=0x894f,nsh_mdtype=1,nsh_spi=0x4321,nsh_c1=
0x44332211,actions=decap(),decap(),1
```

```
table=0,in_port=3,dl_type=0x894f,nsh_mdtype=1,nsh_spi=0x4321,nsh_c1=
0x44332211,actions=decap(),decap(),1
```

- On the way back, when packets from d3 are received at sd3-eth1, they are encapsulated in NSH+EthernetII back and forward it to the ports directed towards the transport network on the basis of dl_dst (mac address of tenants).

- Again st11, st7 and sr3 will match and forward the packet until it receives at the s3 eth-4. Here, the decapsulation of NSH+EthernetII header is performed and the original frame coming from d3 is forwarded to h33.

- The flow rules are configured in the similar sense for tenant h22 to Chatroom server d3.

10.4 Guidelines to deploy the Project

1. To run the configured network emulation for this project, Mininet, Containernet and ovs-vswitchd (Open vSwitch) version 2.9.8 must be pre-installed in the VM.
2. Wireshark must be installed in the system to analyze the traffic.
3. The docker images required to run in the emulation are already pushed on Dockerhub public repository. Therefore, whenever the network emulation runs, the images will automatically be pulled in the system.
4. The folder “Implementation” must be moved as it is to any folder within the containernet directory in the testing system/VM.
5. In the folder “Implementation”, the python script “nsh_project.py” is placed. This file contains the configured network emulation.

6. This “Implementation” folder also contains the flow rules configuration files (.txt) for each Open virtual switch, which will be called when the network topology runs. These files must be inside the similar folder where the nsh_project.py file is present.
7. A file “client.py” is also located in the “Implementation” folder. Save this file also in the similar folder where the nsh_project.py file is present.
8. Now open the terminal of the system and go inside the folder where all the files are saved.
9. Run the “nsh_project.py” file with the command “sudo python3 nsh_project.py”.
10. This will start the configured network emulation of this project. The images that required to be run on edge network’s dockers (e1, e2, e3) and Data centers (d1, d2, d3) will be pulled automatically in the running emulation infrastructure and the command line would look like the following figure.

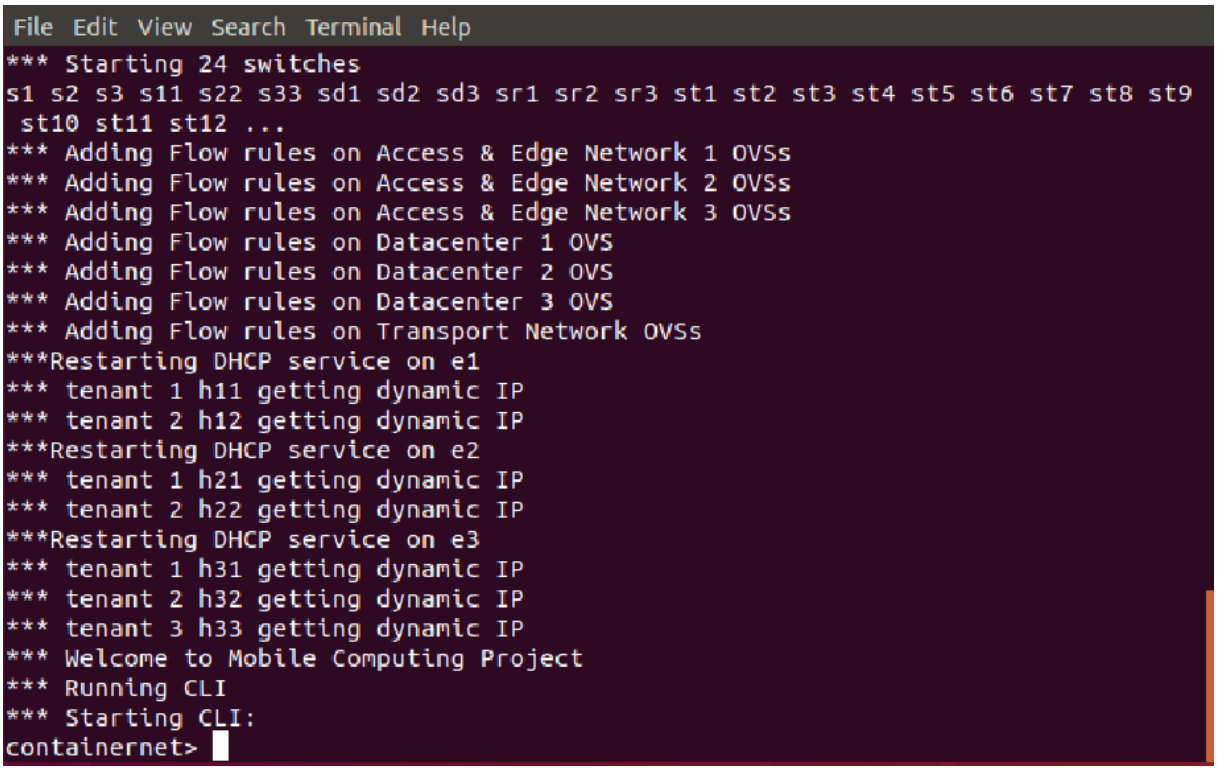

Figure 18 Command line at emulated network

11. Before utilization of DHCP service by hosts, the service “dnsmasq” will be restarted in their respective docker containers (e1, e2, e3).
12. For DHCP services utilization by the hosts of respective access networks, the command “dhclient host.ethx” is already configured in the emulation. Therefore, it is most likely possible that all the hosts already got their dynamic Ips.
13. To analyze the DHCP traffic within the slices, comment out the configured “dhclient” commands. Run the wireshark on the respective interfaces to check the packet´s encapsulation within their defined network slices.

Network Slices	**Access & Edge N/w**	**DHCP client**	**DHCP server**	**Traffic analysis interface**
Slice 1 /Spi=1234, c1=11223344	1	h11	e1	sr1-eth1 or sr1-eth2
	2	h21	e2	sr2-eth1 or sr2-eth2

	3	h31	e3	sr3-eth1 or sr3-eth2
Slice 2/ Spi=5678, c1=55667788	1	h12	e1	sr1-eth1 or sr1-eth2
	3	h32	e3	sr3-eth1 or sr3-eth2
Slice 3 / Spi=4321, c1=44332211	2	h22	e2	sr2-eth1 or sr2-eth2
	3	h33	e3	sr3-eth1 or sr3-eth2

14. Take the terminal of the desired host (DHCP client) eg. h11. Run the command "ifconfig h11-eth0 0" on the h11 terminal. Take the terminal of e1 (DHCP server), restart the service "dnsmasq". After this, run the command "dhclient h11-eth0". The Wireshark should capture the NSH encapsulated DHCP OFFER and DHCP ACK packets, which will be identified as network slice 1.
15. For the Chatroom service, first take the xterm of the datacenter, eg. d1. Run the python script "server.py" on d1. It will ask "Enter port to run the server on >". Give port 80 or 81 etc. After this, it will show the running host IP and port, as presented in the picture.

Figure 19 Data center command line confirming port

16. Take the terminals of the tenant or hosts which are part of the same network slice as datacenter. For example, h11, h21, h31 are the hosts from access networks 1,2 and 3 and chatroom service to all these hosts are provided by d1.
17. After taking the xterm of desired tenant, enter to the folder/directory where the client.py file is saved.
18. Run the "client.py" file at the host h11 using "sudo python3 client.py". Similar, take xterms of h21 and h31 and run the "client.py" on them too.
19. After running the file, it will ask to "enter host name" and "enter port". Give the IP address of d1 (172.10.0.1) in the host name and enter the same port which was provided previously to the d1 (80 in the above scenario). Then, "enter username" of own choice.
20. After the step 18, a message will appear on the data center d1 "new connection. Username:xyz". Now, the data center d1 serves as the chatroom and the tenants of slice 1 h11, h21 and h31 will be able to communicate with each other in this chatroom, as depicted in the figure.

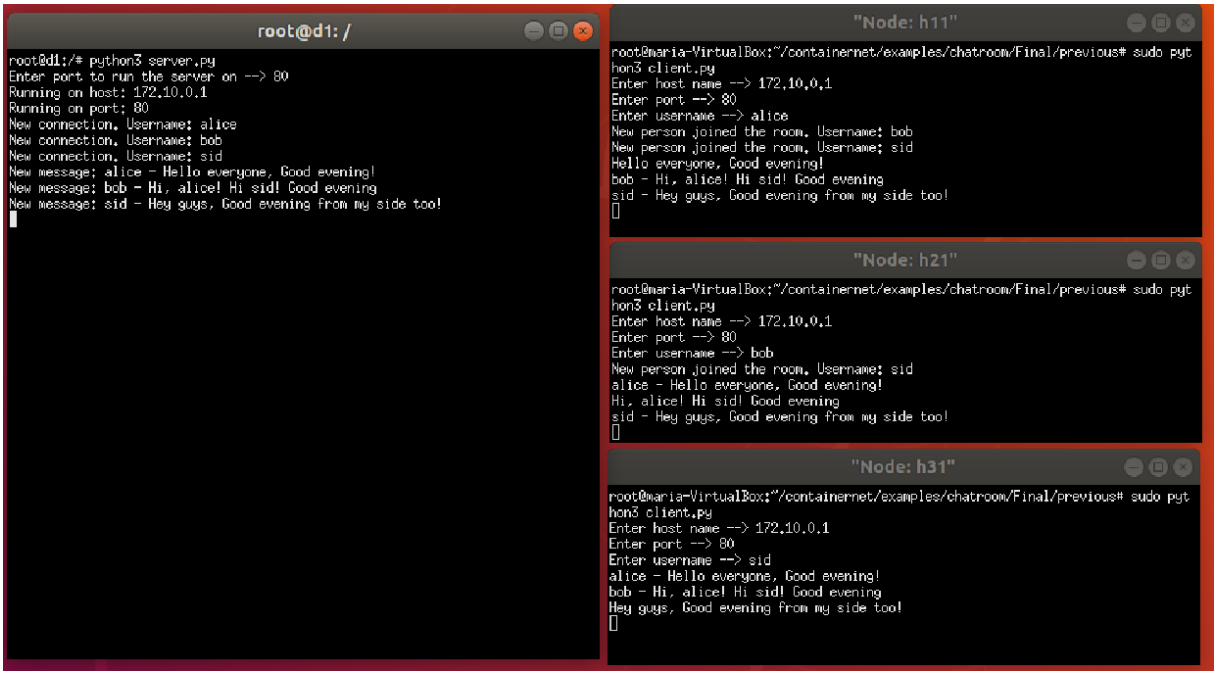

Figure 20 Running Chat room service with client windows and chat room.

21. Other slices will work on the similar pattern.
22. The NSH based data traffic can be analyzed by starting the wireshark capturing process at any interface of transport network OvSs (st1,st4,st7)/ or sd1-eth2, eth3 or eth4. Using the filter “nsh” on wireshark pcap file. On interface sd1-eth1, the packets will be decapsulated so, it is not recommended to examine the NSH traffic on that interface.

11. Results Analysis

Following is the functional testing and concerned Wireshark Captures of all slices.

11.1 Functional Testing of Network Slice 1:

As mentioned in above sections, the whole NSH traffic within network slice 1 is identifiable by NSH fields, NSH-SPI=1234 & NSH-C1=11223344.

Slice 1, NSH_SPI=1234 NSH_C1=11223344	
Clients Addressed	**h11, h21, h31**
Chatroom Server	**d1**
Edge server for DHCP service	**e1 for h11, e2 for h21, e3 for h31**

11.1.1 IP Allocation of slice 1 clients by DHCP Service

- **Client h11:**

The following captures shows the DHCP server **e1** allotting dynamic IP address to client **h11** of access network 1. This traffic is captured at edge switch **sr1-eth1**.

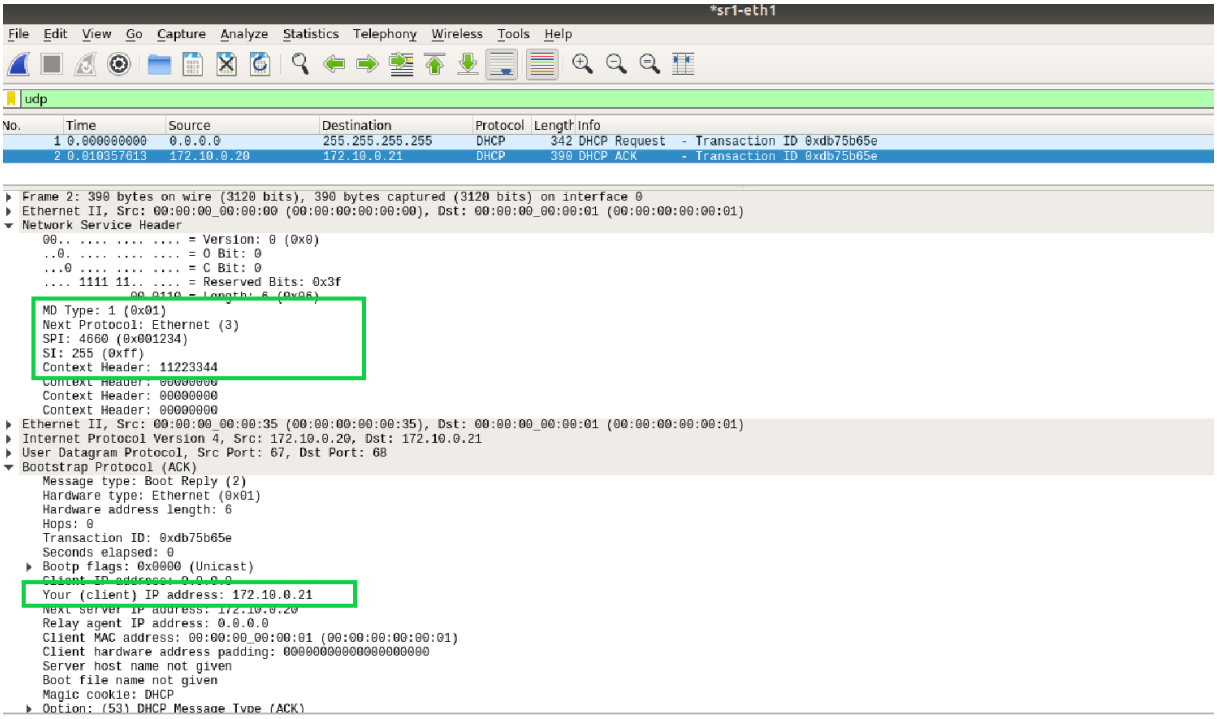

Figure 21 e1 allotting IP to h11

- **Client h21:**

The following captures shows the DHCP server **e2** allotting dynamic IP address to client **h21** of access network 2. This traffic is captured at edge switch **sr2-eth1**.

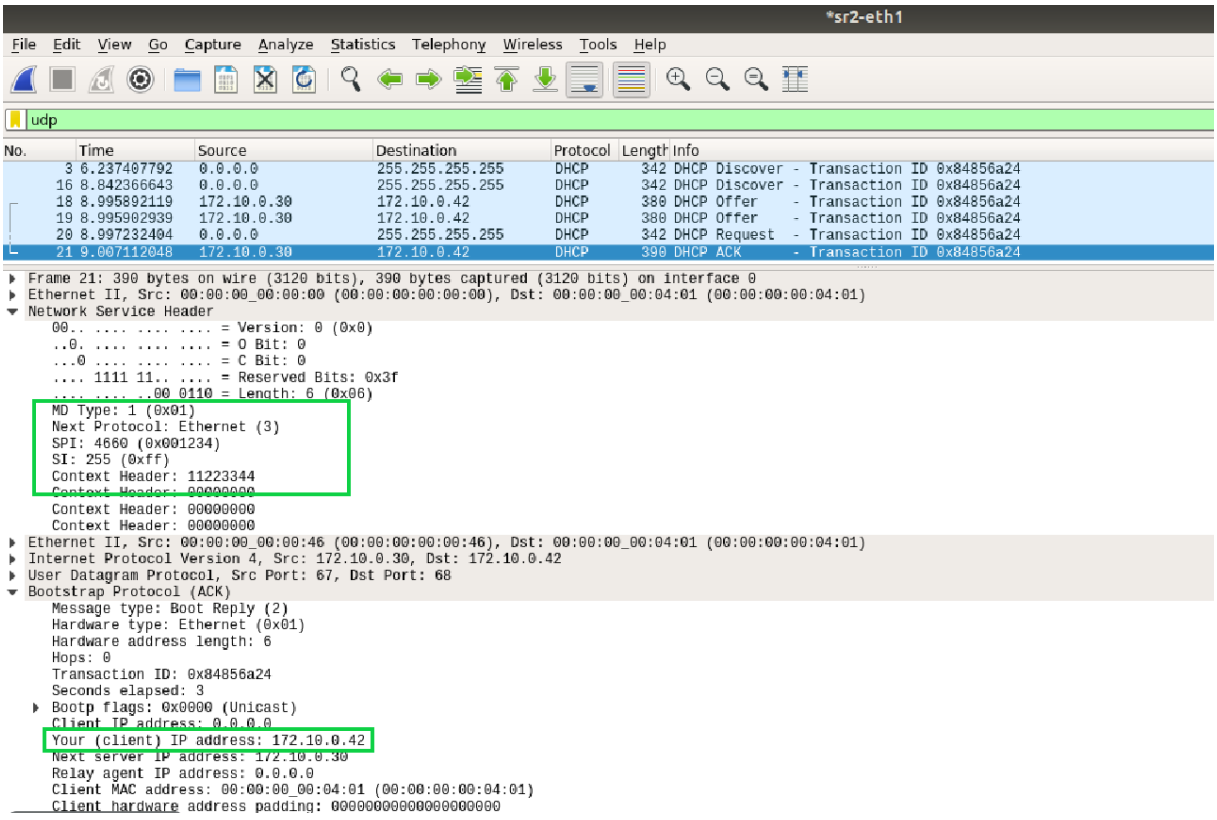

Figure 22 e2 allotting IP to h21

- **Client h31:**

The following captures shows the DHCP server **e3** allotting dynamic IP address to client **h31** of access network 3. This traffic is captured at edge switch **sr3-eth1**.

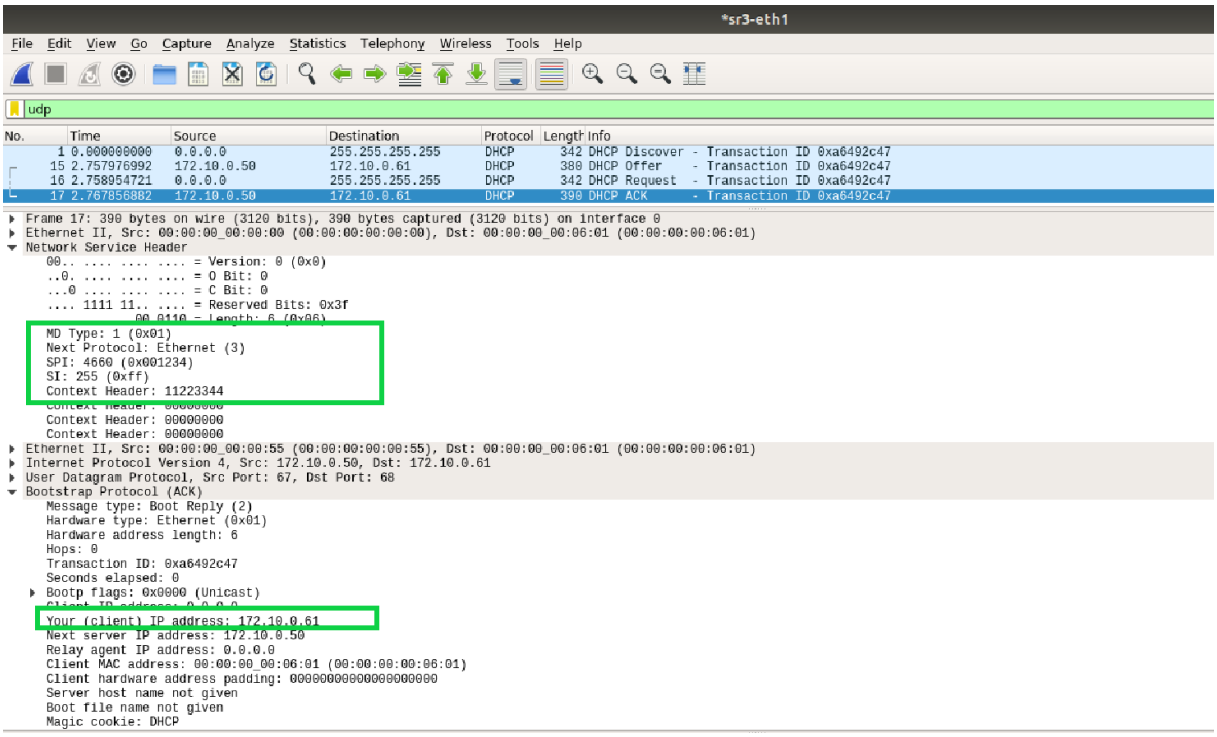

Figure 23 e3 allotting IP to h31

41.2.2 Deployment of Chatroom Service of slice 1 clients

In following figure, it is shown that the assigned chatroom server for network slice 1 at data center **d1,** is providing the chatroom service to the users of this slice. The tenants from different access network are part of this slice and communicating with each other on d1.

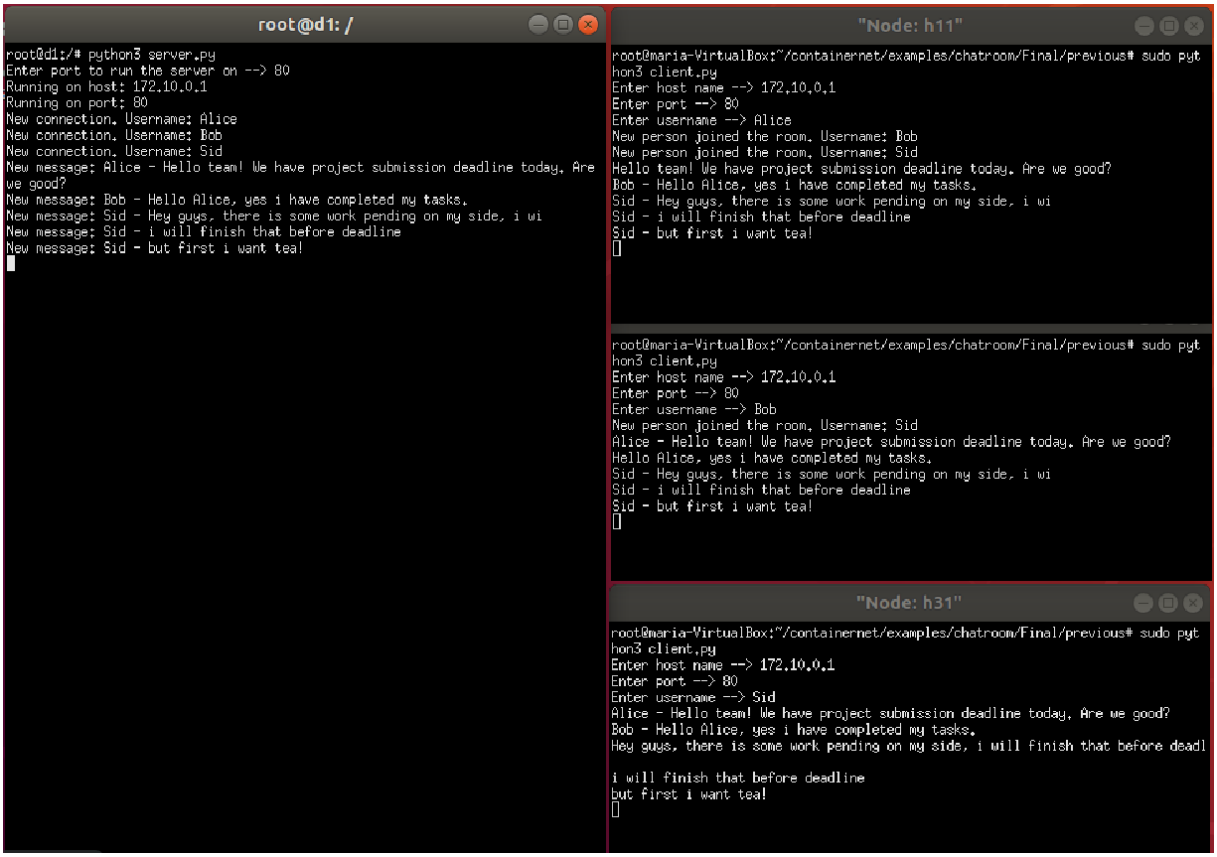

Figure 24 Slice 1 clients using chatroom service.

The following wireshark captures proves that the NSH data traffic is restricted within the network slice 1, which can be perceived by the SPI and C1 identifiers. The NSH traffic showed in the figures are captured at **sd1-eth4**. It can also be captured at **sd1-eth2** & **sd1 eth-3.**

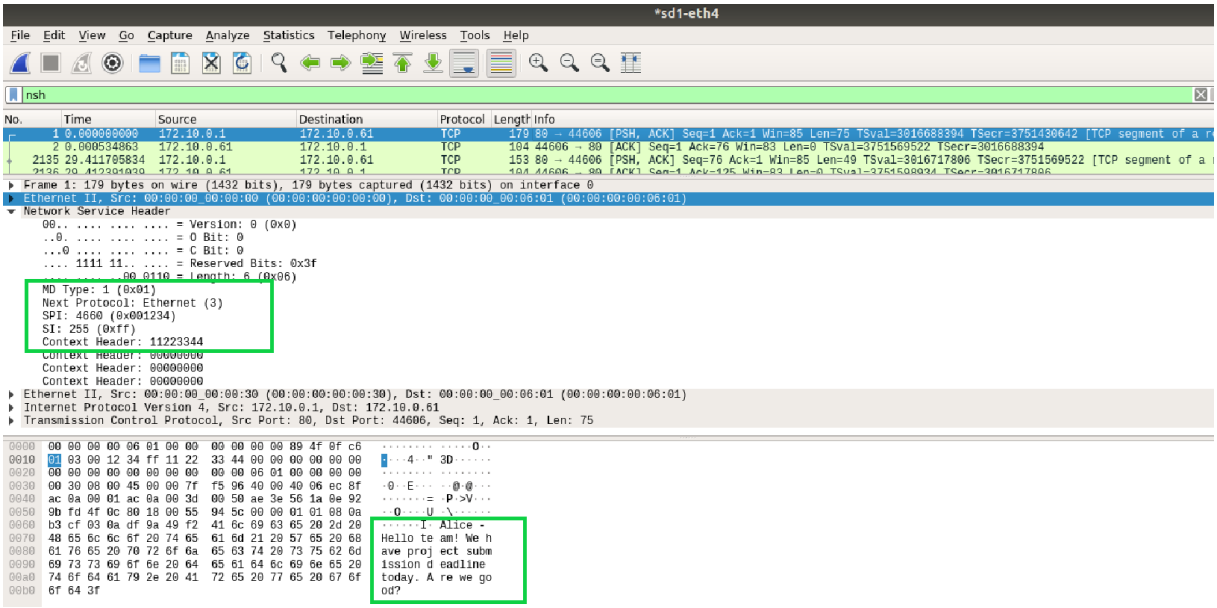

Figure 25 Slice 1 Chat message 1

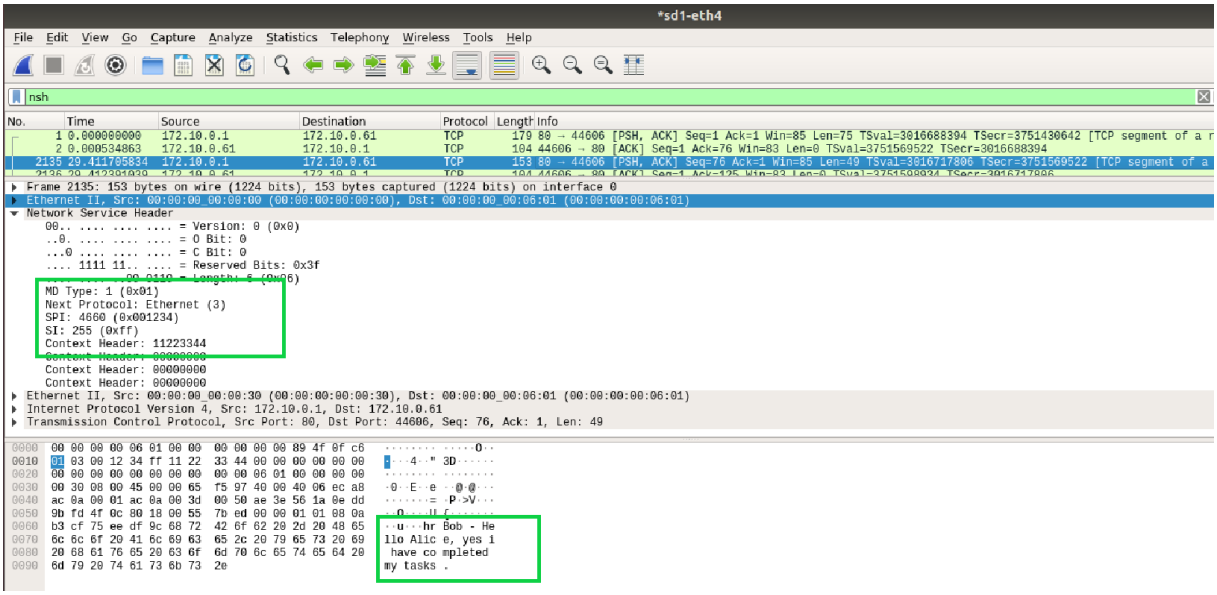

Figure 26 Slice 1 Chat message 2

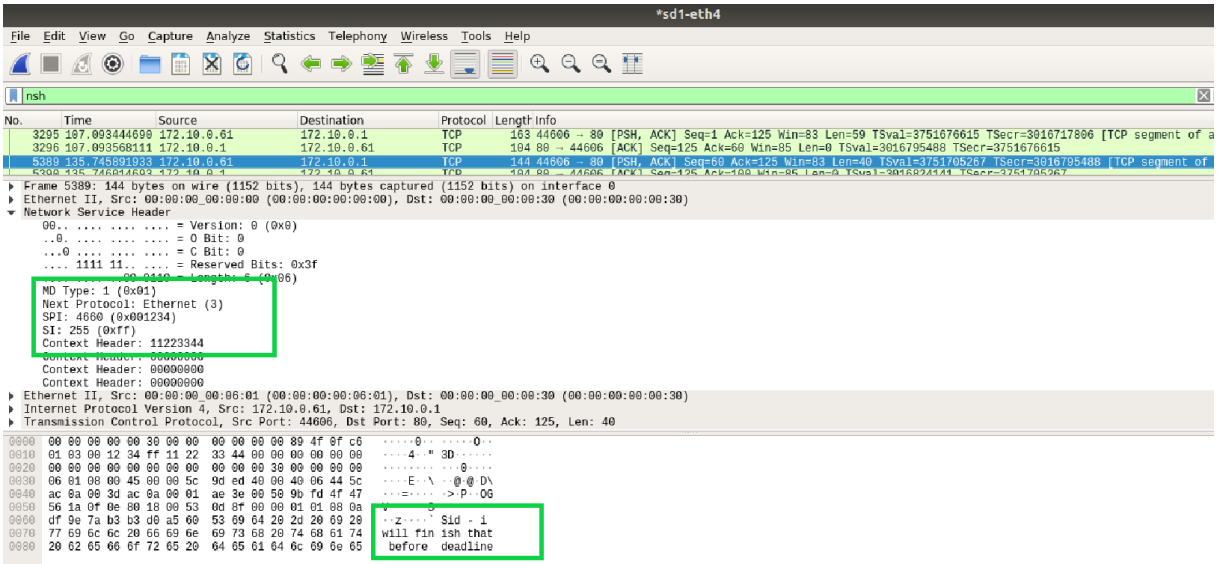

Figure 27 Slice 1 Chat message 3

11.2 Functional Testing of Network Slice 2:

Similar to network slice 1, NSH traffic within network slice 2 is identifiable by NSH fields, NSH-SPI=5678 & NSH-C1=55667788.

Slice 2, NSH_SPI=5678 NSH_C1=55667788	
Clients Addressed	**h12, h32**
Chatroom Server	**d2**
Edge server for DHCP service	**e1 for h12, e3 for h32**

11.2.1 IP Allocation of slice 2 clients by DHCP Service

- **Client h12:**

The following captures shows the DHCP server **e1** allotting dynamic IP address to client **h12** of access network 1. This traffic is captured at edge switch **sr1-eth1**.

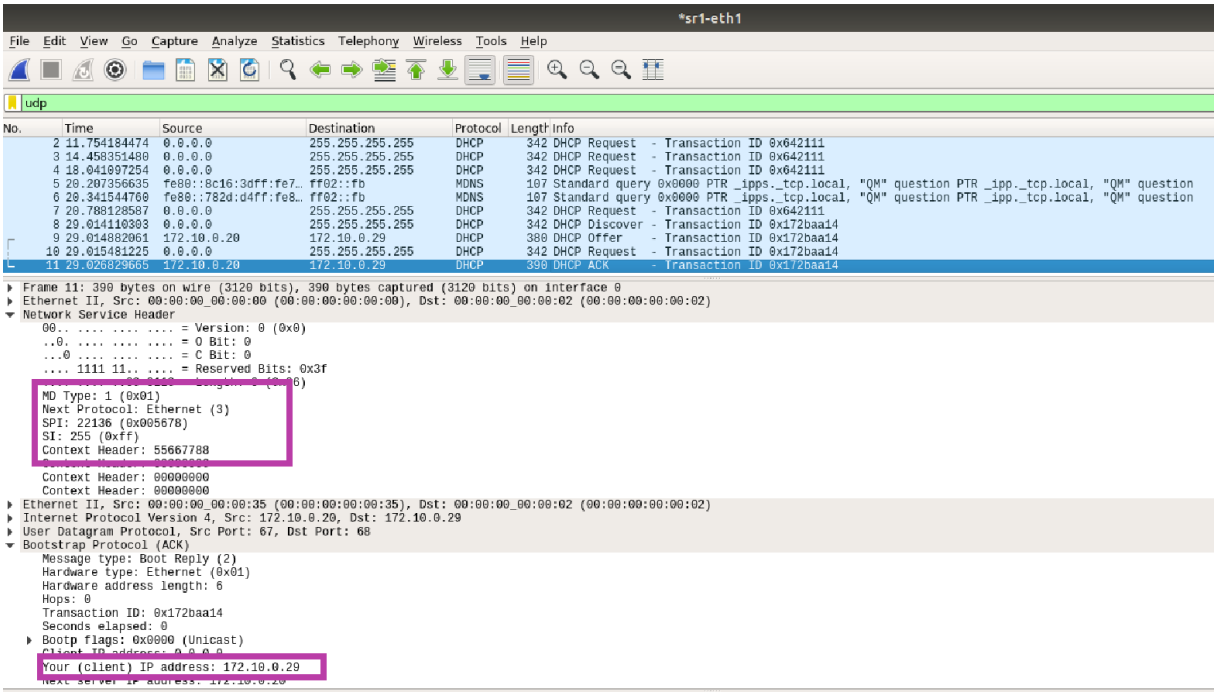

Figure 28 e1 allotting IP to h21

- **Client h32:**

The following captures shows the DHCP server **e3** allotting dynamic IP address to client **h32** of access network 3. This traffic is captured at edge switch **sr3-eth1**.

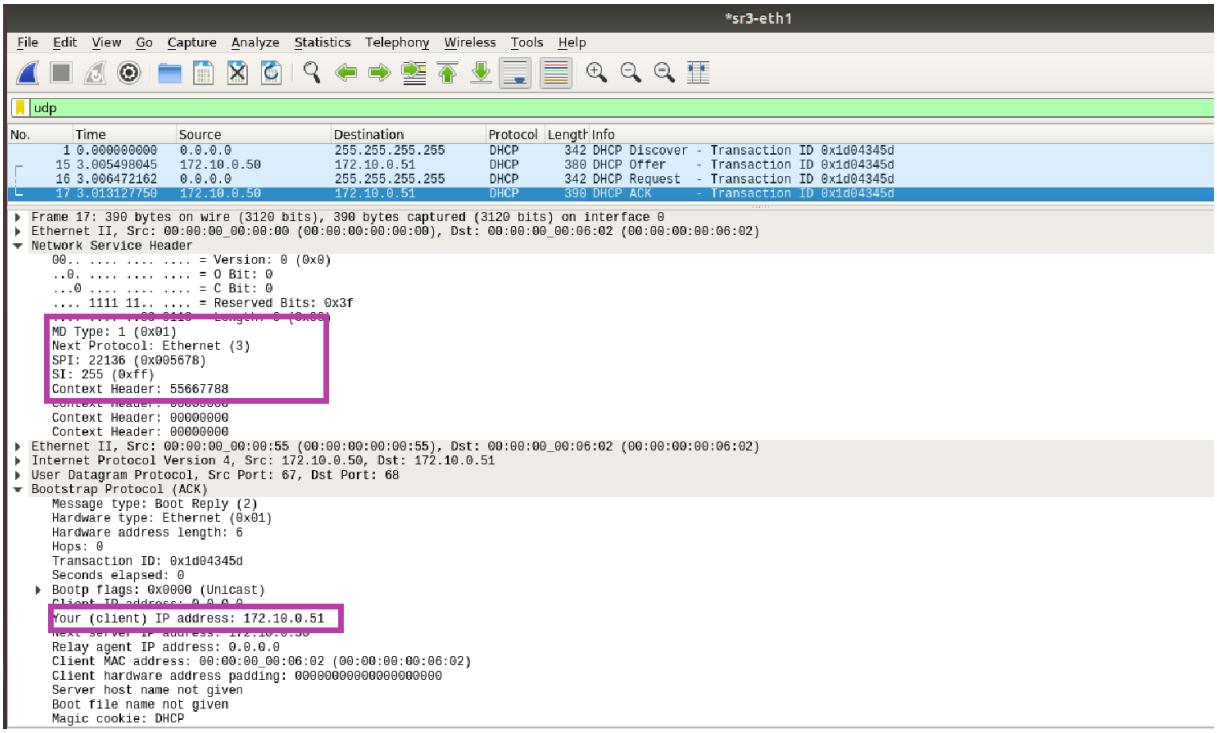

Figure 29 e3 allotting IP to h32

41.2.2 Deployment of Chatroom Service of slice 2 clients

In following figure, it is shown that the assigned chatroom server for network slice 2 at data center **d2,** is providing the chatroom service to the users of this slice. The tenants from different access network are part of this slice and communicating with each other on **d2**.

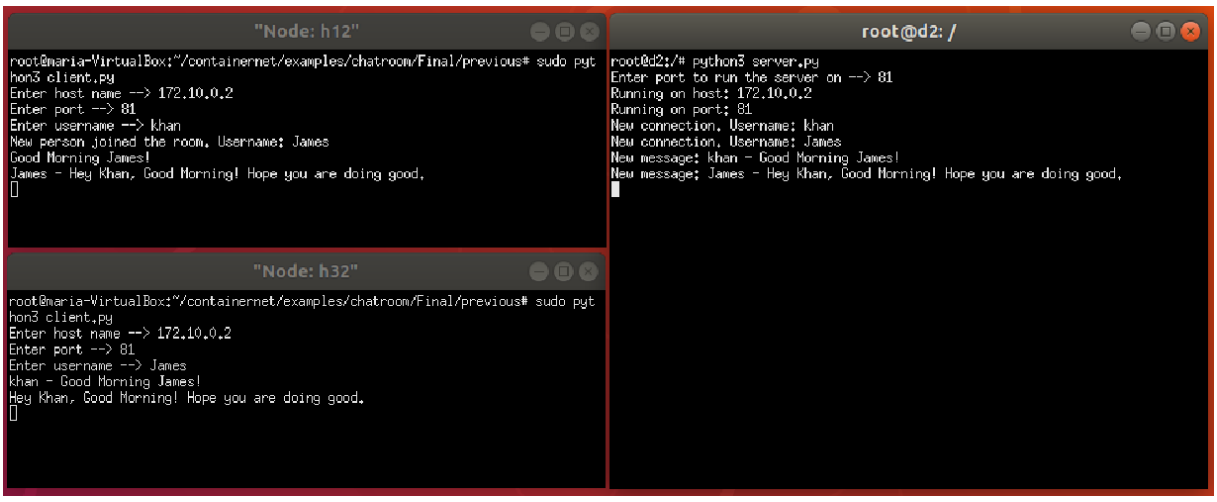

Figure 30 Slice 2 clients using chatroom service.

The following wireshark captures proves that the NSH data traffic is restricted within the network slice 1, which can be perceived by the SPI and C1 identifiers. The NSH traffic showed in the figures are captured at **sd2-eth2**. It can also be captured at **sd2 eth-3.**

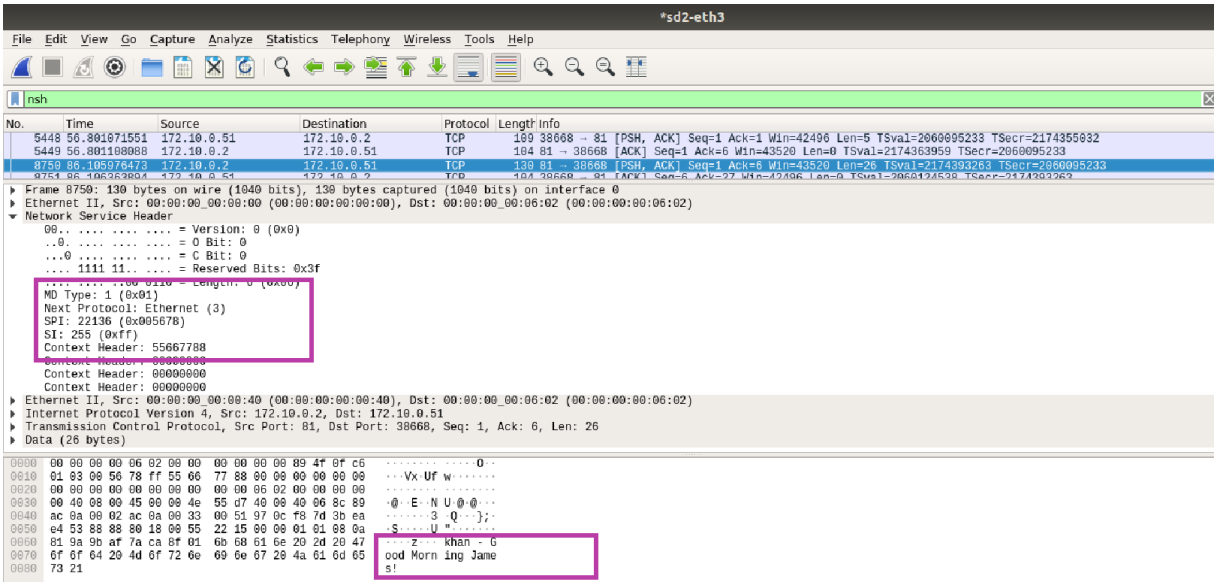

Figure 31 Slice 2 Chat message 1

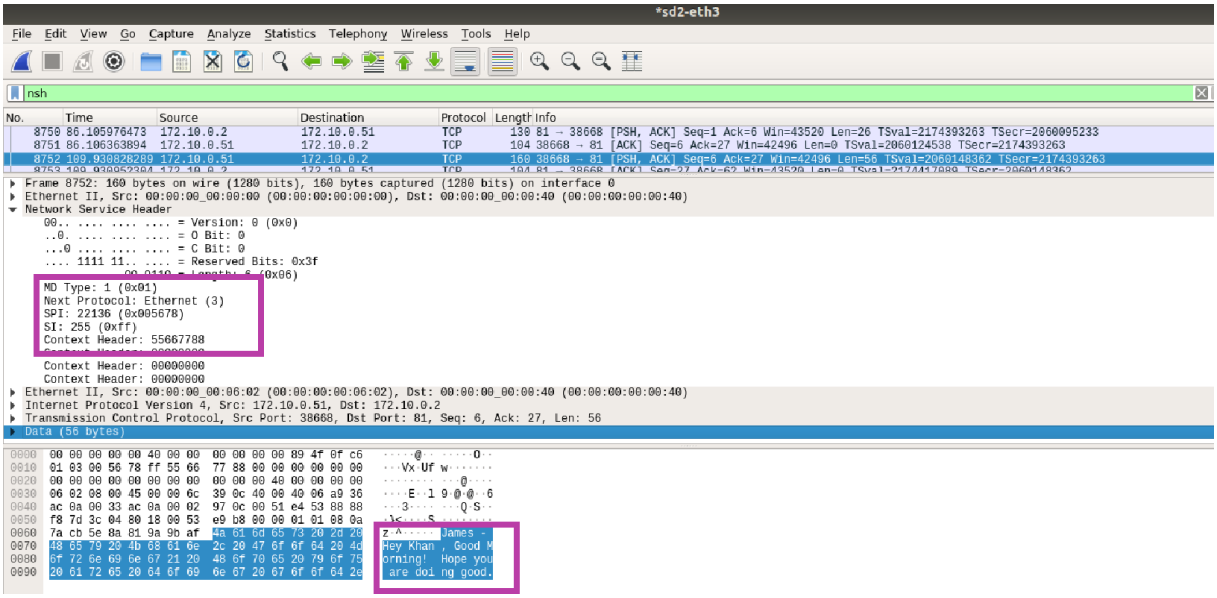

Figure 32 Slice 2 Chat message 2

11.3Functional Testing of Network Slice 3:

The NSH traffic within network slice 3 is identifiable by NSH fields, NSH-SPI=4321 & NSH-C1=44332211.

Slice 3, NSH_SPI=4321 NSH_C1=44332211	
Clients Addressed	**h22, h33**
Chatroom Server	**d3**
Edge server for DHCP service	**e2 for h22, e3 for h33**

11.3.1 IP Allocation of slice 3 clients by DHCP Service

- **Client h22:**

The following captures shows the DHCP server **e2** allotting dynamic IP address to client **h22** of access network 1. This traffic is captured at edge switch **sr2-eth1**.

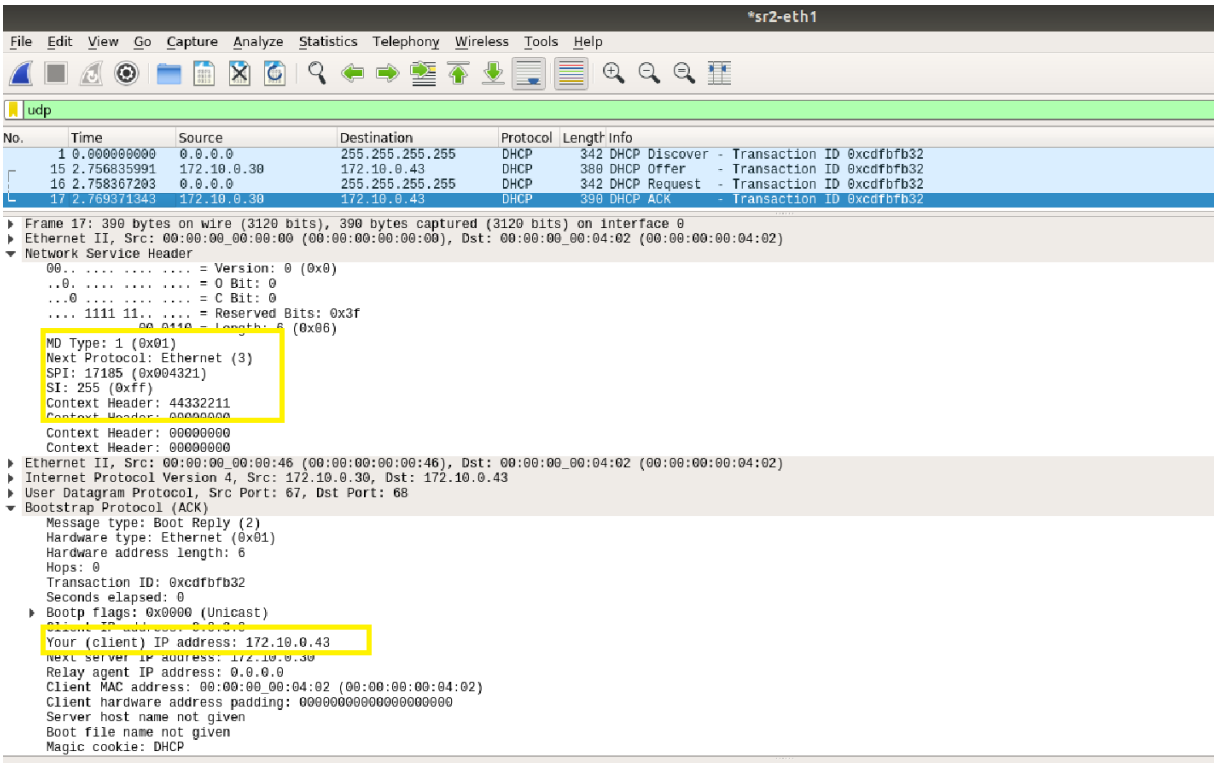

Figure 33 e2 allotting IP to h22

- **Client h33:**

The following captures shows the DHCP server **e3** allotting dynamic IP address to client **h33** of access network 1. This traffic is captured at edge switch **sr3-eth1**.

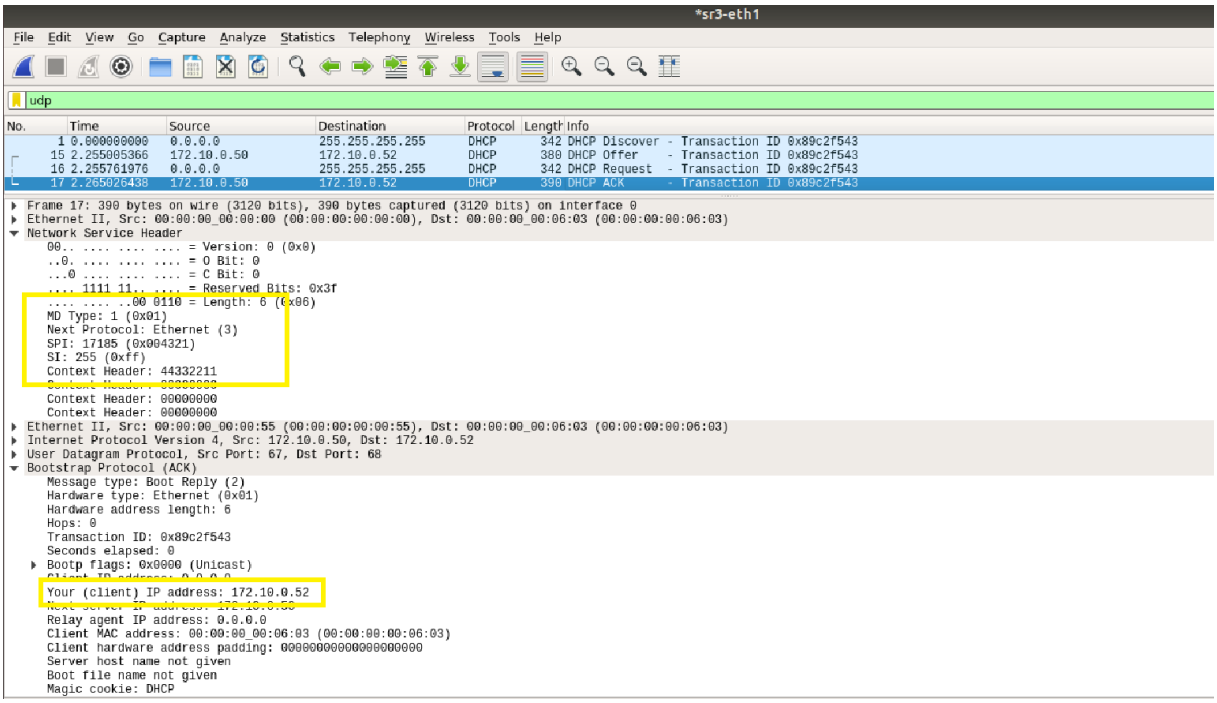

Figure 34 e3 allotting IP to h33.

In figure below, it is shown that the assigned chatroom server for network slice 3 at data center “d3”, is providing the chatroom service to the users of this slice. The tenants from access network 2 & 3 are part of this slice and communicating with each other on d3.

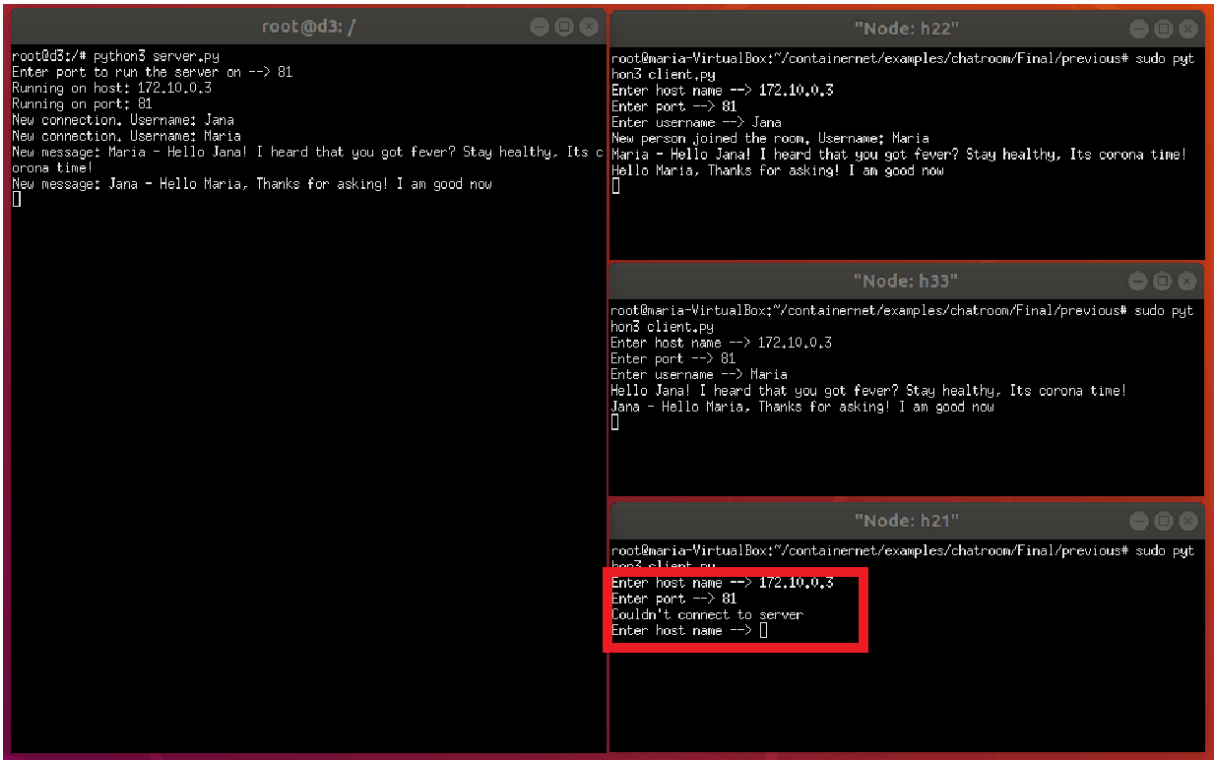

Figure 35 Slice 3 clients using chatroom service.

Also, it can be seen in the figure above that tenant h21 of access network 2, which is not the part of network slice 3, is trying to join the chatroom at d3. Since, Chatroom at d3 is only permitted to serve tenants h22 and h33, therefore h21 is getting the error “Couldn’t connect to server”.

The following wireshark captures, proves that the NSH data traffic is restricted within the network slice 3, which can be perceived by the SPI and C1 identifiers. The NSH traffic showed in the figures are captured at **sd3-eth3**. It can also be captured at **sd3-eth2**.

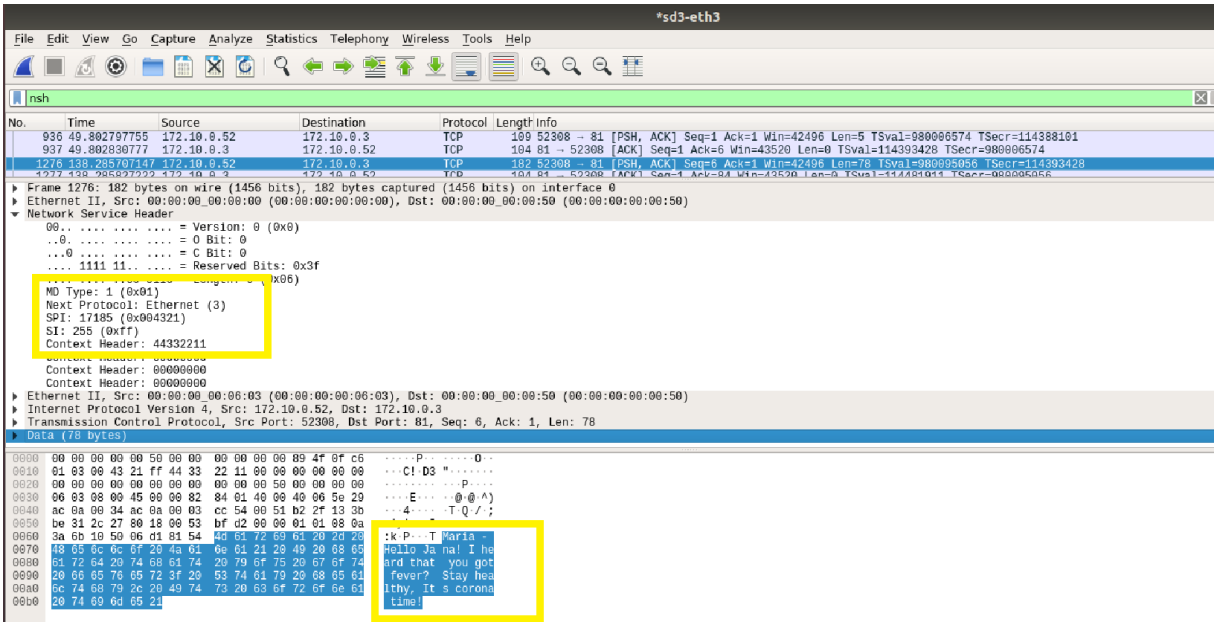

Figure 36 Slice 3 Chat message 1

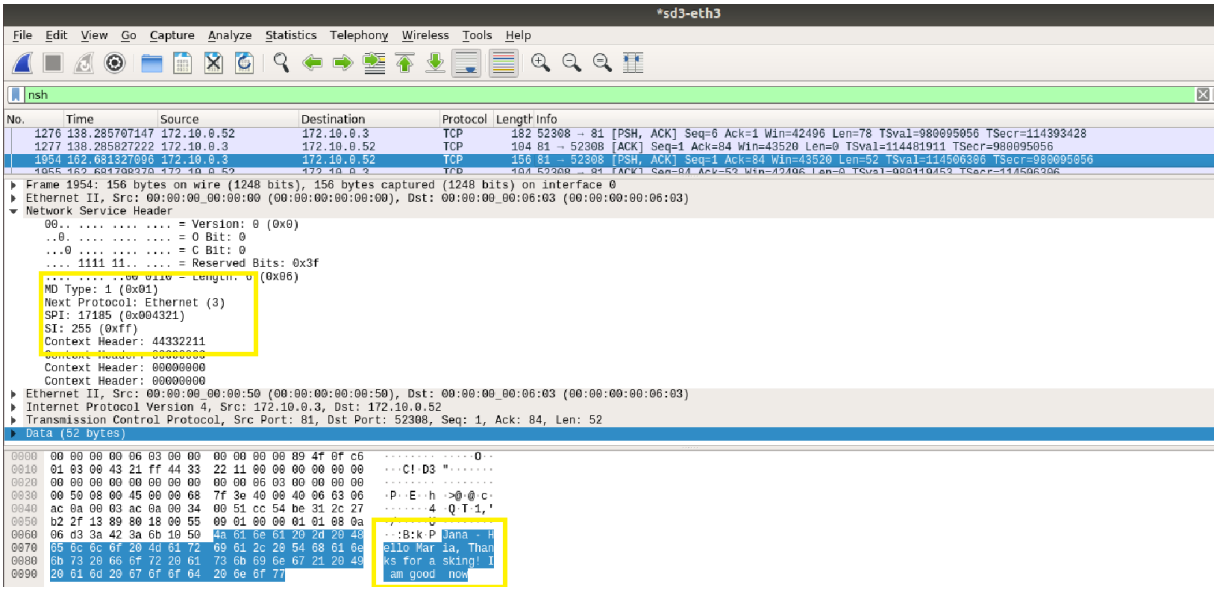

Figure 37 Slice 3 Chat message 2

41.2 Additional Testing

Two chatrooms of different slices can run on same port simultaneously. For example, the following figure, shows that d1 and d2 are running on same port “80” at the same time.

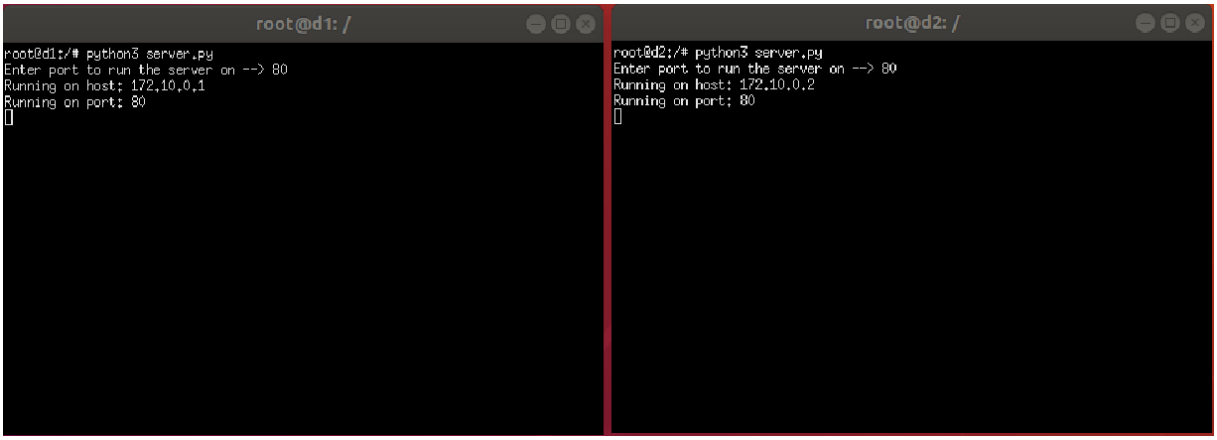

Figure 38 Two server service at same port

12. Open Issues

Following are the tests and trials after which this final network was achieved. These are the detailed explanations of all the issues that were faced in the implementation of the project.

Open Issues:

1. Remote SDN Controller Integration
2. NSH Encapsulation over IP based transport header
3. Unable to fulfil the Redundancy requirement
4. Managing ARP-Packets with NSH encapsulation
5. No NSH encapsulation on DHCP discover & request packets
6. Chatroom Service Implementation

12.1 Problem statement 1: Integration of Remote SDN Controller

Aim:

When we began with our initial research on the assigned project and started reading about Network Service Header Protocol Implementation on Mininet/Containernet environment, the internet search always directed us towards the utilization of Remote SDN controller with Mininet. A few research papers and NSH introductory videos were available on internet, which leads to integration of Remote SDN controller. Since, it was the stage where we were exploring different approaches, we found the usage of remote SDN controller in our network emulation quite fascinating and practical, therefore we tried to integrate that within our network emulation environment.

Evaluated Approaches & related Issues:

To achieve the Remote controller Integration with our Containernet emulation, we explored two different SDN Controllers;

a. POX Controller:

It's a python based open-source controller, which provides the possibility to create a controller of our own choice by using and manipulating the tutorial python codes already available in pox library. This controller would be used as SDN-controller in our network topology to control, manage and steer data packets across the network. There are various github repositories available, which provides a lot of possibilities to use this approach. We started following one of the repositories, tried to understand related python-based codes, and implement it to achieve the NSH approach. However, it required python expertise to create a completely functioned Controller, so we could not succeed to utilize the POX controller for NSH implementation.

b. Open Day Light (ODL) Controller:

As the idea of POX controller did not work for us, we moved to Open Day light controller. The main reason of choosing ODL controller was the availability of features to implement Service Function Chaining (SFC) and NSH. In our research phase, we explored different versions of ODL and based on the available features, we chose to work with ODL oxygen release. Our approach was to use ODL controller's SFC features to implement NSH based network slices. With this approach, we could achieve two possible project extensions i.e. Remote SDN controller integration and Service Function Chaining implementation. We created our Containernet emulation on one VM and configured the ODL controller on another VM ubuntu-20.10 live server. We were successful in connecting both VMs together and control the packet flow using ODL controller. After going through Opendaylight documentation, we figured out that installation of SFC features plays a major role in implementation of NSH based network slicing. Therefore, after basic features installation, we tried to install the following related SFC-features in our OpenDayLight.

Odl-sfc-model
odl-sfc-provider
odl-sfc-provider-rest
odl-sfc-netconf
odl-sfc-ovs
odl-sfc-scf-openflow
odl-sfcofl2
odl-sfclisp
odl-sfc-sb-rest
odl-sfc-ui

This always leads us to a huge list of error;

"Unsatisfiedrequirements:osgi.wiring.package;filter:="(&(osgi.wiring.package=org.opendaylight.openflowplugin.api)(&(version>=0.6.0)(!(version>=1.0.0)))) "Bundle was not resolved because of a uses constraint violation…"

We tried to troubleshoot the errors to make it work, but this issue didn't resolve. Afterwards, we found another SFC-ODL oxygen setup which is based on the ODL Gerrit project, so we tried to install it on a new VM by following the steps given in [12]. This approach also did not work as we always encountered the compilation error, as shown in fig.

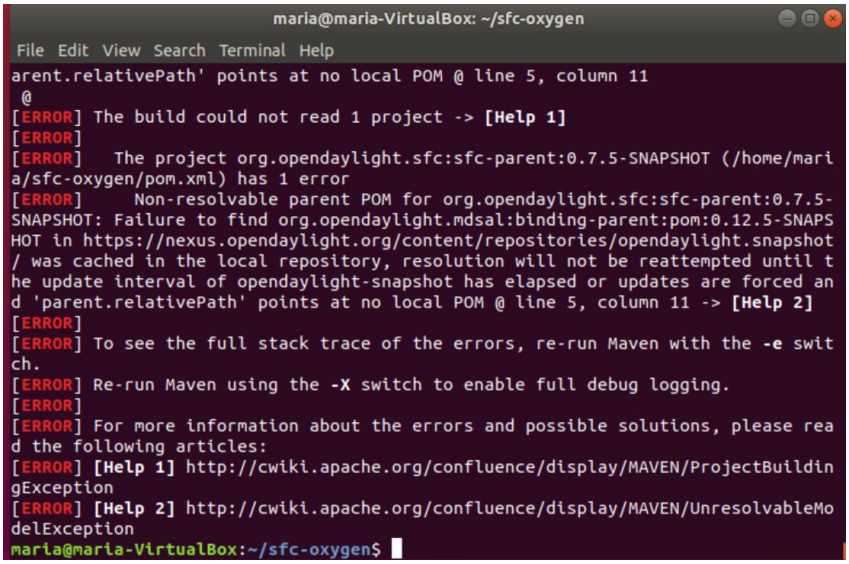

Figure 39 Errors in implementing SFC ODL Oxygen Setup

Result/Conclusion:

We tried to troubleshoot the indicated errors with the solution provided in the same link and recompiled it, but even that didn't resolve the issue. Therefore, we stopped working on this approach and started our research on implementation of network slicing with the configuration of flow rules on open virtual Switches OvS.

12.2 Problem statement 2: NSH Encapsulation over IP based transport header

Aim:

Our first Containernet emulation network for this project had routers in the transport network. As we had put so much effort in creating that topology and implementing OSPF based routing in it, we wanted to utilize it throughout our project. Therefore, our goal was to encapsulate NSH packets over IP based outer transport header, so that routers in the transport network can handle the NSH encapsulated packets.

Evaluated Approaches & related Issues:

Upon research and discussion with the Mobile Computing project panel, we found out that transport routers can only be used in our project topology, if we manage to implement the NSH over IP approach. To achieve this, we have two options;

a) **VXLAN-GPE tunnel;** By evaluating the provided approach in [13], we tried to setup vxlan-gpe tunnel in our topology by using following command,

ovs-vsctl add-port s1 s1-vxlangpe1 – set interface s1-vxlangpe1 type=vxlan options:exts=gpe options:remote_ip=192.168.1.1 options:packet_type=ptap ofport_request=30

It didn't work and when we investigated the root cause, we couldn't find type=vxlan & ext=gpe, which can be seen in the fig.

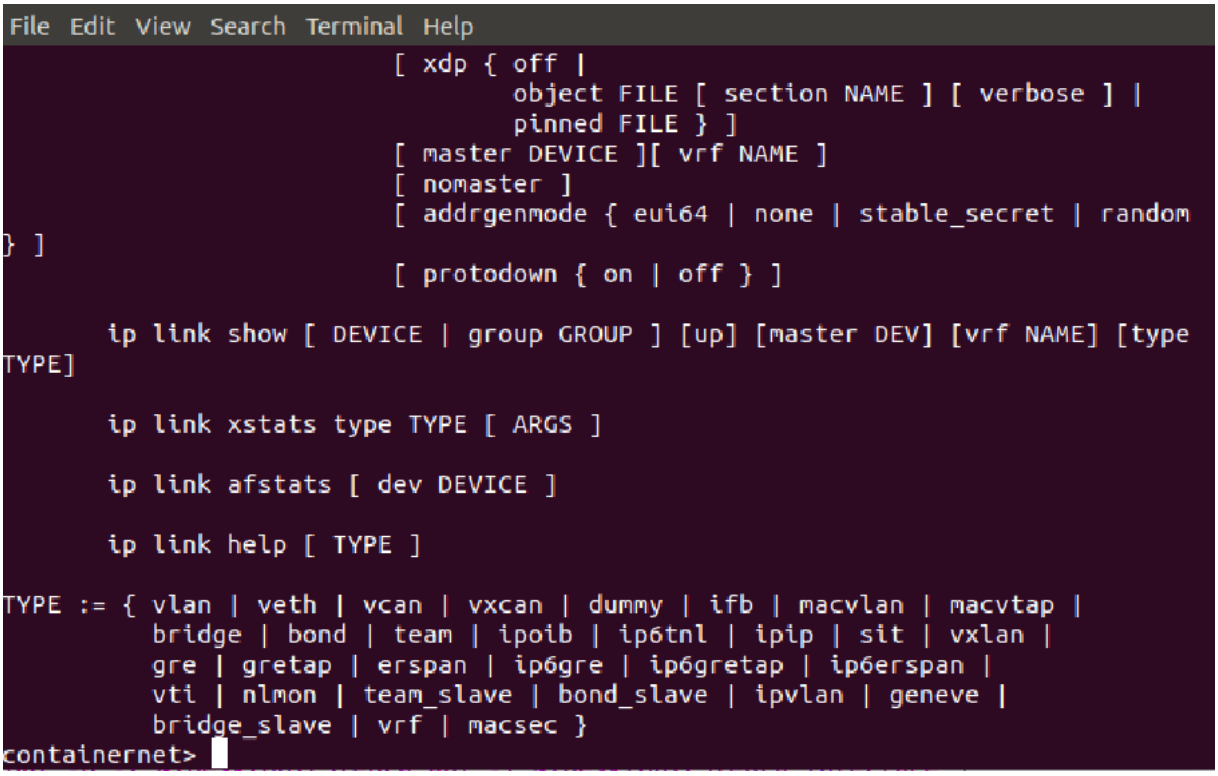

Figure 40 type :vxlan and ext: gpe not found

b) **Encapsulate NSH in IP header;** To encapsulate the NSH packets in the outer transport IP header, we used same NSH+EthernetII encapsulation flow rule which we are using currently in our project, and replaced ethernet to IP. For Example;

```
actions=encap(nsh(md_type=1)),set_field:0x1234-
>nsh_spi,set_field:0x11223344-
>nsh_c1,encap(IP),set_field:172.10.0.1->nw_dst,2
```

When we added this type of flow, we got the error, which can be seen in the fig.

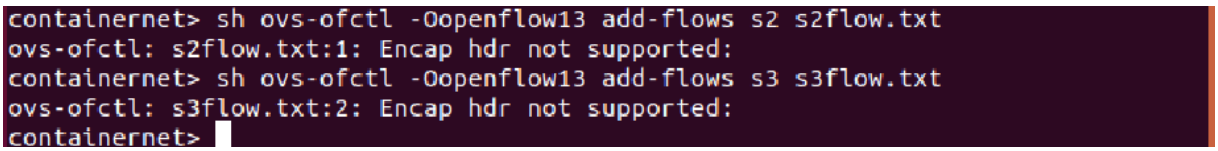

Figure 41 NSH encapsulating over IP error.

Results/Conclusion:

Although the similar issue was highlighted in thread [14] but we still tried to find any suitable approach for this issue. After trying various other ways to transport NSH over IP, we dropped this idea and replaced all of the routers in our transport network to OvS switches making the whole emulation network "NSH-Aware". Although, in our opinion this approach is not practical enough, but this was the only option left to us. All OVS based network proposes only one IP network throughout the network infrastructure, which initiated problem#3.

12.3 Problem Statement 3: Unable to fulfil the Redundancy requirement.

Aim:

In our project description document, on of the requirements stated that "The configuration of the network slices within the transport network should not rely on the same path through the transport network".

Evaluated Approaches & related Issues:

With the above stated requirement, we perceived that we need to add redundant paths in our topology. For testing purpose, we added only one redundant path and assigned the flows rules in one path the “high priority” and the other path “low priority”. This approach failed, as we received same packet twice at the destination. Therefore, we assigned the same path to all the slices till the first transport switch which is connected directly to the edge OVS. Then this switch directs the packets of different slices to their specified paths.

12.4 Problem Statement 4: Managing ARP-Packets with NSH encapsulation

Aim:

Defining the flow rules for ARP-Packets was the toughest part of this project. Encapsulating and forwarding the ARP packets with NSH/IP header would be achievable if we were using the IP based outer transport header, as we could define the ARP flows based on network source and destination. But in our case NSH over EthernetII, we are using Mac address of the destination node to define the flow rules on OvSs to steer the NSH traffic within the slice. As we know that, the ARP packets are broadcasted across whole network, hence the destination mac address in the ARP packet is “ff:ff:ff:ff:ff:ff”, consequently, we were unable to define proper flow rules on the OvSs to encapsulate and forward ARP packets.

Evaluated Approaches & related Issues:

a) We tried to use different ARP-fields to define the flow rules on OvSs for ARP packet encapsulation in NSH+EthernetII by manipulating fields like dl_src, dl_dst, arp_sha and arp_tha. But it caused following error when we run the configuration, which can be seen in the fig.

```
containernet> sh ovs-ofctl -Oopenflow13 add-flows s4 s4flow.txt
2021-03-02T15:05:41Z|00001|ofp_actions|WARN|set_field arp_sha lacks correct prerequisites
ovs-ofctl: s4flow.txt:1: actions are invalid with specified match (OFPBAC_MATCH_INCONSISTENT)
containernet> sh ovs-ofctl -Oopenflow13 add-flows s4 s4flow.txt
OFPT_ERROR (OF1.3) (xid=0xa): OFPBMC_BAD_PREREQ
OFPT_FLOW_MOD (OF1.3) (xid=0xa): ***decode error: OFPBMC_BAD_PREREQ***
00000000  04 0e 00 a0 00 00 00 0a-00 00 00 00 00 00 00 00 |................|
00000010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 80 00 |................|
00000020  ff ff ff ff ff ff ff ff-ff ff ff ff 00 00 00 00 |................|
00000030  00 01 00 33 80 00 00 04-00 00 00 02 80 00 0a 02 |...3............|
00000040  08 06 ff ff 04 05 00 5a-d6 50 01 ff ff 08 08 00 |.......Z.P......|
00000050  5a d6 50 00 00 12 34 ff-ff 0c 08 00 5a d6 50 11 |Z.P...4.....Z.P.|
00000060  22 33 44 00 00 00 00 00-00 04 00 38 00 00 00 00 |"3D........8....|
00000070  ff ff 00 10 00 00 23 20-00 2f 00 00 00 00 ff fe |......# ./......|
00000080  ff ff 00 10 00 00 23 20-00 2f 00 00 00 00 ff fe |......# ./......|
00000090  00 00 00 10 00 00 00 01-00 00 00 00 00 00 00 00 |................|
containernet> sh ovs-ofctl -Oopenflow13 add-flows s4 s4flow.txt
OFPT_ERROR (OF1.3) (xid=0xa): OFPBMC_BAD_PREREQ
OFPT_FLOW_MOD (OF1.3) (xid=0xa): ***decode error: OFPBMC_BAD_PREREQ***
00000000  04 0e 00 a0 00 00 00 0a-00 00 00 00 00 00 00 00 |................|
00000010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 80 00 |................|
00000020  ff ff ff ff ff ff ff ff-ff ff ff ff 00 00 00 00 |................|
00000030  00 01 00 33 80 00 00 04-00 00 00 02 80 00 0a 02 |...3............|
00000040  08 06 ff ff 04 05 00 5a-d6 50 01 ff ff 08 08 00 |.......Z.P......|
00000050  5a d6 50 00 00 12 34 ff-ff 0c 08 00 5a d6 50 11 |Z.P...4.....Z.P.|
00000060  22 33 44 00 00 00 00 00-00 04 00 38 00 00 00 00 |"3D........8....|
00000070  ff ff 00 10 00 00 23 20-00 2f 00 00 00 00 ff fe |......# ./......|
00000080  ff ff 00 10 00 00 23 20-00 2f 00 00 00 00 ff fe |......# ./......|
00000090  00 00 00 10 00 00 00 01-00 00 00 00 00 00 00 00 |................|
```

Figure 42 NSH encapsulation over ARP error.

b) After trying several approaches, we finally managed to encapsulate the ARP packets coming from one tenant to edge server by defining following rule on S1;

```
table=0,in_port=1,arp,dl_src=00:00:00:00:00:01,dl_dst=ff:ff:ff
:ff:ff:ff,actions=encap(nsh(md_type=1)),set_field:0x1234-
>nsh_spi,set_field:0x11223344-
>nsh_c1,encap(ethernet),set_field:ff:ff:ff:ff:ff:ff-
>dl_dst,set_field:00:00:00:00:00:01->dl_src,3
```

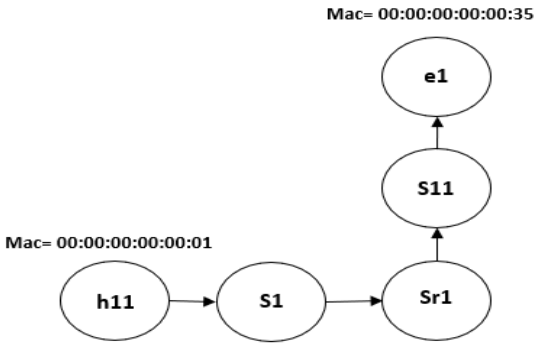

Figure 43 ARP packet simple forward by OvS

Sr1 simply forwards the ARP packets coming from S1-eth3 to Sr1-eth1 to the s11 on the basis of dl_src (tenant's mac address). The encapsulated ARP packets decapsulates at s11 and forwarded to e1. This approach was test firstly from h11 to e1 only, which resulted in successful ARP and IP packets encapsulation within this slice, as proved in the following Wireshark captures.

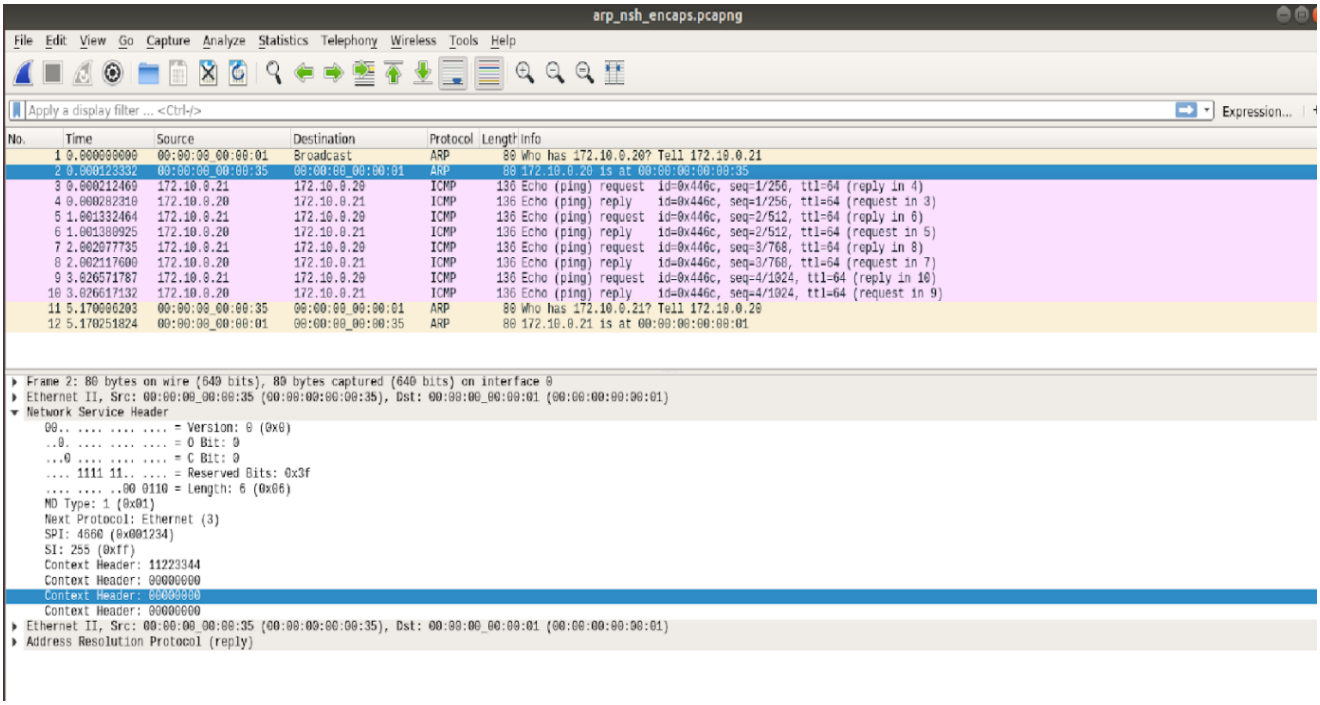

Figure 44 ARP Request packet Encapsulation

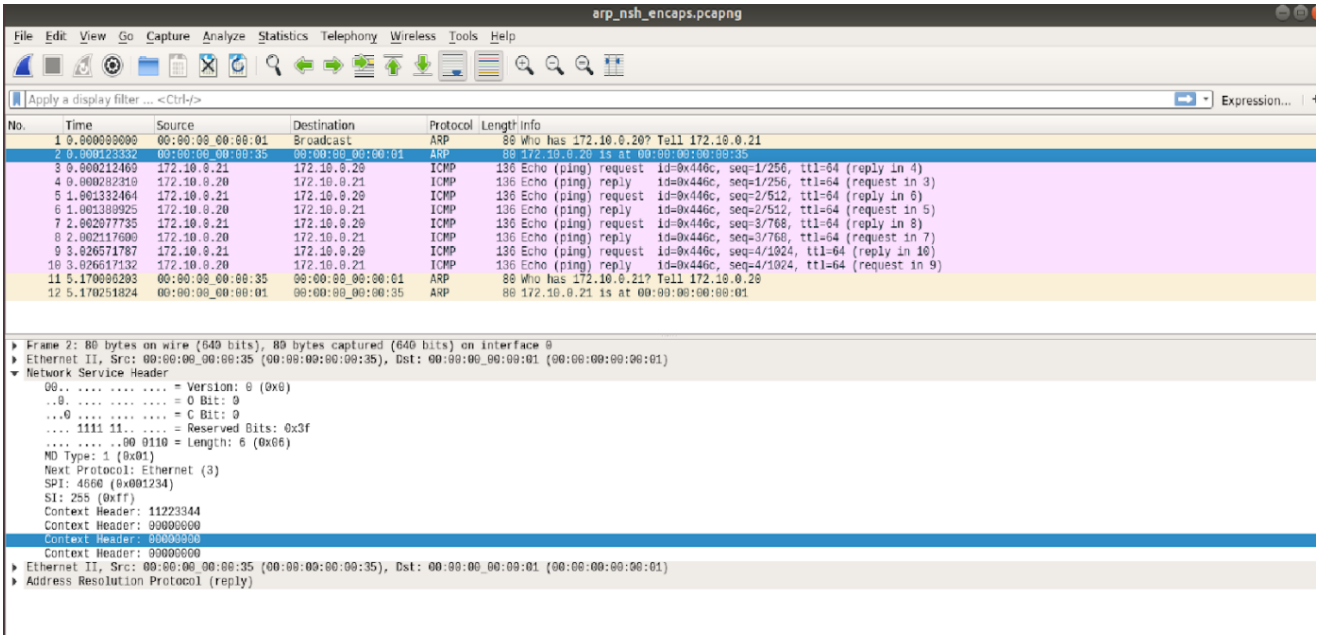

Figure 45 ARP Reply packet Encapsulation

When we extended this approach from h11 to datacenter by defining a flow on sr1, for example, when NSH+EthernetII encapsulated packet receives at sr1-eth1 with dl_src=00:00:00:00:00:01 and dl_src=ff:ff:ff:ff:ff:ff, then forward it towards both direction edge server e1 and datacenter d1. After defining this rule, we traced the packet direction coming from h11 and found that Sr1 is forwarding these packets only towards d1, no packet is forwarded towards e1. We verified this by starting wireshark packet capture process at sr1-eth2 in the direction of e1.

Hence, we were unable to integrate this approach in our complete network slice and submission deadline was near, thus we dropped this idea and decided to forward the ARP-Packets without NSH+EthernetII encapsulation by using the following rule.

```
table=0,in_port=1,arp,nw_src=172.10.0.0/24,nw_dst=172.10.0.0/24
,actions=local,flood,2

table=0,in_port=2,arp,nw_src=172.10.0.0/24,nw_dst=172.10.0.0/24
,actions=local,flood,1
```

c) These ARP flooding rules in all switches, initiated the looping process in our network topology which teared down our network, as its OVS based layer 2 network. However, we were able to break this loop by defining the following rule on 2 of the transport network OVS switches; st8 and st9.

```
table=0,in_port=1,arp,dl_dst=ff:ff:ff:ff:ff:ff,actions=drop
table=0,in_port=2,arp,dl_dst=ff:ff:ff:ff:ff:ff,actions=drop
```

12.5 Problem statement 5: No NSH encapsulation on DHCP Discover & Request packets

Aim:

The case of DHCP discover and request packets was same as the ARP packets, hence we were unable to encapsulate these packets in NSH.

Evaluated Approaches & related Issues:

The evaluated approaches are same as ARP packets. Here, we even could not define the flow rules based on network source and destination, as DHCP Discover and Request packets have the source network address 0.0.0.0 and destination IP address 255.255.255.255.
Therefore, only DHCP Offer and Ack packets are encapsulated in NSH+EthernetII. However, DHCP Discover and Request packets are broadcasted normally without NSH encapsulation towards Edge DHCP server.

12.6 Problem statement 6: Chatroom Service Implementation

Aim:

We aimed to deploy a Chatroom Webserver on a docker container to provide Instant Message Conferencing service to the tenants within their network slices.

Evaluated Approaches & related Issues:

a) **SIP IMC Module:** As we have already implemented the SIP based chat service in our previous course, we tried to use our prior knowledge to deploy SIP based chatrooms within our network infrastructure. The IMC Module is not updated and missed a Kamailio license. The reason behind this is the updated version of IMS services present instead of the IMC that included only the chat. For these reasons IMC service could not be deployed in the project.

b) **RocketChat:** We tried to deploy Rocket.Chat by using Rocket.Chat's official docker image [15]. To run Rocket.Chat, additional instances of "MongoDB" and "Node.js" were required. As the linking functionality Docker has been deprecated and Containernet does not provide this functionality, we were unable to link both of these extra instances with the Rocket.Chat's container in our Containernet emulation environment.

c) **Lets-Chat:** In the next trial, we attempted to deploy LetsChat by using sdelements/lets-chat image [16]. Lets-Chat also utilize MongoDB instance. But we made it work by running Lets-chat container from Containernet topology and passing the URI of mongodb (Ip address of the running mongoDB container on linux host) as environment variable without linking both containers. With this approach, we were succeeded in the deployment of this Lets-Chat server on our data center d1 and it was running perfectly on http://localhost:8091.

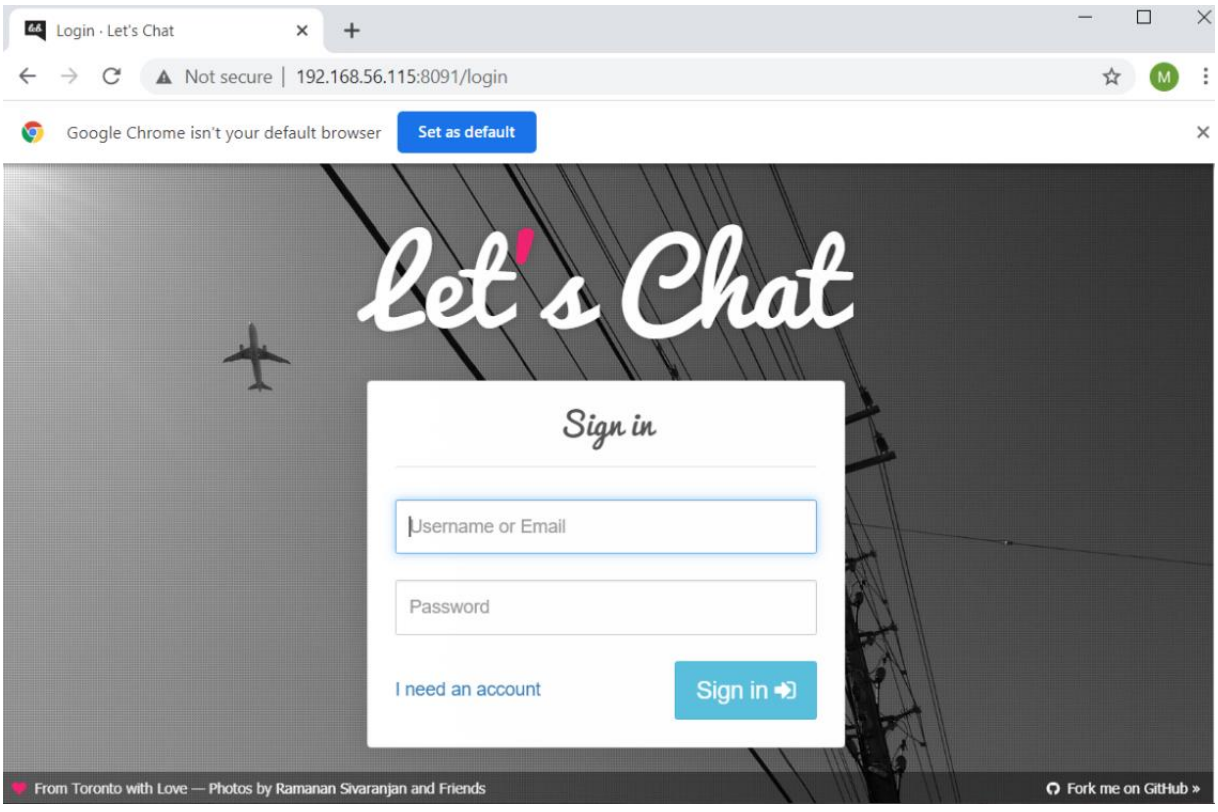

Figure 46 LetsChat running on Local Host 8091

But when we pinged d1 from the host, there was no connectivity. To check the issue, we went inside d1 and found out that the interface of d1 was down, as shown in the figure.

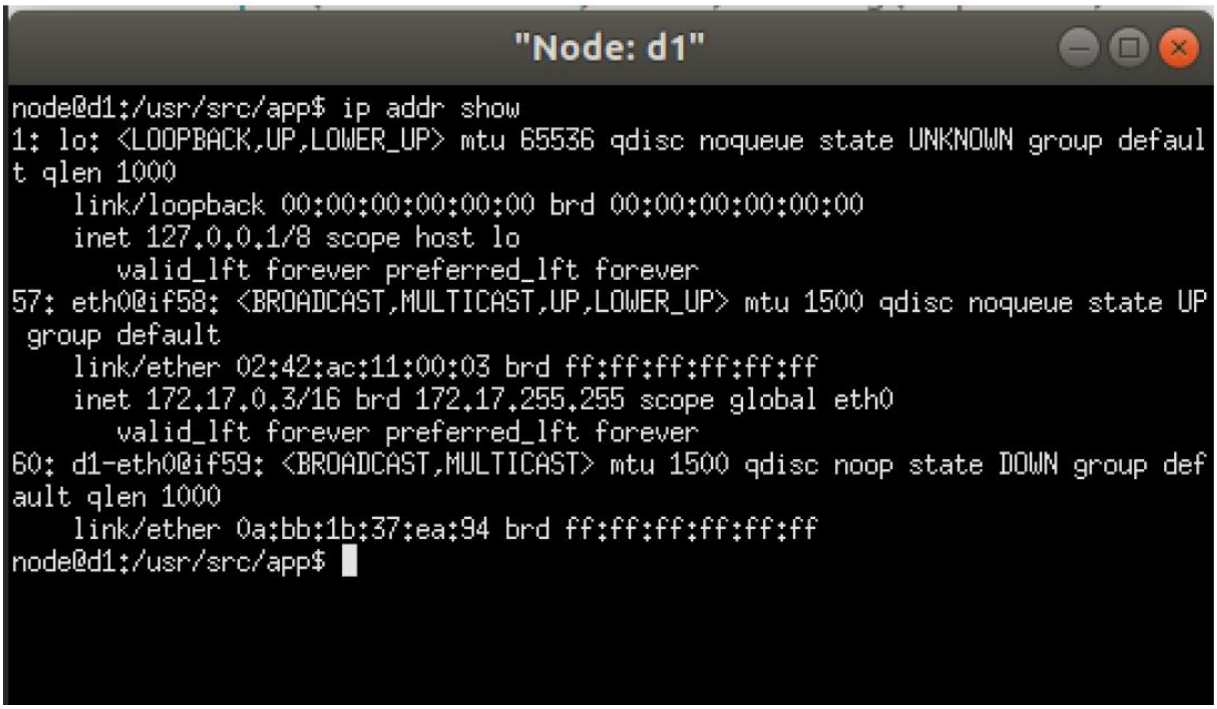

Figure 47 Letschat Container's Terminal

We tried to up the interface by using ip command but encountered the error "Operation not permitted". After applying different troubleshooting steps, we found out that the image of Lets-Chat does not match the requirements to be executed within Containernet because it is based on an old version of node.js, which is not supported by Containernet. Based on this reason, the Containernet is not capable to set the IP within the Container which caused the error "permission denied".

d) **IRC-Server:** We also deployed and explored the old-fashioned IRC chatroom server, but some of the commands were not functioning well, so we moved on from this too.

e) **Matrix Chatserver:** By using element-web docker images from [17]. We were able to run this chat server too, but this service deployment inside the containernet emulation was unachievable, as this container required to be run with "entrypoint.sh". We searched about this issue on internet to find any solution for this and figured out that Containernet does not support this too.

After all of these unsuccessful trials, we finally managed to find a simple python-based script to deploy on our datacenters which serves the Chatroom functionality as discussed in section 7.

13. Acknowledgement

We would like to express our gratitude towards Prof Ulrich Trick for giving us the required insight and knowledge to work on this project. With that we would especially extend our thanks to Mr. Gregor Frick for supporting us through the implementation and research of the problem statements at hand. The assistance provided by the Mobile Computing Project panel was unparalleled and we would like to show our deep appreciation to the whole committee for helping us in finalizing this project.

14. References

[1] X. Li et al., "Network Slicing for 5G: Challenges and Opportunities," in IEEE Internet Computing, vol. 21, no. 5, pp. 20-27, 2017, doi: 10.1109/MIC.2017.3481355.

[2] C. Bouras, A. Kollia and A. Papazois, "SDN & NFV in 5G: Advancements and challenges," 2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN), Paris, France, 2017, pp. 107-111, doi: 10.1109/ICIN.2017.7899398.

[3] Quinn, P., Ed., Elzur, U., Ed., and C. Pignataro, Ed., "Network Service Header (NSH)", RFC 8300, DOI 10.17487/RFC8300, January 2018, https://www.rfc-editor.org/info/rfc8300

[4] Pignataro, C., and Quinn, P., "Service Function Chaining", CISCO, USA, April 2017, https://www.cisco.com/c/dam/m/en_us/network-intelligence/service-provider/digital-transformation/knowledge-network-webinars/pdfs/0426-techad-ckn.pdf

[5] mininet, Mininet: Rapid Prototyping for Software Defined Networks, (2021), Github Repository, https://github.com/mininet/mininet

[6] containernet, Containernet, (2019), Github Repository, https://github.com/containernet/containernet

[7] W. Braum, M. Menth, "Software Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices", Future Internet 2014, 6, 302-336, 12, May, 2014. [Online serial]. Available: https://www.researchgate.net/publication/284696928_Software-Defined_Networking_Using_OpenFlow_Protocols_Applications_and_Architectural_Design_Choices. [Accessed Feb. 4, 2021]

[8] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer and P. Shelar, "The Design and Implementation of Open vSwitch", presented in Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSD'15), Oakland, CA, USA, 2015

[9] ovs-actions(7), Linux Manual Page,[Online].Available: https://man7.org/linux/man-pages/man7/ovs-actions.7.html

[10] Tom, Python Multi-person Internet Chat Room, (2020), Github Repository, https://github.com/TomPrograms/Python-Internet-Chat-Room

[11] intrig-unicamp, Containernet, (2021), Github Repository, https://mininet-wifi.github.io/containernet/

[12] *"Cisco VPP SFC" 4, ODL SFC installation*,ProgrammerSought. Accessed on: Feb 12, 2021. [Online]. Available: https://www.programmersought.com/article/71403119192/

[13]openvswitch, ovs, (2019), Github Repository, https://github.com/openvswitch/ovs/blob/master/tests/nsh.at

[14] Y. Y. Yang, "[ovs-discuss] Supported transport encapsulation for NSH" August 6, 2018. [Accessed Feb 12, 2021]. [Online]. Available: https://mail.openvswitch.org/pipermail/ovs-discuss/2018-August/047148.html

[15] Docker Official Images, Rocket.chat, DockerHub, https://hub.docker.com/_/rocket-chat?tab=description

[16] sdelements, Lets chat , DockerHub, (2018), https://hub.docker.com/r/sdelements/lets-chat

[17] bubuntux, element, (2021), Github Repository, https://github.com/bubuntux/element-web