Network Slicing in an Emulated Network Environment(VoIP over MPLS)

Introduction: (Sami Patwary)

Network slicing is assumed to be one of the key enablers for the 5G concept. Our project has the functionality of this network slicing in an emulated environment. Our project tells us to allow the end-users to establish a Voice-over-IP communication to other end-users within the network slice. The network slicing concept prevents the establishment of a communication between end-users of two independent slices. The network slicing in the edge and transport network needs to be achieved through the utilization of a networking approach such as Multiprotocol Label Switching (MPLS). Possible extension made to the network to make it more versatile and user friendly.

Network slicing: (Abu Syeem MD Dipu)

Network slicing covers multiple virtual networks on top of a shared network. Each slice of the network can have its own logical topology, security commands and performance features, within the limits imposed by the underlying physical networks. Different slices can be assigned to different purposes, such as ensuring a specific application or service gets priority access to capacity and delivery or separating traffic for particular users or device classes. Slicing networks facilitates the network administrator to maximize the use of network resources and service versatility.

Why network slicing is needed? : The need for network slicing arises when each application should "see" a network configured in the optimal way to manage its traffic. This is possible with slicing, even if the network that the application sees is a "slice", a virtual "slice" of the physical network. The advantage of network slicing is not only in showing an application its ideal network. The various "slices" of the physical network are also isolated from each other and this, for example, guarantees both greater communication security and the possibility of modifying the operation of one slice without affecting that of the others. This means that an operator can drastically modify a service that it offers to its customers by only acting on one slice in a targeted manner. This would also allow conducting tests and research on dedicated slices, where each slice can mirror a production network, without affecting other slices that serve the live network with real services on top.[1]

When it is used? : The need comes when the network provider wants to share a common hardware infrastructure in order to support diverse services, each with specific requirements. It allows infrastructure sharing in a flexible and more efficient way, since the virtual slice can be “tailored” for the specific service, without including all the features of the underlying network. It can be said that slicing is used whenever there is a common infrastructure where on-demand customization is available, by assuring isolation, guaranteed performance, scalability, support for multi-vendor and multiple-operator scenarios.[1]

MPLS: (Abu Syeem MD Dipu)

Multiprotocol Label Switching (MPLS) is a routing strategy in telecommunications networks that directs data from one node to the next node. Rather than using long network addresses, MPLS is based on short path labels. In this way, it avoids complex lookups in a routing table and speed traffic flows.[4] The labels identify virtual links (paths) between distant nodes rather than endpoints. MPLS can capsulize packets of various network protocols, that's why it has 'multiprotocol' in its name. MPLS supports a variety of access technologies, like T1/E1, ATM, Frame Relay, DSL. In an MPLS network, each packet gets labeled on entry into the service provider's network by the ingress router, also known as the label edge router (LER). This is also the router that determines the LSP the packet will take until it reaches its destination address. All the following label-switching routers (LSRs) perform packet forwarding based only on those MPLS labels -- they never look as far as the IP header. At last, the egress router removes the labels and forwards the original IP packet toward its final destination. At the point when an LSR gets a packet, it performs one or more of the following tasks:

Push: Adds a label. This is normally performed by the ingress router.

Swap: Replaces a label. This is typically performed by LSRs between the ingress and egress routers.

Pop: Removes a label. This is frequently done by the egress router.[2]

In our project, we have used GRE tunneling over MPLS to send the packets securely.

VOIP: (Abu Syeem MD Dipu)

Voice over Internet Protocol (VoIP), is a technology that enables you to make voice calls using a broadband Internet connection rather than a regular (or analog) phone line. VoIP technology enables traditional telephony services to operate over computer networks using packet-switched protocols. Packet-switched VoIP puts voice signals into packets, similar to an electronic envelope. VoIP packets can be transmitted over any VoIP-compatible network, such as a local area network (LAN). In four steps, here’s how VoIP works:[3]

1. User’s phone connects to your switch or router in his Local Area Network (LAN).

2. When user dial a telephone number, his IP phone tells his VoIP service provider to call the other party.

3. User's VoIP service establishes the call and exchanges data packets from his IP phone.

4. User's VoIP phone converts these digital signals back into the sound you can hear.

There are two major reasons to use VOIP: Lower Cost & Increased functionality.

VoIP can also perform routing of incoming and outgoing calls through existing telephone networks. However, some VoIP services may only work over a computer or VoIP phone.

Environment: (Abu Syeem MD Dipu)

1.**Mininet:** Mininet works on a single machine that emulates a complete network of hosts, links, and switches. Interactive development, testing, and demos, especially those using OpenFlow and SDN, these are the use case of mininet. For full line-rate execution, OpenFlow-based network controllers prototyped in Mininet can normally be moved to hardware with insignificant changes.

Mininet produces virtual networks by implementing process-based virtualization and network namespaces. These features are available in recent Linux kernels. In Mininet, hosts are emulated as bash processes that run in a network namespace. So, any code that would normally run on a Linux server (like a web server or client program) should run perfectly within a Mininet "Host". The Mininet "Host" will have its individual private network interface and can only observe its own processes. Switches that present in Mininet are software-based switches like Open vSwitch or the OpenFlow reference switch. Links that live in the Linux kernel and connect our emulated switches to emulated hosts (processes) are virtual ethernet pairs.[4]

2. Containernet: There is a fork of the famous Mininet network emulator which is called Containernet. It allows to use Docker containers as hosts in emulated network topologies. This facilitates interesting functionalities to build networking/cloud emulators and testbeds. Containernet is actively practised by the research community, concentrating on experiments in the field of cloud computing, fog computing, network function virtualization (NFV), and multi-access edge computing (MEC).[5]

Open vSwitch: (Abu Syeem MD Dipu)

Open vSwitch is a production quality, multilayer virtual switch. It is outlined to enable huge network automation through programmatic extension, while still supporting standard management interfaces and opens the forwarding functions to programmatic extension and control. Open vSwitch is well suited to work as a virtual switch in VM environments. In addition to revealing standard control and visibility interfaces to the virtual networking layer, it was designed to support distribution across multiple physical servers. Open vSwitch supports multiple Linux-based virtualization technologies. Open vSwitch can also operate completely in userspace without support from a kernel module. Open vSwitch is targeted at multi-server virtualization deployments, a landscape for which the former stack is not well suited. These environments are often portrayed by highly dynamic end-points, the maintenance of logical abstractions, and (sometimes) combination with or offloading to special purpose switching hardware.[6]

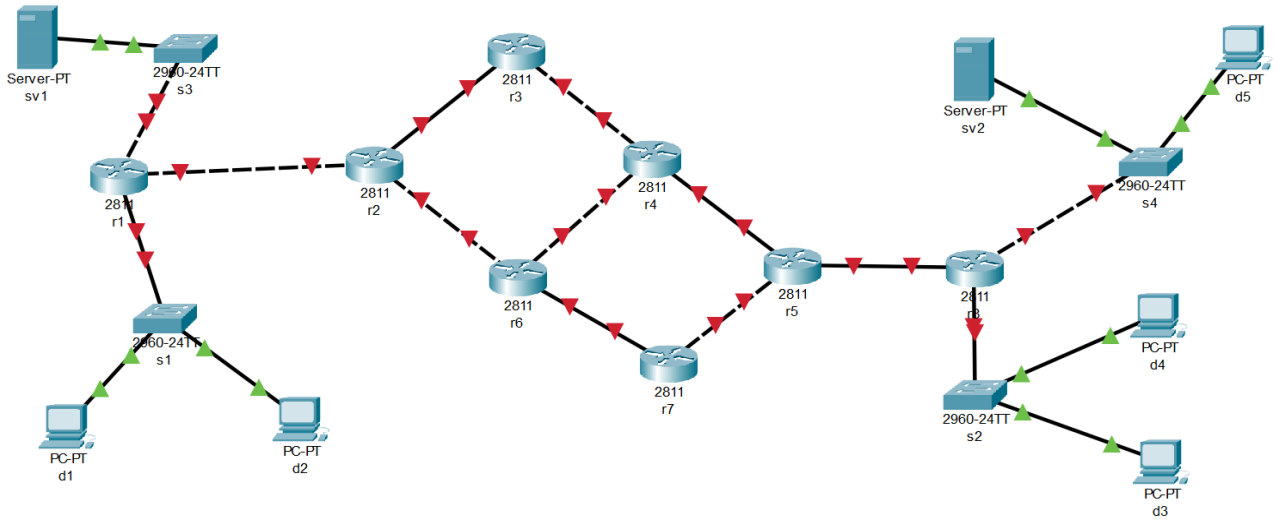
Network architecture: (Sami Patwary)

In this project we have built a network within the network emulation environment(Containernet) with 5 hosts (d1, d2, d3, d4 & d5) and 2 servers (sv1 & sv2). sv1 is a call server and sv2 is a DNS server. The nodes and switches within the network are configured statically. There are 4 switches (s1, s2, s3 & s4) and 8 routers (r1, r2, r3, r4, r5, r6, r7 & r8) in the network. We have created a GRE tunnel between r1 and r8 to transfer the packets securely. Then, we used an MPLS label over the GRE tunnel. We established a call server in R1 at port 2. DNS server is connected at port 2 of router r8. The 4 switches that we have used here are connected with a controller. With the help of these 4 switches, we have done the network slicing. We set up Linphone in the hosts to initiate a call between the hosts. Two users were created named 'syeem@20.20.20.5' and 'sami@20.20.20.5'. Docker images are used in the call server, the DNS server and in the hosts. Links of the docker images are given below:

1. DNS server: docker pull syeem007/dns:dns

2. Hosts: docker pull syeem007/phonehost:phone

3. Call server: docker pull syeem007/callserver:callserv



Containernet setup: (Sami Patwary)

1. At first, we have defined containernet, 8 routers, 1 controller and 4 switches. Ip is assigned to each of the router.

```
net = Containernet( controller=Controller, link=TCLink, switch=OVSKernelSwitch )
# define 8 routers and assign IP to them
r1 = net.addHost('r1', cls=LinuxRouter, ip='10.0.0.1/24')
r2 = net.addHost('r2', cls=LinuxRouter, ip='1.1.1.2/24')
r3 = net.addHost('r3', cls=LinuxRouter, ip='2.2.2.2/24')
r4 = net.addHost('r4', cls=LinuxRouter, ip='4.4.4.2/24')
r5 = net.addHost('r5', cls=LinuxRouter, ip='6.6.6.2/24')
r6 = net.addHost('r6', cls=LinuxRouter, ip='3.3.3.2/24')
r7 = net.addHost('r7', cls=LinuxRouter, ip='7.7.7.2/24')
r8 = net.addHost('r8', cls=LinuxRouter, ip='10.0.1.1/24')

c0 = net.addController( 'c0', controller=Controller, ip='127.0.0.1', port=6633 )
```

```
# Add 2 switches
s1 = net.addSwitch('s1')
s2 = net.addSwitch('s2')
s3 = net.addSwitch('s3')
s4 = net.addSwitch('s4')
```

2. Next, we have created the connection between routers and switches . At the routers' interfaces, we have assigned ip.

```
# connection between routers and switches and assign ip to the routers' interfaces
net.addLink(
            r1,s1,
            intfName1='r1-eth0',
            params1={'ip': '10.0.0.1/24'})

net.addLink(
            r8,s2,
            intfName1='r8-eth0',
            params1={'ip': '10.0.1.1/24'})
net.addLink(
            r1,s3,
            intfName1='r1-eth2',
            params1={'ip': '20.20.20.1/24'})
net.addLink(
            r8,s4,
            intfName1='r8-eth2',
            params1={'ip': '10.5.5.1/24'})
```

3. Now, we have built up the connection between routers and assigned ip to the routers' interfaces.

```
# router-router connection and assign ip to the routers' interfaces
net.addLink(r1,
            r2,
            intfName1='r1-eth1',
            intfName2='r2-eth0',
            params1={'ip': '1.1.1.1/24'},
            params2={'ip': '1.1.1.2/24'})

net.addLink(r2,
            r3,
            intfName1='r2-eth1',
            intfName2='r3-eth0',
            params1={'ip': '2.2.2.1/24'},
            params2={'ip': '2.2.2.2/24'})

net.addLink(r2,
            r6,
            intfName1='r2-eth2',
            intfName2='r6-eth0',
            params1={'ip': '3.3.3.1/24'},
            params2={'ip': '3.3.3.2/24'})

net.addLink(r3,
            r4,
            intfName1='r3-eth1',
            intfName2='r4-eth0',
            params1={'ip': '4.4.4.1/24'},
            params2={'ip': '4.4.4.2/24'})
```

```
net.addLink(r4,
            r6,
            intfName1='r4-eth1',
            intfName2='r6-eth1',
            params1={'ip': '5.5.5.1/24'},
            params2={'ip': '5.5.5.2/24'})

net.addLink(r6,
            r7,
            intfName1='r6-eth2',
            intfName2='r7-eth0',
            params1={'ip': '7.7.7.1/24'},
            params2={'ip': '7.7.7.2/24'})

net.addLink(r4,
            r5,
            intfName1='r4-eth2',
            intfName2='r5-eth0',
            params1={'ip': '6.6.6.1/24'},
            params2={'ip': '6.6.6.2/24'})

net.addLink(r5,
            r8,
            intfName1='r5-eth1',
            intfName2='r8-eth1',
            params1={'ip': '8.8.8.2/24'},
            params2={'ip': '8.8.8.1/24'})

net.addLink(r7,
            r5,
            intfName1='r7-eth1',
            intfName2='r5-eth2',
            params1={'ip': '9.9.9.1/24'},
            params2={'ip': '9.9.9.2/24'})
```

4. Then, we have specified the hosts and the servers. We have also assigned the default routes and the docker images for them.

```
# Adding hosts specifying the default route
d1 = net.addDocker(name='d1',
                   ip='10.0.0.251/24',
                   defaultRoute='via 10.0.0.1',dimage='phonehost:latest')
d2 = net.addDocker(name='d2',
                   ip='10.0.0.252/24',
                   defaultRoute='via 10.0.0.1',dimage='phonehost:latest')
d3 = net.addDocker(name='d3',
                   ip='10.0.1.251/24',
                   defaultRoute='via 10.0.1.1',dimage='phonehost:latest')
d4 =  net.addDocker(name='d4',
                   ip='10.0.1.252/24',
                   defaultRoute='via 10.0.1.1',dimage='phonehost:latest')
d5 = net.addDocker(name='d5',
                   ip='10.5.5.15/24',
                   defaultRoute='via 10.5.5.1',dimage='phonehost:latest')
sv1 = net.addDocker(name='sv1',
                   ip='20.20.20.5/24',
                   defaultRoute='via 20.20.20.1',dimage='callserv:latest')
sv2 = net.addDocker(name='sv2',
                   ip='10.5.5.5/24',
                   defaultRoute='via 10.5.5.1',dimage='dhcp_server:latest')
```

5. Host-switch connection has also applied therefore.

```
# Add host-switch link
net.addLink(d1, s1)
net.addLink(d2, s1)
net.addLink(d3, s2)
net.addLink(d4, s2)
net.addLink(sv1,s3)
net.addLink(sv2,s4)

net.build()
c0.start()
s1.start( [c0] )
s2.start( [c0] )
s3.start( [c0] )
s4.start( [c0] )
```

6. We have assigned MAC in both interfaces of router r1 and router r8 and have done the OVS filtering in all switches for network slicing.

```
# assign MAC in the routers'(r1 & r8) interfaces and OVS Filtering in the switches
info(net['r1'].cmd("ifconfig r1-eth0 hw ether 00:00:00:00:01:01"))
info(net['r8'].cmd("ifconfig r8-eth0 hw ether 00:00:00:00:01:02"))
info(net['r1'].cmd("ifconfig r1-eth2 hw ether 00:00:00:00:01:03"))
info(net['r8'].cmd("ifconfig r8-eth2 hw ether 00:00:00:00:01:04"))
info(net['s1'].cmd("ovs-ofctl add-flow s1 priority=1,arp,actions=flood"))
info(net['s1'].cmd("ovs-ofctl add-flow s1 priority=65535,ip,dl_dst=00:00:00:00:01:01,actions=output:1"))
info(net['s1'].cmd("ovs-ofctl add-flow s1 priority=10,ip,nw_src=10.0.1.251,nw_dst=10.0.0.251,actions=output:2"))
info(net['s1'].cmd("ovs-ofctl add-flow s1 priority=10,ip,nw_src=20.20.20.5,nw_dst=10.0.0.251,actions=output:2"))
info(net['s1'].cmd("ovs-ofctl add-flow s1 priority=10,ip,nw_src=10.0.1.252,nw_dst=10.0.0.252,actions=output:3"))
info(net['s1'].cmd("ovs-ofctl add-flow s1 priority=10,ip,nw_src=20.20.20.5,nw_dst=10.0.0.252,actions=output:3"))
info(net['s1'].cmd("ovs-ofctl add-flow s1 priority=10,ip,nw_src=10.5.5.5,nw_dst=10.0.0.251,actions=output:2"))
info(net['s1'].cmd("ovs-ofctl add-flow s1 priority=10,ip,nw_src=10.5.5.5,nw_dst=10.0.0.252,actions=output:3"))
info(net['s2'].cmd("ovs-ofctl add-flow s2 priority=1,arp,actions=flood"))
info(net['s2'].cmd("ovs-ofctl add-flow s2 priority=65535,ip,dl_dst=00:00:00:00:01:02,actions=output:1"))
info(net['s2'].cmd("ovs-ofctl add-flow s2 priority=10,ip,nw_src=10.0.0.251,nw_dst=10.0.1.251,actions=output:2"))
info(net['s2'].cmd("ovs-ofctl add-flow s2 priority=10,ip,nw_src=20.20.20.5,nw_dst=10.0.1.251,actions=output:2"))
info(net['s2'].cmd("ovs-ofctl add-flow s2 priority=10,ip,nw_src=10.0.0.252,nw_dst=10.0.1.252,actions=output:3"))
info(net['s2'].cmd("ovs-ofctl add-flow s2 priority=10,ip,nw_src=20.20.20.5,nw_dst=10.0.1.252,actions=output:3"))
info(net['s2'].cmd("ovs-ofctl add-flow s2 priority=10,ip,nw_src=10.5.5.5,nw_dst=10.0.1.251,actions=output:2"))
info(net['s2'].cmd("ovs-ofctl add-flow s2 priority=10,ip,nw_src=10.5.5.5,nw_dst=10.0.1.252,actions=output:3"))
info(net['s3'].cmd("ovs-ofctl add-flow s3 priority=1,arp,actions=flood"))
info(net['s3'].cmd("ovs-ofctl add-flow s3 priority=65535,ip,dl_dst=00:00:00:00:01:03,actions=output:1"))
info(net['s3'].cmd("ovs-ofctl add-flow s3 priority=10,ip,nw_dst=20.20.20.5,actions=output:2"))
info(net['s4'].cmd("ovs-ofctl add-flow s4 priority=1,arp,actions=flood"))
info(net['s4'].cmd("ovs-ofctl add-flow s4 priority=65535,ip,dl_dst=00:00:00:00:01:04,actions=output:1"))
info(net['s4'].cmd("ovs-ofctl add-flow s4 priority=10,ip,nw_dst=10.5.5.5,actions=output:2"))
```

7. Then we have done the static routing.

```
# Add routing for reaching networks that aren't directly connected
    info(net['r1'].cmd("ip route add 2.2.2.0/24   via 1.1.1.2 dev r1-eth1"))
    info(net['r1'].cmd("ip route add 4.4.4.0/24   via 1.1.1.2 dev r1-eth1"))
    info(net['r1'].cmd("ip route add 6.6.6.0/24   via 1.1.1.2 dev r1-eth1"))
    info(net['r1'].cmd("ip route add 4.4.4.0/24   via 1.1.1.2 dev r1-eth1"))
    info(net['r1'].cmd("ip route add 8.8.8.0/24   via 1.1.1.2 dev r1-eth1"))
    info(net['r1'].cmd("ip route add 3.3.3.0/24   via 1.1.1.2 dev r1-eth1"))
    info(net['r1'].cmd("ip route add 7.7.7.0/24   via 1.1.1.2 dev r1-eth1"))
    info(net['r1'].cmd("ip route add 9.9.9.0/24   via 1.1.1.2 dev r1-eth1"))
    info(net['r1'].cmd("ip route add 5.5.5.0/24   via 1.1.1.2 dev r1-eth1"))
    info(net['r2'].cmd("ip route add 4.4.4.0/24   via 2.2.2.2 dev r2-eth1"))
    info(net['r2'].cmd("ip route add 6.6.6.0/24   via 2.2.2.2 dev r2-eth1"))
    info(net['r2'].cmd("ip route add 8.8.8.0/24   via 2.2.2.2 dev r2-eth1"))
    info(net['r2'].cmd("ip route add 7.7.7.0/24   via 3.3.3.2 dev r2-eth2"))
    info(net['r2'].cmd("ip route add 5.5.5.0/24   via 3.3.3.2 dev r2-eth2"))
    info(net['r2'].cmd("ip route add 9.9.9.0/24   via 3.3.3.2 dev r2-eth2"))
    info(net['r3'].cmd("ip route add 1.1.1.0/24   via 2.2.2.1 dev r3-eth0"))
    info(net['r3'].cmd("ip route add 3.3.3.0/24   via 2.2.2.1 dev r3-eth0"))
    info(net['r3'].cmd("ip route add 5.5.5.0/24   via 4.4.4.2 dev r3-eth1"))
    info(net['r3'].cmd("ip route add 7.7.7.0/24   via 4.4.4.2 dev r3-eth1"))
    info(net['r3'].cmd("ip route add 6.6.6.0/24   via 4.4.4.2 dev r3-eth1"))
    info(net['r3'].cmd("ip route add 9.9.9.0/24   via 4.4.4.2 dev r3-eth1"))
    info(net['r3'].cmd("ip route add 8.8.8.0/24   via 4.4.4.2 dev r3-eth1"))
    info(net['r4'].cmd("ip route add 1.1.1.0/24   via 4.4.4.1 dev r4-eth0"))
    info(net['r4'].cmd("ip route add 2.2.2.0/24   via 4.4.4.1 dev r4-eth0"))
    info(net['r4'].cmd("ip route add 3.3.3.0/24   via 5.5.5.2 dev r4-eth1"))
    info(net['r4'].cmd("ip route add 7.7.7.0/24   via 6.6.6.2 dev r4-eth2"))
    info(net['r4'].cmd("ip route add 9.9.9.0/24   via 6.6.6.2 dev r4-eth2"))
    info(net['r4'].cmd("ip route add 8.8.8.0/24   via 6.6.6.2 dev r4-eth2"))
    info(net['r5'].cmd("ip route add 4.4.4.0/24   via 6.6.6.1 dev r5-eth0"))
    info(net['r5'].cmd("ip route add 2.2.2.0/24   via 6.6.6.1 dev r5-eth0"))
    info(net['r5'].cmd("ip route add 1.1.1.0/24   via 6.6.6.1 dev r5-eth0"))
    info(net['r5'].cmd("ip route add 3.3.3.0/24   via 6.6.6.1 dev r5-eth0"))
    info(net['r5'].cmd("ip route add 5.5.5.0/24   via 6.6.6.1 dev r5-eth0"))
    info(net['r5'].cmd("ip route add 7.7.7.0/24   via 9.9.9.1 dev r5-eth2"))
    info(net['r6'].cmd("ip route add 1.1.1.0/24   via 3.3.3.1 dev r6-eth0"))
    info(net['r6'].cmd("ip route add 2.2.2.0/24   via 3.3.3.1 dev r6-eth0"))
    info(net['r6'].cmd("ip route add 4.4.4.0/24   via 5.5.5.1 dev r6-eth1"))
    info(net['r6'].cmd("ip route add 6.6.6.0/24   via 3.3.3.1 dev r6-eth0"))
    info(net['r6'].cmd("ip route add 9.9.9.0/24   via 7.7.7.2 dev r6-eth2"))
    info(net['r6'].cmd("ip route add 8.8.8.0/24   via 7.7.7.2 dev r6-eth2"))
    info(net['r7'].cmd("ip route add 1.1.1.0/24   via 7.7.7.1 dev r7-eth0"))
    info(net['r7'].cmd("ip route add 2.2.2.0/24   via 7.7.7.1 dev r7-eth0"))
    info(net['r7'].cmd("ip route add 3.3.3.0/24   via 7.7.7.1 dev r7-eth0"))
    info(net['r7'].cmd("ip route add 5.5.5.0/24   via 7.7.7.1 dev r7-eth0"))
    info(net['r7'].cmd("ip route add 4.4.4.0/24   via 7.7.7.1 dev r7-eth0"))
    info(net['r7'].cmd("ip route add 6.6.6.0/24   via 9.9.9.2 dev r7-eth1"))
    info(net['r7'].cmd("ip route add 8.8.8.0/24   via 9.9.9.2 dev r7-eth1"))
    info(net['r8'].cmd("ip route add 1.1.1.0/24   via 8.8.8.2 dev r8-eth1"))
    info(net['r8'].cmd("ip route add 2.2.2.0/24   via 8.8.8.2 dev r8-eth1"))
    info(net['r8'].cmd("ip route add 3.3.3.0/24   via 8.8.8.2 dev r8-eth1"))
    info(net['r8'].cmd("ip route add 4.4.4.0/24   via 8.8.8.2 dev r8-eth1"))
    info(net['r8'].cmd("ip route add 5.5.5.0/24   via 8.8.8.2 dev r8-eth1"))
    info(net['r8'].cmd("ip route add 6.6.6.0/24   via 8.8.8.2 dev r8-eth1"))
    info(net['r8'].cmd("ip route add 7.7.7.0/24   via 8.8.8.2 dev r8-eth1"))
    info(net['r8'].cmd("ip route add 9.9.9.0/24   via 8.8.8.2 dev r8-eth1"))
```

8. GRE has done on the routers r1 & r8.

```
# Apply GRE on r1 & r8

    info(net['r1'].cmd('ip tun a foo1 mode gre remote 8.8.8.1 local 1.1.1.1'))
    info(net['r1'].cmd('ip addr a 100.100.100.1 dev foo1'))
    info(net['r1'].cmd('ip link set foo1 up'))
    info(net['r1'].cmd('ip rou a 100.100.100.0/24 dev foo1'))
    info(net['r8'].cmd('ip tun a foo1 mode gre remote 1.1.1.1 local 8.8.8.1'))
    info(net['r8'].cmd('ip addr a 100.100.100.2 dev foo1'))
    info(net['r8'].cmd('ip link set foo1 up'))
    info(net['r8'].cmd('ip rou a 100.100.100.0/24 dev foo1'))
```

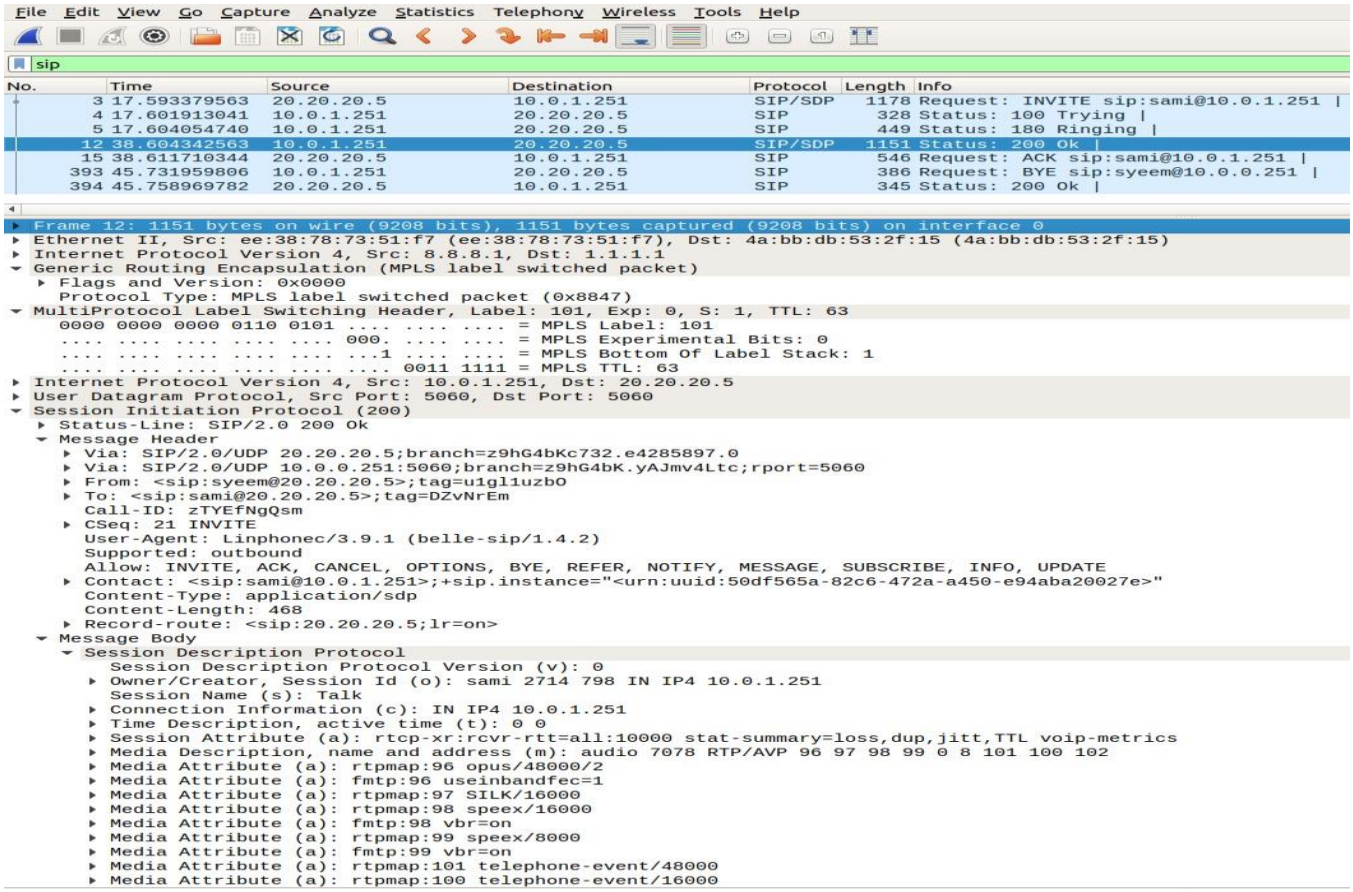
9. Finally, we have applied MPLS on r1 & r8.

```
    #Apply MPLS on r1 & r8

        info(net['r1'].cmd('sysctl -w net.ipv4.conf.all.rp_filter=2'))
        info(net['r1'].cmd('sysctl -w net.mpls.platform_labels=65535'))
        info(net['r1'].cmd('sysctl -w net.mpls.conf.foo1.input=1'))
        info(net['r1'].cmd('ip rou c 100.100.100.0/24 encap mpls 100 dev foo1'))
        info(net['r1'].cmd('ip -f mpls rou a 101 dev lo'))

        info(net['r8'].cmd('sysctl -w net.ipv4.conf.all.rp_filter=2'))
        info(net['r8'].cmd('sysctl -w net.mpls.platform_labels=65535'))
        info(net['r8'].cmd('sysctl -w net.mpls.conf.foo1.input=1'))
        info(net['r8'].cmd('ip rou c 100.100.100.0/24 encap mpls 101 dev foo1'))
        info(net['r8'].cmd('ip -f mpls rou a 100 dev lo'))
```

VoIP over MPLS: (Sami Patwary)

1. At first, we cleared the cache memory and then run the python code. The name of our python code is "newencapsulation.py". We can see that the images within the host is loaded. We can see the ethernet connection by "links" command.

2. Then, we started the call server. For that, we initiated mysql and kamailio service. In another window, we started router r1 and run wireshark there. In another window, we started d1 & d3 hosts. In d1 host, we registered user syeem@20.20.20.5 and in host d2, we registered sami@20.20.20.5. Then we placed call from host d1 to host d3. On d3 window, call was answered and then the call was terminated by any of the hosts. In the wireshark window we can see the information of all packets.



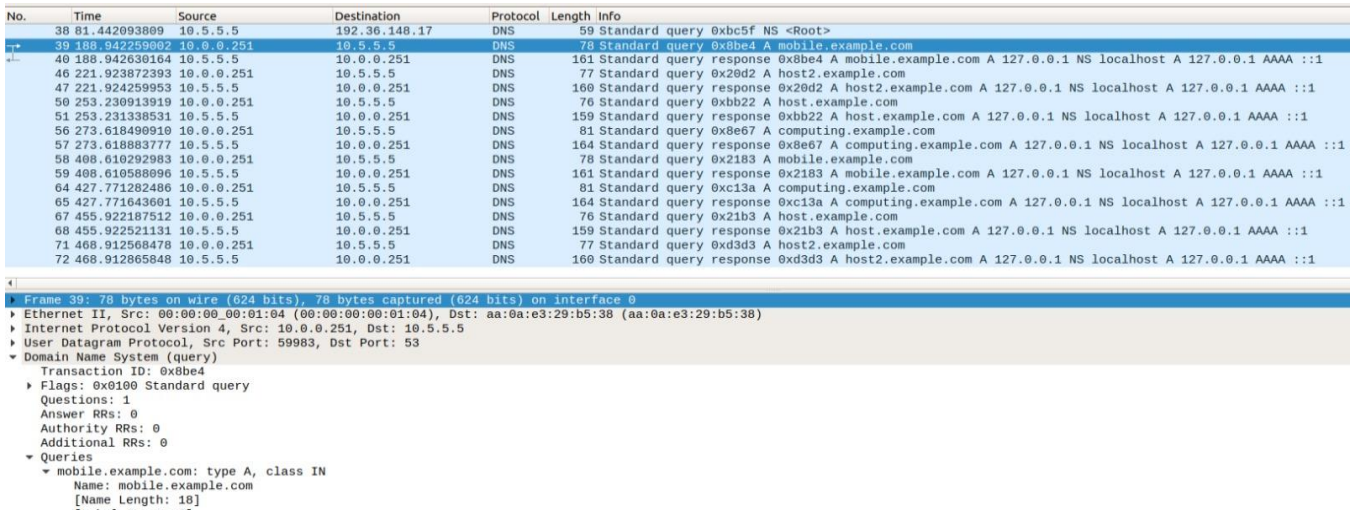
Similarly, we can make a call from d2 to d4 and vice-versa.

In our project, we have used network slicing between d1, d3 & d2, d4. No communication can be made except d1, d3 and d2, d4. If we ping from d1 to d4, we can see that no packet will be transferred.

Project extension: We have added a DNS server at router r8.

DNS service step: (Abu Syeem MD Dipu)

First, we have opened 'bind9" service at the DNS server "sv2". Then we have declared the DNS server to the hosts(d1,d2,d3,d4). we have created the hosts name as mobile@example.com , host@example.com, host2@example.com, computing@example.com . If we ping any of these hosts, we can see the corresponding DNS packets in the wireshark running at router r8.



Problem faced & how did we overcome it: (Abu Syeem MD Dipu)

In our Project, we were doing static routing. While sending the traffics, the traffic were not going through MPLS but other routing. But our VOIP packets supposed to be sent off through MPLS. To overcome this, we created a GRE tunnel. Virtual IP was there over the GRE tunnel where we created the tag for MPLS. Then we did static routing to the hosts network that all the

```
r1.cmd('ip route add 10.0.1.0/24 encap mpls 100 via 100.100.100.2 dev foo1')
r8.cmd('ip route add 10.0.0.0/24 encap mpls 101 via 100.100.100.1 dev foo1')
r8.cmd('ip route add 20.20.20.0/24 encap mpls 101 via 100.100.100.1 dev foo1')
r1.cmd('ip route add 10.5.5.0/24 encap mpls 100 via 100.100.100.2 dev foo1')
```

traffic should go over the MPLS tag. We have let the virtual IP of tunnel know that all the incoming traffic should go over the MPLS tag.

Conclusion: (Sami Patwary)

In this project, we have achieved network slicing in an emulated network environment. Our category was VOIP over MPLS. For achieving this, we have created a network in containernet and did network slicing between 4 hosts (d1,d3 & d2,d4). We have successfully initiated VoIP call between d1-d3 and d2-d4 and vice-versa. No call can be initiated except these. In this way, we have applied the network slicing. As a means of extension, we have built a DNS server in the network to achieve the DNS service. While implementing this project, we have faced a few difficulties which we have overcome successfully.

References:

1. https://www.reply.com/en/industries/telco-and-media/5g-network-slicing
2. https://searchnetworking.techtarget.com/definition/Multiprotocol-Label-Switching-MPLS
3. https://whatis.techtarget.com/definition/network-slicing https://www.nextiva.com/blog/what-is-voip.html
4. https://github.com/mininet/mininet
5. https://containernet.github.io/
6. https://docs.openvswitch.org/en/latest/intro/what-is-ovs/