Hypervisor Overlay Networking in Multi-tenant Environment

Team 6

Aneesh Joshi, Luv Khurana, Pranjal Sharma, Sai Jayesh Bondu

Project Description:

SDN-based deployment of Hypervisor Overlay Networks using VXLAN and GRE tunnels to connect customer containers in a multi-tenant virtual private cloud environment, where customer has control over their subnets.

Background:

Network Virtualization:
Network Virtualization is the technique that creates logical (virtual) networks that are decoupled from the underlying hardware. This decoupling ensures that the network can support virtual environments such as modern data centers, cloud etc. Network Virtualization came into existence to solve problems in data center networking such as:

MAC-table overflow:
In Data Center and Cloud environments, number of entries in a switch's MAC address table can fill up very quickly due to the presence of a really large number of virtual end systems in addition to physical end systems. This causes subsequent frames to be broadcasted in the LAN, thereby wasting bandwidth.

Solution:
Network Virtualization makes use of tunnels to encapsulate original L2 frames. Therefore, the only mac addresses that appear on the physical network are those of tunnel endpoints.

VLAN exhaustion: 4096 VLANs were enough in the past, but with the advent of cloud computing and large data centers, this number is not enough.

Solution: Most Network Virtualization technologies that exist today allow a minimum of 2^{24} tunnels (~16 Million) to be configured, thereby overcoming the VLAN exhaustion problem at a virtual network level.

Current Technologies:

VXLAN (Virtual eXtensible LAN):
In VXLAN, entire frame is encapsulated in a new packet. VXLAN uses UDP (destination port 8472). VXLAN uses a Network Identifier which is 24 bits. Like other tunneling technologies, there are unicast packets exchanged between the two tunnel endpoints.

NVGRE (Network Virtualization using Generic Routing Encapsulation):
In NVGRE, entire frame is encapsulated in a new packet. NVGRE doesn't use TCP/UDP, it uses GRE tunneling protocol. Like VXLAN, NVGRE uses a Network Identifier of 24 bits. Like other tunneling technologies, there are unicast packets exchanged between the two tunnel endpoints. Unlike, VXLAN, NVGRE header contains an optional FlowID field, which can be used to differentiate between flows.

STT (Stateless Transport Tunneling):
In STT, entire frame is encapsulated in a new packet. STT uses TCP, although as the name suggests, it is stateless. TCP is used to make use of functionality in server NICs. Network Identifier for STT is 64 bits. Even with STT, unicast packets are exchanged between the tunnel endpoints.

Related work:

The following 3 technologies have been evaluated.

1. VMWare NSX Data Center

2. Nuage VSP

3. Hyper-V network virtualization

All the above-mentioned products are offering network virtualization facilities to the user through control of their own subnets.

The interface provided by the VMWare's NSX Extensibility is very intuitive, easy to operate and provides high flexibility. It uses VxLAN tunneling. REST APIs have also been used prominently. Apart from this, features such as Load-balancing, VPN and Firewall are also present. Another thing to note is that NSX needs hardware termination, while the others use MPLS over GRE.

Nuage's VSP offers excellent cloud-infrastructure management service. A special feature is that customer does not need proprietary hardware for its operation. It is compatible with customer's existing hardware infrastructure. Scaling up the network is very simple.

Hyper-V network virtualization platform uses NVGRE technology. Feature of live migration is provided. Another interesting feature is compatibility with existing network infrastructure, which provides plug and play operation.

Our system tries to mimic from the current providers. Here we are providing the control of subnets and end to end connectivity across VPCs via tunnels. We offer both GRE and VXLAN tunneling solutions. We also provide features such as cold VM migration and fail-over case by providing back-up bridges in tenant VPCs.
Above solutions also do provide similar features but there is the feature of live VM migration.

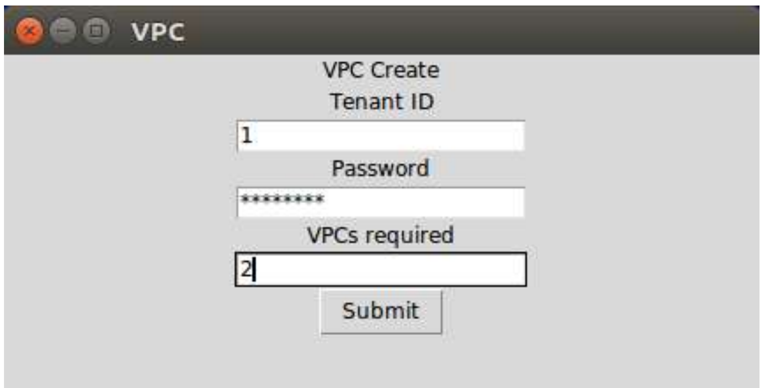
Management Features:

1) Authentication

The tenant is supposed to authenticate himself by providing accurate login details. This ensures that only eligible user gets access to the network database.

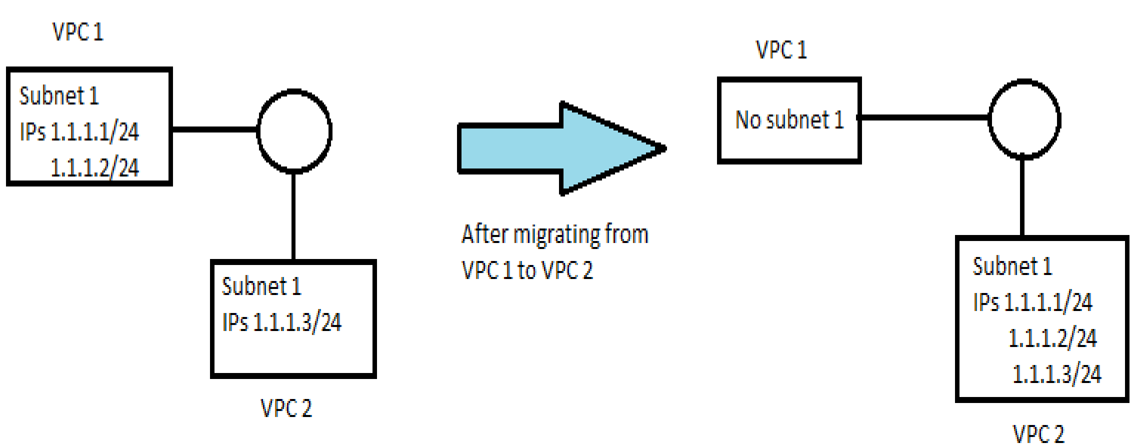
2) Configurability

When an overlay network is deployed, the designer must ensure that the overlay can be reconfigured based on the demands of the customer. Reconfiguration can be on the grounds of number of VPCs, number of subnets and number of VMs inside the subnets. Special importance must be given to ease of reconfiguration. APIs can be exposed for the purpose of automation.



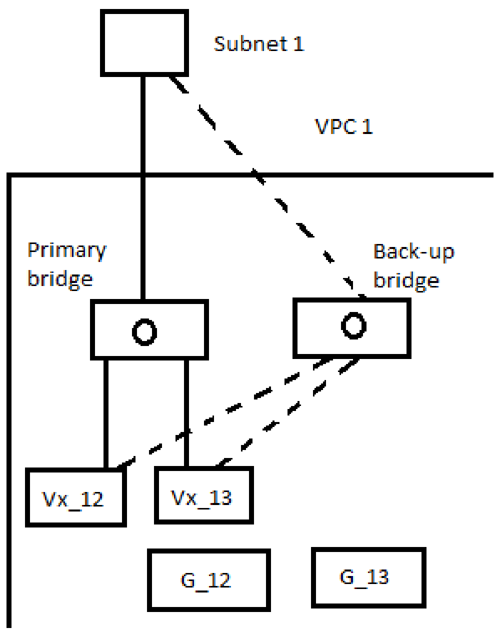
3) Flexibility

To achieve flexibility, we offer on-demand subnet migration capability (cold migration) to the customer. Through this feature the user can control and modify the presence of his subnets in different VPCs. Tunnels are adjusted accordingly.



4) Availability

To provide uninterrupted service, the overlay network should always be available. We intend to provide uninterrupted service through a standby bridge in case the primary bridge goes down. So, we provide back-up bridges and join original tunnel interfaces there.



Software Architecture:

North-bound interface

A form will be provided to the user. For demo purpose, we have used an interactive form, but user cannot automate through a UI. For that purpose, we intend to expose APIs to the user.
User is prompted with:

- Authentication password
- Number of VPCs
- Subnet IDs

If the user doesn't specify some options, defaults will be used.
Eg. Number of VPCs for that tenant will be chosen as 2 if not specified.

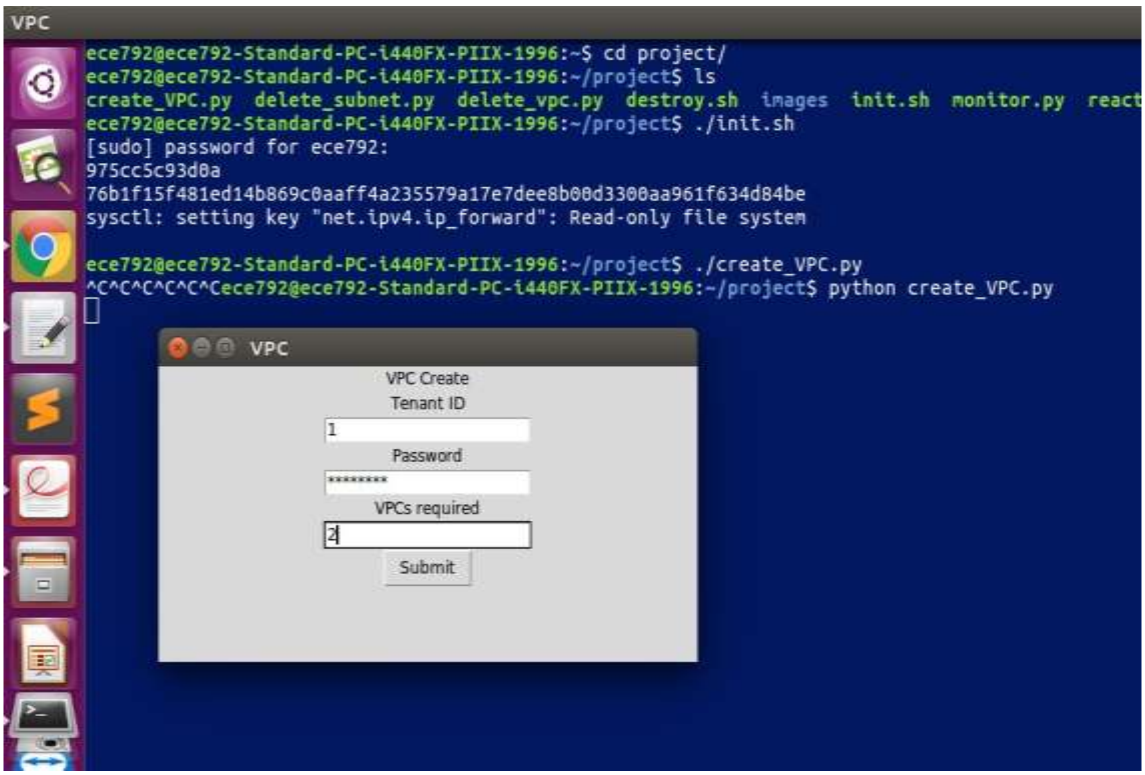


Figure 1: Form displayed to the user

Logic Layer:

When parameters are provided by the user, they are used to generate different parameters associated with their topology. Their database file is maintained as <database_TenantID.db>

The key operation is storing of all the newly-created fields in a database. This is a very important factor as any further operations to be performed on the tenant's subnets or locations, assigned names, etc. will all be accessed through the above maintained table.

The scripts have been developed by employing the CRUD model (Create, Read, Update, Delete). This same model will be employed for operation on VPCs, subnets and tunnels. Default configurations will also be taken care of.

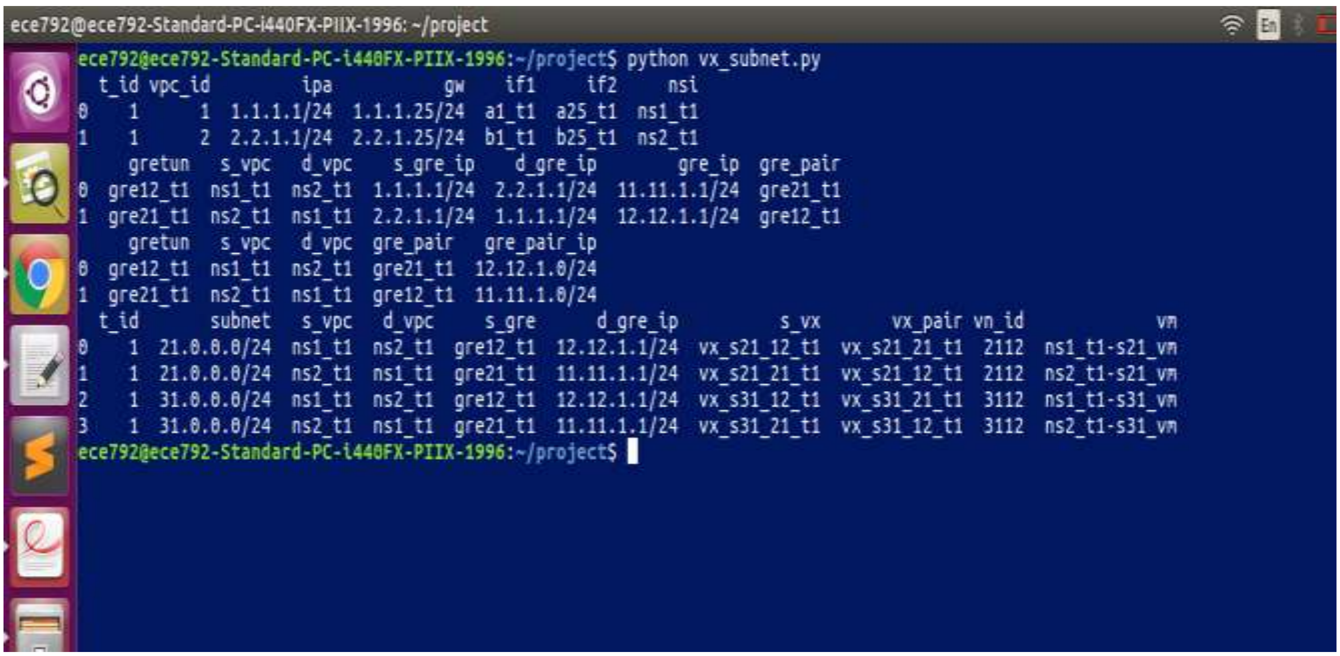


Figure 2: Database table

South-bound interface:

The southbound layer comprises of iproute2 package of linux with python. This will help do the infrastructure related tasks such as creation of namespaces, tunnel devices, v-eth pairs and different configurations (Including default setup)

After the setup has been done, and the tunneling interfaces for VXLAN and GRE up, then tunneling functionality can be accessed.

We expect the user to provide parameters regarding the VPCs. But in case, the customer doesn't know, a default topology will be used.

Another thing to note is that, by default, we will provide VxLAN devices for connecting VPCs. If the customer doesn't want to use them, we set the VxLAN devices down. This way of provisioning enables easy toggling of the VxLAN service as per customer's future wishes.

<u>Southbound topology:</u>

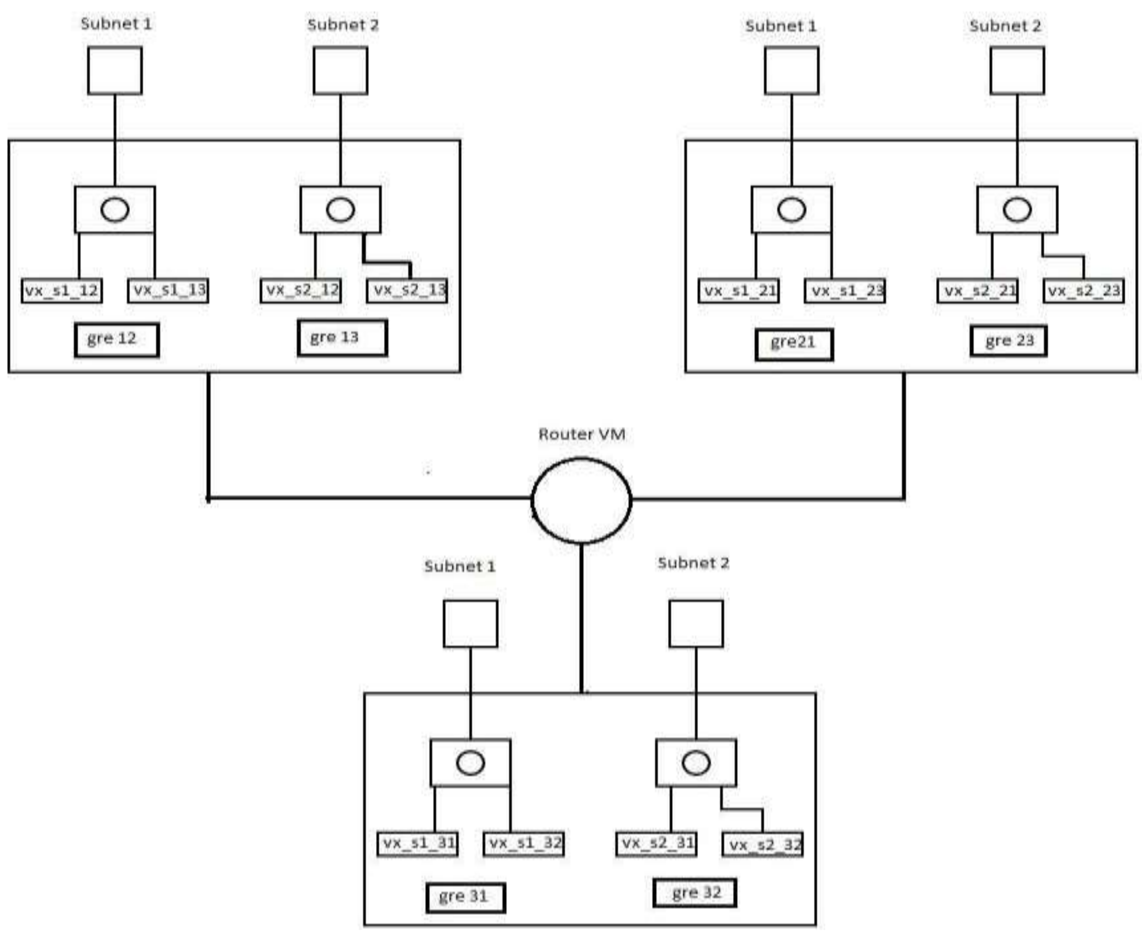


Figure 3: Typical architecture for 1 tenant with multiple subnets in a VPC.

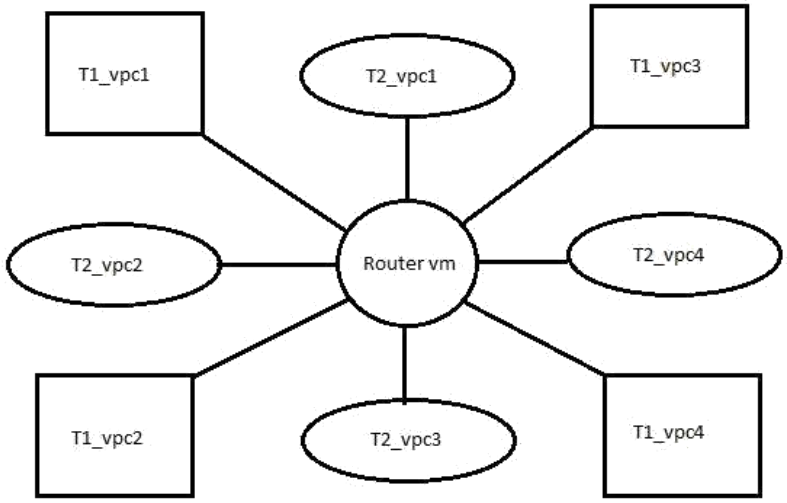


*Figure 4: Possible architecture for mu*ltiple tenants.*

-------------- User Guide ---------------

I. New tenant Guide

This is for the new tenant. You can skip this section, if you are an existing tenant

1.1 Getting started:

As a new tenant you are required to setup a password for accessing our solution, and further to configure your VPCs

1.2 Setting the tenant ID:

- You are free to choose any tenant id - (must be an integer)
- It must be unique. This will be checked by comparing with other tenants and will be prompted to re-enter tenant ID.

1.3 Setting up the password:

- Every new tenant is required to setup the password for accessing their data and cloud environment.
- These passwords are stored in the separate providers database and will be used to authenticate.

II. Existing tenant Guide

These will guide you through some of the management features that we will be providing for configuring your topology, meeting your needs.

2. Setting up the topology

Tenants are provided with 3 options to setup their cloud environment. For starters, we would suggest going with the default solution.

2.1 Default option:

As a default option. Enter the tenant ID, the password and the number of VPCs required. We will be creating an underlay, setting up the connectivity among your VPCs

Run the C_vpc.py and vx_subnet.py – refer to the readme

2.2 JSON input:

An option to provide the JSON file, for configuring the tunnels among your existing subnets present in different VPCs. You are specified to provide the subnets that you want to connect them using tunnels

Upload the json file in the /home/demo/topology.json

2.3 Default through REST API:

We are providing a REST API to create VPC for a tenant, only the Read, Create and Delete endpoints are been exposed, we are working on the Edit endpoint. This will be exposed in our 2.0 version.

3. Configuring VPCs

CRUD operations for VPCs

3.1 Create VPC:

Specify the tenant ID and password, provider will retrieve the existing topology, just enter the VPC number that you want to add as extra

We will configure the underlay to connect this VPC with other VPCs

Run create_vpc.py – refer readme

3.2 Read VPC:

Enter the tenant ID and vpc number from the existing list of VPCs, this will allow you to view the specific VPC's and the subnets available in them

Run read_vpc.py – refer readme

3.3 Delete VPC:

Deleting a VPC will delete all the existing subnets present in that VPC. This will result in removal of the existing tunnels.

Run delete_vpc.py – refer readme

4. Configuring Subnets

4.1 Create Subnet:

Specify the tenant ID and password and the VPC that you want to create the subnet in, provider will retrieve the existing topology, just enter the subnet ID number that you want to add as extra

We will configure the underlay to connect this subnet with bridges and link them to other existing subnets by adding a bridge and creating underlay for communication

Run create_subnet.py – refer readme

4.2 Read Subnet:

Enter the tenant ID and vpc number from the existing list of VPCs, this will allow you to view the specific VPC's and the subnets available in them

Run read_subnet.py – refer readme

4.3 Delete Subnet:

Deleting a subnet will delete all the existing VMs present in that Subnet. This will result in removal of the existing tunnels.

Run delete_vpc.py – refer readme

5. Configuring Subnets - High availability

To achieve high availability of the VPC service, a tenant can choose to create subnets with High Availability option. Doing so, would set up backup bridges in case the primary bridges fail. Once an active bridge fails, the event will be detected, and changes will be made to ensure that the backup bridge is now active. The backup bridge will now start handling all the traffic for that subnet.

5.1 Create Subnets with high availability

To create subnets with high availability, user executes:

python vx_subnet_HA.py

This script creates the infrastructure for active as well as backup paths.

5.2 Monitor and Failover

To detect bridge failures, user executes:

#python monitor.py

This program starts tracking bridge failures for each tenant. Once a failure is detected, this program invokes another that makes the backup bridge the active one.

6. Subnet Migration

To migrate a subnet from VPC to another VPC, user is prompted to enter the source VPC, destination VPC and subnet ID. The new vxlan devices are created and end points are taken based on the destination VPC to which subnet is migrating.

Run migrate_sub.py – refer readme

Note: The constraint for this subnet migration is that, the user cannot migrate his subnet to a VPC which already has that subnet, so the prime requirement is that, the subnet must not exist in that VPC

-------------- Developer Guide ---------------

1. Configuring Subnet

The function to create the subnet resides with the script vx_subnet.py
To implement a new subnet at a specific VPC has been provided by using the following blocks of the script
Subnet info is obtained from the user, store that value to use it in the query
Query to get the existing vpcs:

```
SELECT * from vx_pair WHERE subnet = $subnet
```

1.1 Create subnet:

The subnet info obtained from the query is used in the following commands

```
#creating a namespace for the subnet  -   attaching a namespace to an existing namespace
   res = subprocess.check_output(["sudo","ip","netns","add",vm])
 #adding interfaces
   res = subprocess.check_output(["sudo","ip","link","add",if1,"type","veth","peer","name",if2])
   res = subprocess.check_output(["sudo","ip", "link", "set", if1, "netns", vm])
   res = subprocess.check_output(["sudo","ip", "link", "set", if2, "netns", j])

   #adding ip address to veth pairs
   res = subprocess.check_output(["sudo","ip","netns","exec",vm,"ip","addr","add",ipa+"/24","dev",if1])
   #res = subprocess.check_output(["sudo","ip","netns","exec",j,"ip","addr","add",gw,"dev",if2])

   #Turning on veth pairs
   res = subprocess.check_output(["sudo","ip","netns","exec",vm,"ip","link","set","dev",if1,"up"])
   res = subprocess.check_output(["sudo","ip","netns","exec",j,"ip","link","set","dev",if2,"up"])
   # create bridge
   res = subprocess.check_output(["sudo","ip","netns","exec",j,"brctl","addbr",br])
```

1.2 Read Subnet:

This is used to read the subnet information existing in the VPC of one particular tenant.

```
conn = sqlite3.connect(database)
c = conn.cursor()
c.execute("SELECT name FROM sqlite_master WHERE type='table';")
l = c.fetchall()
for table_name in l:
   #print(table_name)
   db = sqlite3.connect(database)
   table = pd.read_sql_query("SELECT * from %s" % table_name, db)
   print(table)
```

1.3 Delete Subnet:

The changes can be made based on the VPC and specify the place where the tenant needs his subnet to be removed

```
c = conn.cursor()
def get_br_by_subnet(subnet_id,nsi):
```

Query to get a specific subnet

```
  SELECT * FROM vx_lan WHERE subnet_id=:subnet_id AND nsi = :nsi
br = get_br_by_subnet(sub,ns+'_'+tid)
nsi = "ns"+str(ns)
conn.close()
subprocess.check_output(["sudo","ip","netns",nsi,"exec","brctl","delbr",br])
```

2. Configuring Subnets - High availability

To achieve high availability of the VPC service, a tenant can choose to create subnets with High Availability option. Doing so, would set up backup bridges in case the primary bridges fail. Once an active bridge fails, the event will be detected, and changes will be made to ensure that the backup bridge is now active. The backup bridge will now start handling all the traffic for that subnet.

2.1 Create Subnets with high availability

To enhance the functionality for creation of subnets with high availability, 'vx_subnet_HA.py' needs to be edited.
By default, the code creates 1 backup switch (and a backup veth pair) per created subnet.
To increase the number of backup of bridges:

a) Create backup veth pair:

```
subprocess.check_output(["sudo","ip","link","add",bk1,"type","veth","peer","name",bk2])
```

b) Attach backup veth pair ends to container (or vm) as well as VPC namespace:

```
pid_vm = subprocess.check_output(["sudo","docker","inspect","--format","'{{.State.Pid}}'", vm])
subprocess.check_output(["sudo","ip","link","set","netns",pid_vm.strip("\n'"),"dev",bk1,"up"])
pid_j = subprocess.check_output(["sudo","docker","inspect","--format","'{{.State.Pid}}'", j])
subprocess.check_output(["sudo","ip","link","set","netns",pid_j.strip("\n'"),"dev",bk2,"up"])
```

c) Create backup bridges and set them up:

```
subprocess.check_output(["sudo","docker","exec","--privileged",j,"brctl","addbr",br_bk])
subprocess.check_output(["sudo","docker","exec","--privileged",j,"ip","link","set", "dev", br_bk,"up"])
```

d) Add vxlan device to backup bridge:

```
subprocess.check_output(["sudo","docker","exec","--privileged",s_vpc,"brctl","addif",br_bk,s_vx]) #bk
```

2.2 Monitor failures

To enhance the functionality of bridge failure detection, the developer must edit 'monitor.py'
For each tenant, program identifies bridge failures across all VPCs and subnets.
To detect a failure, the value of state is checked as shown below:

```
state=subprocess.check_output(["sudo","docker","exec","--privileged",ns,"cat","/sys/class/net/"+br+"/operstate"])
```

When a failure is detected, another program is run:

```
if(state=="down"):
    subprocess.check_output(["sudo","python","react_fail.py",t_id,ns,sub,br_bk])
```

2.3. Recovery from failure

To enhance the functionality of bridge failure recovery, the developer has to edit 'react_fail.py'.
The recovery code has 3 important steps:

a) Deletion of IP from primary interface of user container (VM):

```
subprocess.check_output(["sudo","docker","exec","--privileged",vm,"ip","addr","del",ipa+"/24","dev",if1])
```

b) Addition of the removed IP to the backup interface of user container (VM):

```
subprocess.check_output(["sudo","docker","exec","--privileged",vm,"ip","addr","add",ipa+"/24","dev",bk1])
```

c) Deletion of the IP on original SVI:

```
subprocess.check_output(["sudo","docker","exec","--privileged",vpc_ns,"ip","addr","del",brip+"/24","dev",br])
```

d) Addition of the original SVI IP to the backup bridge:

subprocess.check_output(["sudo","docker","exec","--privileged",vpc_ns,"ip","addr","add",brip+"/24","dev",br_bk])

e) Addition of default route on the VM:

subprocess.check_output(["sudo","docker","exec","--privileged",vm,"ip","route","add",”default”, “via”,brip])

3. Subnet Migration

To enhance the functionality of subnet migration in the aspect of VM migration
Implementing new feature:

Creating subnet in destination vpc:
vm = j+"-s"+s_ip+"_vm"
if j == do_vpc:
res = subprocess.check_output(["sudo","ip","link","add",if1,"type","veth","peer","name",if2])

pid_vm = subprocess.check_output(["sudo","docker","inspect","--format","'{{.State.Pid}}'", vm])
subprocess.check_output(["sudo","ip","link","set","netns",pid_vm.strip("\n'"),"dev",if1,"up"])

Underlay for new subnet:
if do_v1 in s_vpc and do_vpc in d_vpc: --- This condition can be changed to target VXlan creation on specific desired subnet

subprocess.check_output(["sudo","docker","exec","--privileged",s_vpc,"ip","link","add","name",s_vx,"type","vxlan","id",str(vn_id),"dev",s_gre,"remote",d_gre_ip.split('/')[0],"dstport","4789"])
subprocess.check_output(["sudo","docker","exec","--privileged",s_vpc,"ip","link","set","dev",s_vx,"up"])
subprocess.check_output(["sudo","docker","exec","--privileged",s_vpc,"brctl","addif",br,s_vx])

#To add this additional feature to the existing script
VM migration:

1. To the VPC where the subnet already:

Writing query to get subnet and vpc specific information

SELECT nsi FROM vpc
SELECT * FROM gre_pair WHERE s_vpc = $s_vpc AND d_vpc =$d_vpc

2. To the VPC where the subnet doesn’t exist:

#run the script to create the subnet
#adding interfaces
Res = subprocess.check_output(["sudo","ip","link","add",if1,"type" ,"veth","peer","name", if2)
res = subprocess.check_output(["sudo","ip", "link", "set", if1, "netns", vm])
res = subprocess.check_output(["sudo","ip", "link", "set", if2, "netns", j])

#adding ip address to veth pairs
Res = subprocess.check_output(["sudo","ip","netns","exec",vm,"ip","add r","add",ipa+ "/2 4"," dev",if1])
#res = subprocess.check_output(["sudo","ip","net ns","exec",j,"ip","addr","add",g w,"dev",If2])
#Turning on veth pairs
res = subprocess.check_output(["sudo","ip","netns","exec",vm,"ip","link","set", "dev",if1,"up"])
res = subprocess.check_output(["sudo","ip","netns","exec",j,"ip","link","set","dev" ,if2,"up])

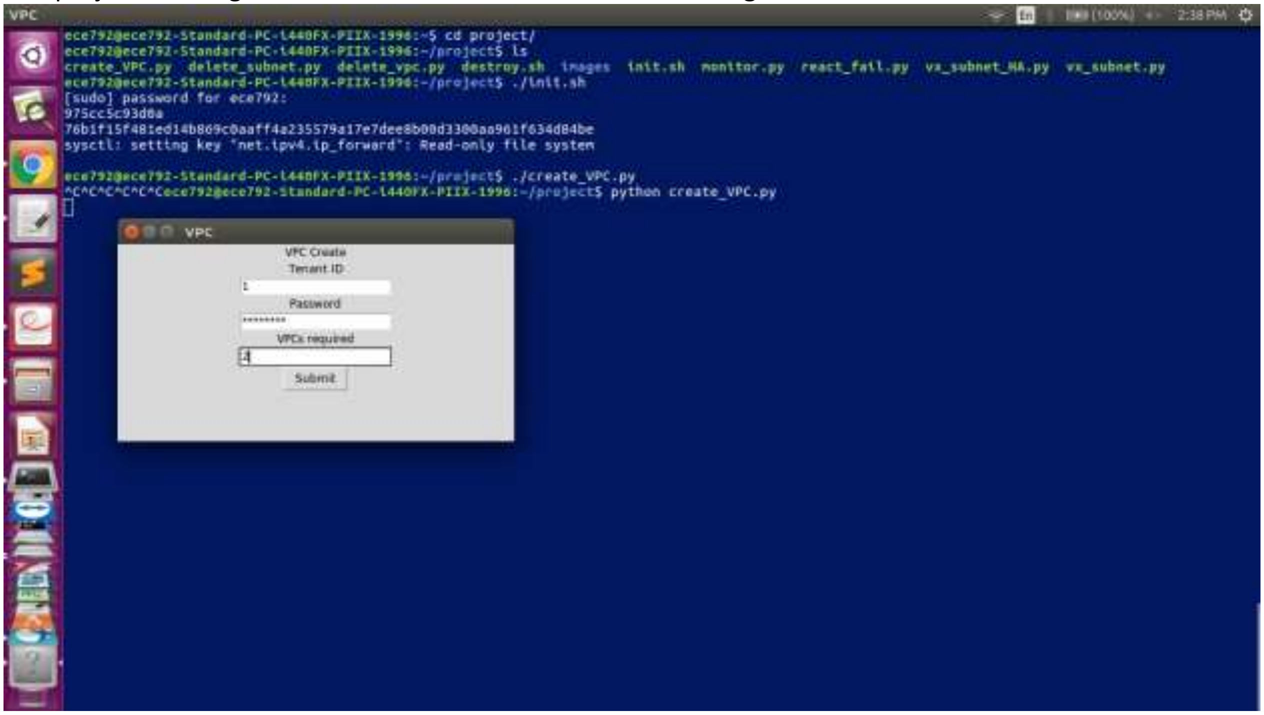
EVALUATION (containerized environment)

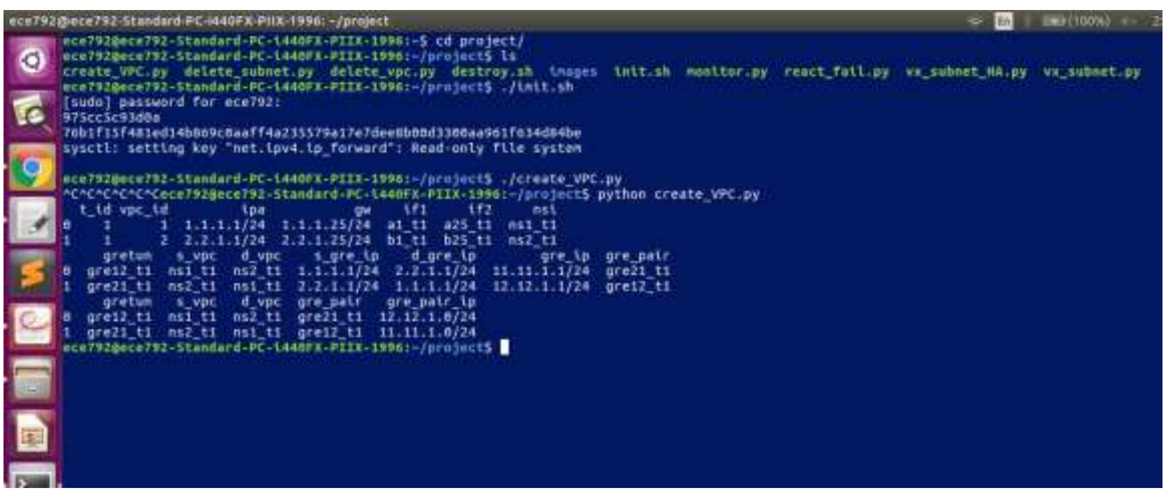
The working topology can be explained by considering all the three possible scenarios or datapaths.

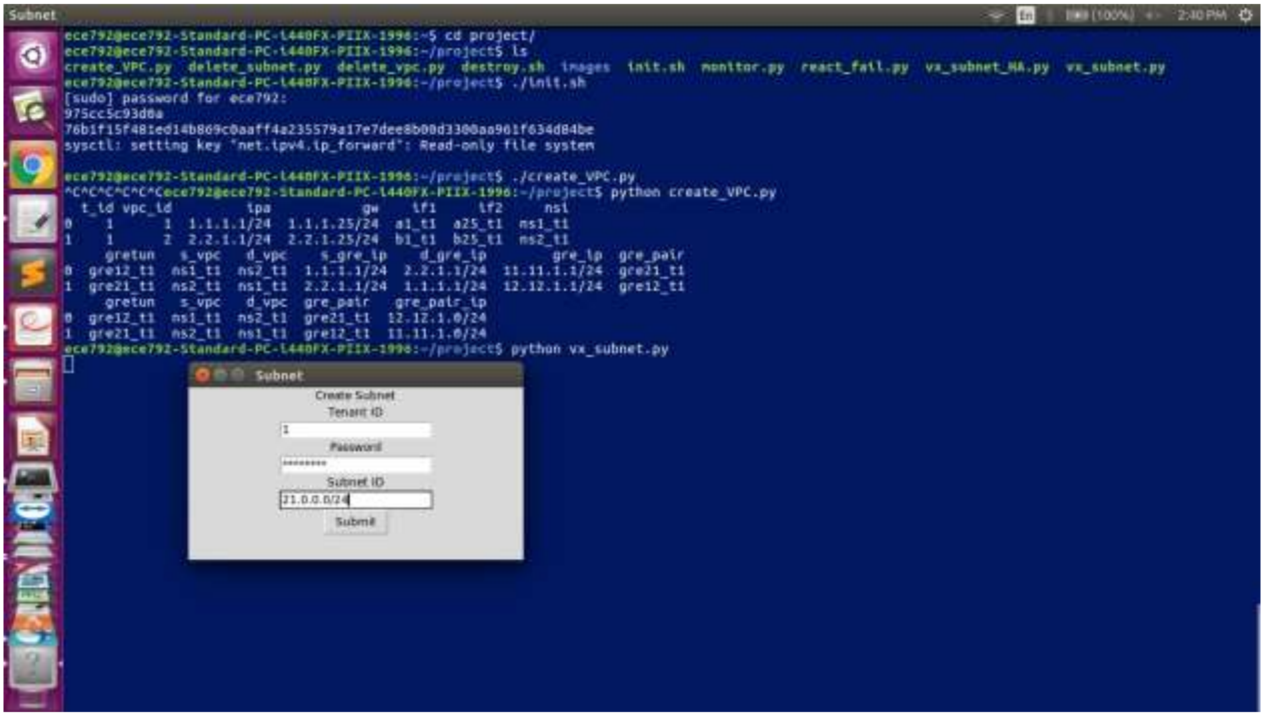
1) Same Subnets in Different VPCs

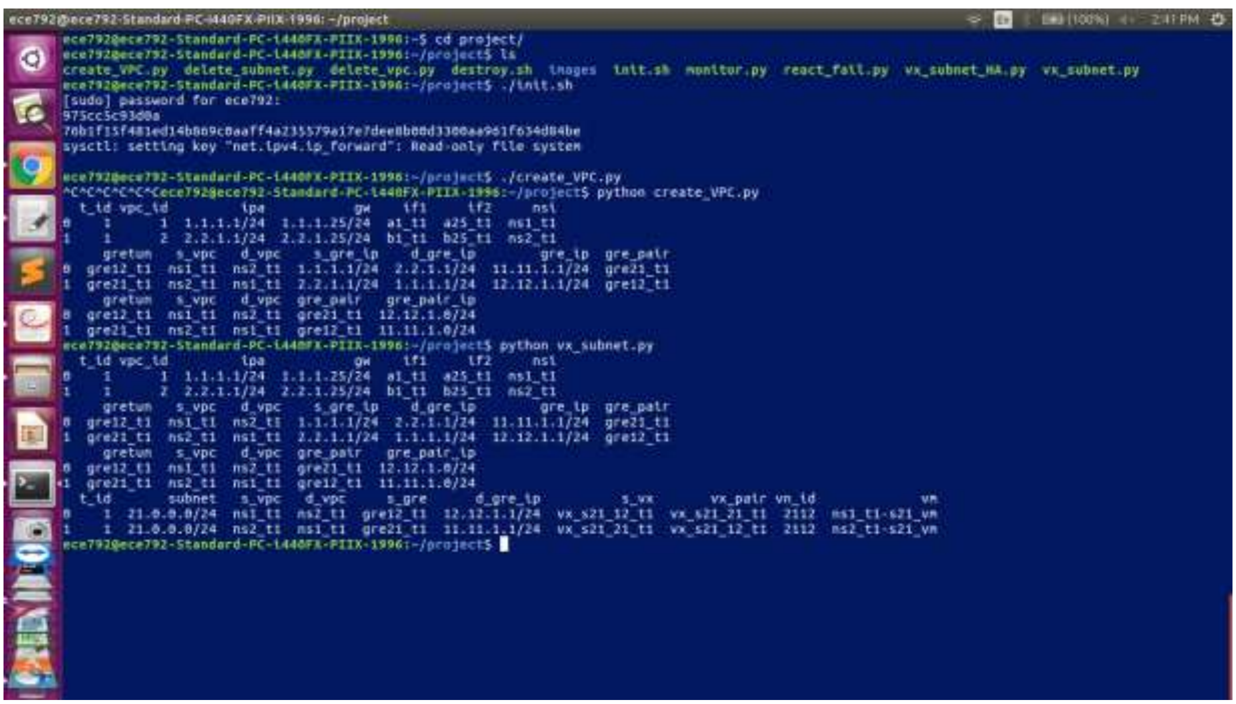
In this scenario user on one Subnet(S1) is communicating with another user on same Subnet(S1) but on different VPC. The data path will include VxLAN and GRE devices.

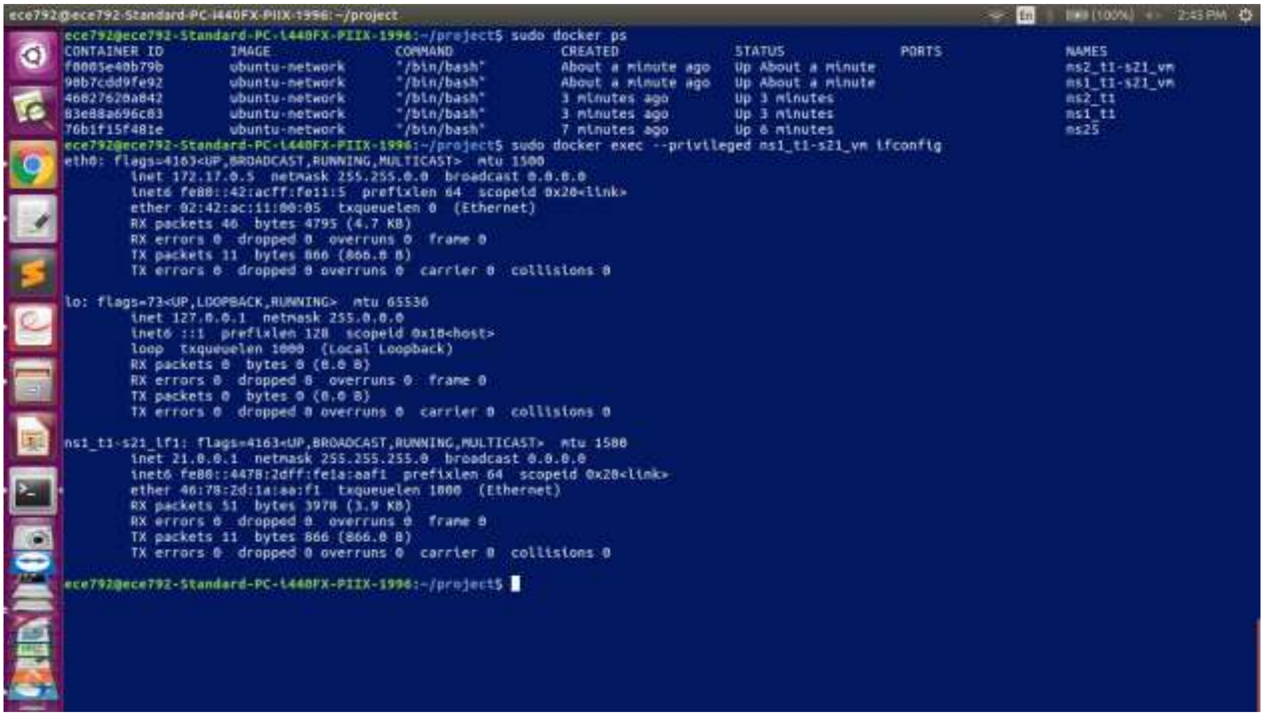
The project working scenario or demo can be understood using the below screenshots.



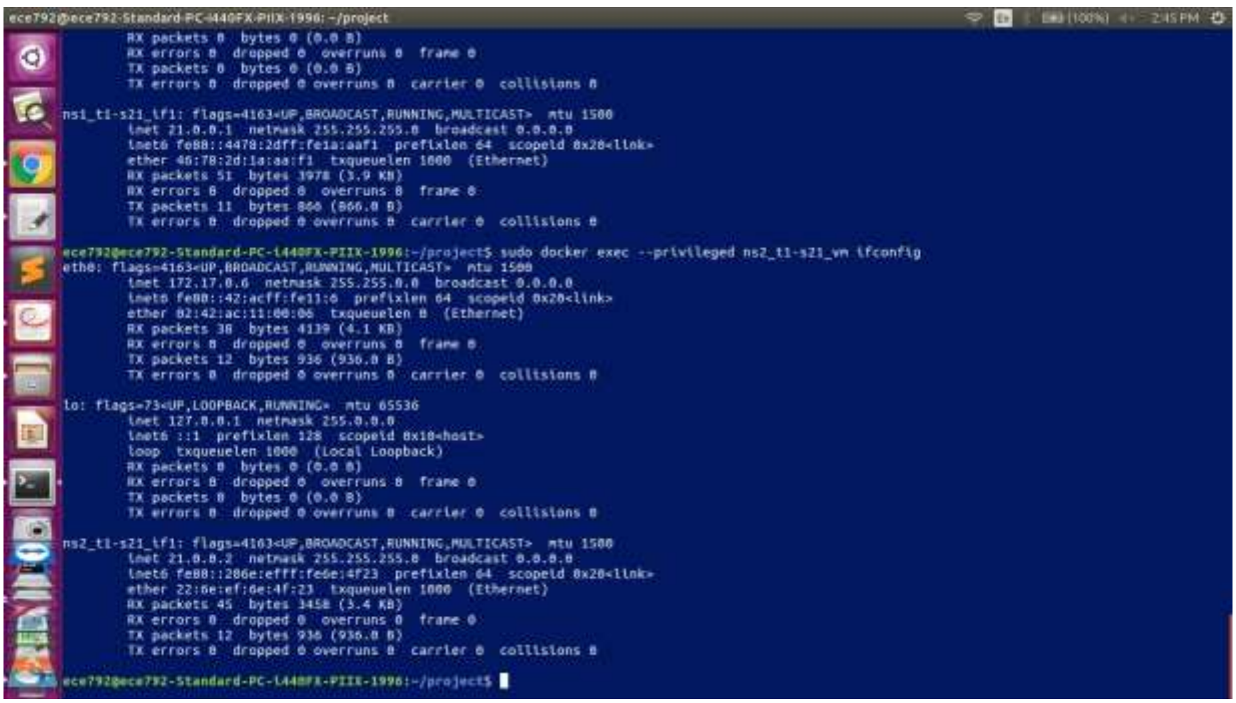



Subnet
ece792@ece792-Standard-PC-i440FX-PIIX-1996:~$ cd project/
ece792@ece792-Standard-PC-i440FX-PIIX-1996:~/project$ ls
create_VPC.py delete_subnet.py delete_vpc.py destroy.sh images init.sh monitor.py react_fail.py vx_subnet_HA.py vx_subnet.py
ece792@ece792-Standard-PC-i440FX-PIIX-1996:~/project$ ./init.sh
[sudo] password for ece792:
sysctl: setting key "net.ipv4.ip_forward": Read-only file system
ece792@ece792-Standard-PC-i440FX-PIIX-1996:~/project$ ./create_VPC.py
ece792@ece792-Standard-PC-i440FX-PIIX-1996:~/project$ python vx_subnet.py
Subnet
Create Subnet
Tenant ID
Password
Subnet ID
Submit

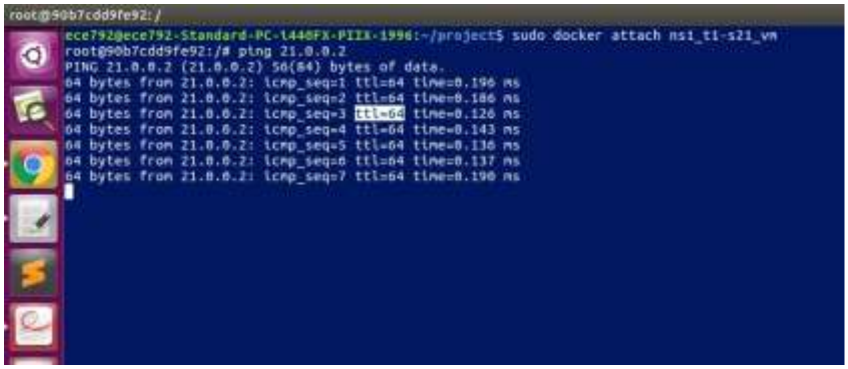

ece792@ece792-Standard-PC-i440FX-PIIX-1996: ~/project
ece792@ece792-Standard-PC-i440FX-PIIX-1996:~$ cd project/
ece792@ece792-Standard-PC-i440FX-PIIX-1996:~/project$ ls
create_VPC.py delete_subnet.py delete_vpc.py destroy.sh images init.sh monitor.py react_fail.py vx_subnet_HA.py vx_subnet.py
ece792@ece792-Standard-PC-i440FX-PIIX-1996:~/project$ ./init.sh
[sudo] password for ece792:
sysctl: setting key "net.ipv4.ip_forward": Read-only file system
ece792@ece792-Standard-PC-i440FX-PIIX-1996:~/project$ ./create_VPC.py
ece792@ece792-Standard-PC-i440FX-PIIX-1996:~/project$ python vx_subnet.py
ece792@ece792-Standard-PC-i440FX-PIIX-1996:~/project$

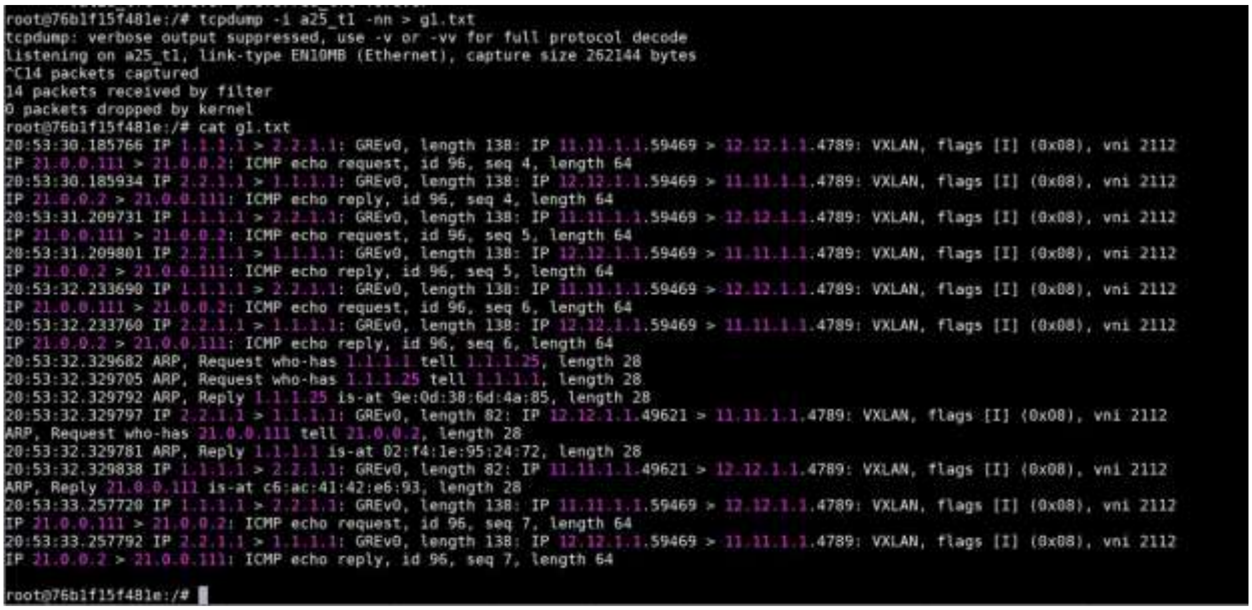


ece792@ece792-Standard-PC-i440FX-PIIX-1996: ~/project
ece792@ece792-Standard-PC-i440FX-PIIX-1996:~/project$ sudo docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
f0005e40b79b ubuntu-network "/bin/bash" About a minute ago Up About a minute ns2_t1-s21_vm
90b7cdd9fe92 ubuntu-network "/bin/bash" About a minute ago Up About a minute ns1_t1-s21_vm
46027620a842 ubuntu-network "/bin/bash" 3 minutes ago Up 3 minutes ns2_t1
83e88a696c03 ubuntu-network "/bin/bash" 3 minutes ago Up 3 minutes ns1_t1
76b1f15f481e ubuntu-network "/bin/bash" 7 minutes ago Up 6 minutes ns25
ece792@ece792-Standard-PC-i440FX-PIIX-1996:~/project$ sudo docker exec --privileged ns1_t1-s21_vm ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
inet 172.17.0.5 netmask 255.255.0.0 broadcast 0.0.0.0
inet6 fe80::42:acff:fe11:5 prefixlen 64 scopeid 0x20<link>
ether 02:42:ac:11:00:05 txqueuelen 0 (Ethernet)
RX packets 40 bytes 4795 (4.7 KB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 11 bytes 866 (866.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
inet 127.0.0.1 netmask 255.0.0.0
inet6 ::1 prefixlen 128 scopeid 0x10<host>
loop txqueuelen 1000 (Local Loopback)
RX packets 0 bytes 0 (0.0 B)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 0 bytes 0 (0.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
ns1_t1-s21_lf1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
inet 21.0.0.1 netmask 255.255.255.0 broadcast 0.0.0.0
inet6 fe80::4478:2dff:fe1a:aaf1 prefixlen 64 scopeid 0x20<link>
ether 46:78:2d:1a:aa:f1 txqueuelen 1000 (Ethernet)
RX packets 51 bytes 3978 (3.9 KB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 11 bytes 866 (866.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
ece792@ece792-Standard-PC-i440FX-PIIX-1996:~/project$



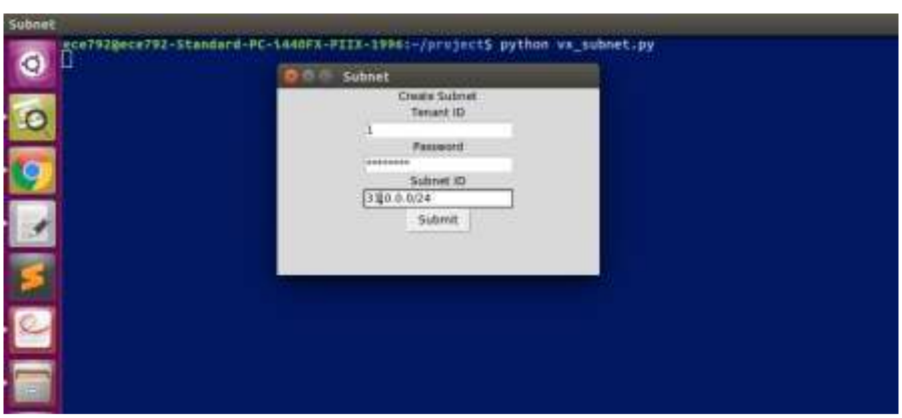
ece792@ece792-Standard-PC-i440FX-PIIX-1996: ~/project
RX packets 0 bytes 0 (0.0 B)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 0 bytes 0 (0.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
ns1_t1-s21_lf1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
inet 21.0.0.1 netmask 255.255.255.0 broadcast 0.0.0.0
inet6 fe80::4478:2dff:fe1a:aaf1 prefixlen 64 scopeid 0x20<link>
ether 46:78:2d:1a:aa:f1 txqueuelen 1000 (Ethernet)
RX packets 51 bytes 3978 (3.9 KB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 11 bytes 866 (866.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
ece792@ece792-Standard-PC-i440FX-PIIX-1996:~/project$ sudo docker exec --privileged ns2_t1-s21_vm ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
inet 172.17.0.6 netmask 255.255.0.0 broadcast 0.0.0.0
inet6 fe80::42:acff:fe11:6 prefixlen 64 scopeid 0x20<link>
ether 02:42:ac:11:00:06 txqueuelen 0 (Ethernet)
RX packets 38 bytes 4139 (4.1 KB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 12 bytes 936 (936.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
inet 127.0.0.1 netmask 255.0.0.0
inet6 ::1 prefixlen 128 scopeid 0x10<host>
loop txqueuelen 1000 (Local Loopback)
RX packets 0 bytes 0 (0.0 B)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 0 bytes 0 (0.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
ns2_t1-s21_lf1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
inet 21.0.0.2 netmask 255.255.255.0 broadcast 0.0.0.0
inet6 fe80::206e:efff:fe6e:4f23 prefixlen 64 scopeid 0x20<link>
ether 22:6e:ef:6e:4f:23 txqueuelen 1000 (Ethernet)
RX packets 45 bytes 3458 (3.4 KB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 12 bytes 936 (936.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
ece792@ece792-Standard-PC-i440FX-PIIX-1996:~/project$

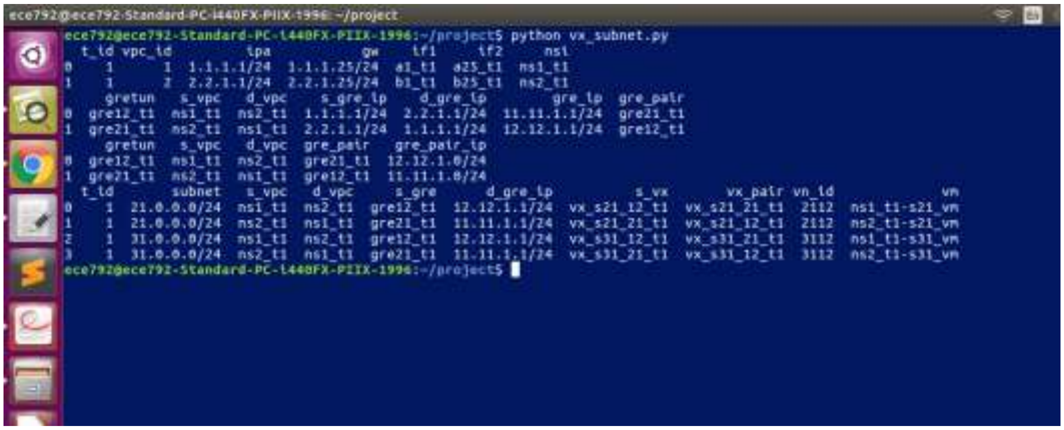


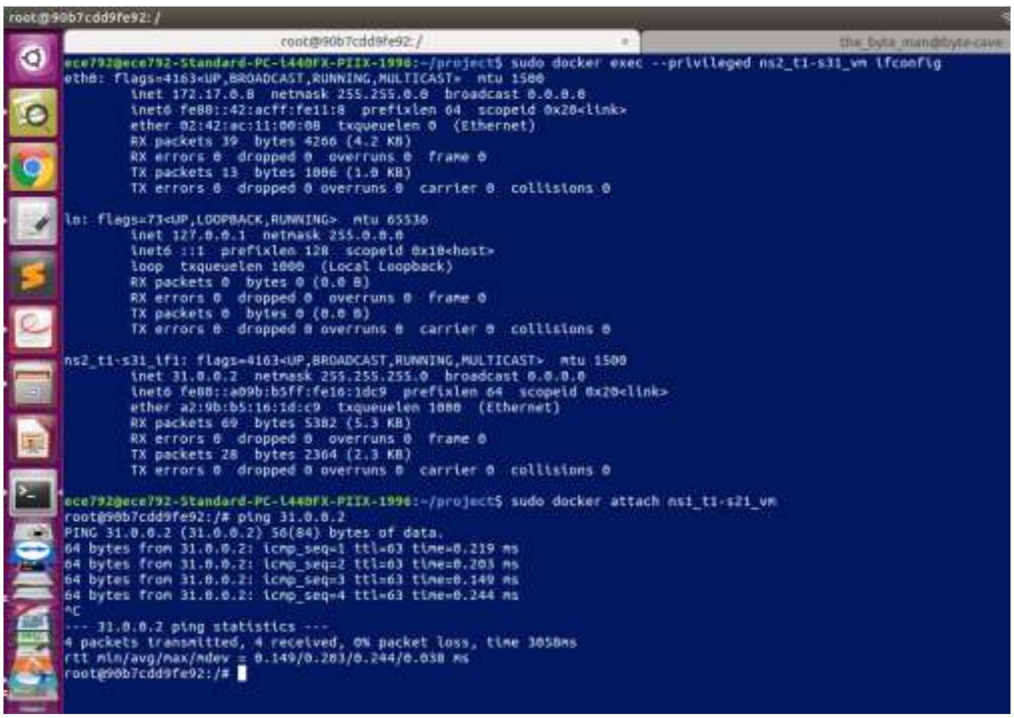


2) Different Subnet in Different VPC

In this scenario user on Subnet(S1) is communicating with another user on Subnet(S2) in a different VPC. The data path will include the Gre tunnel device.

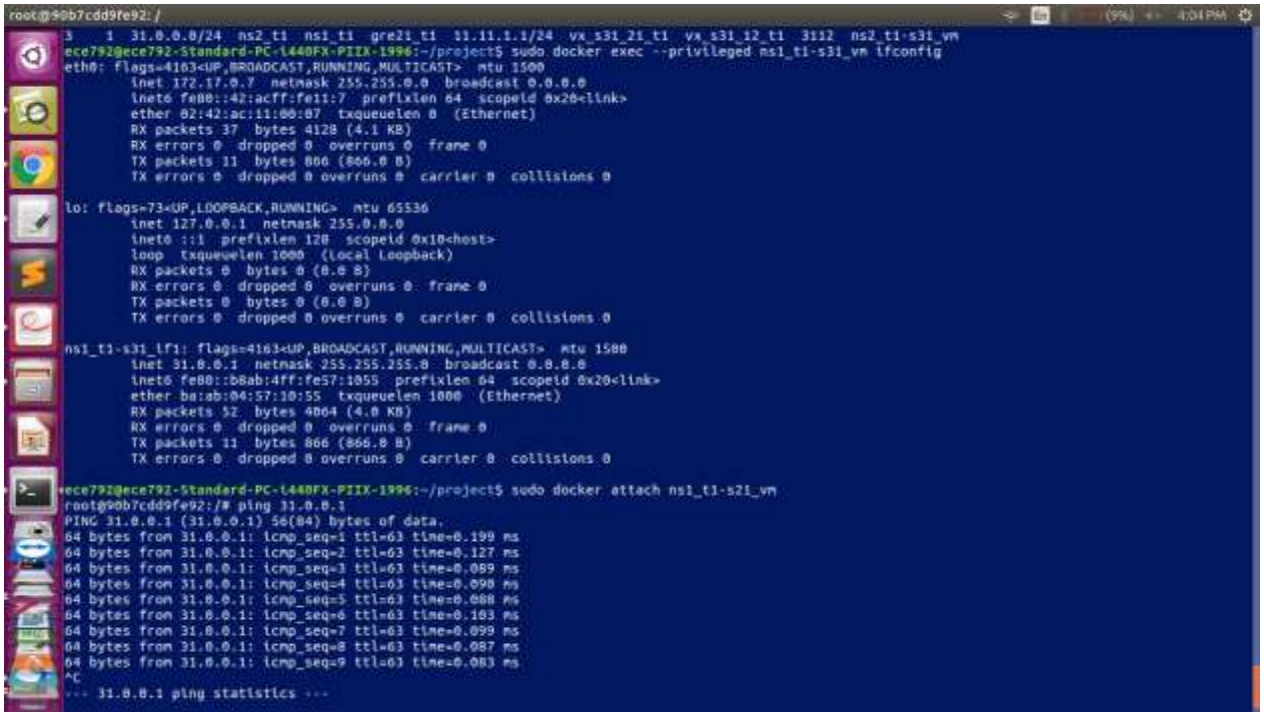




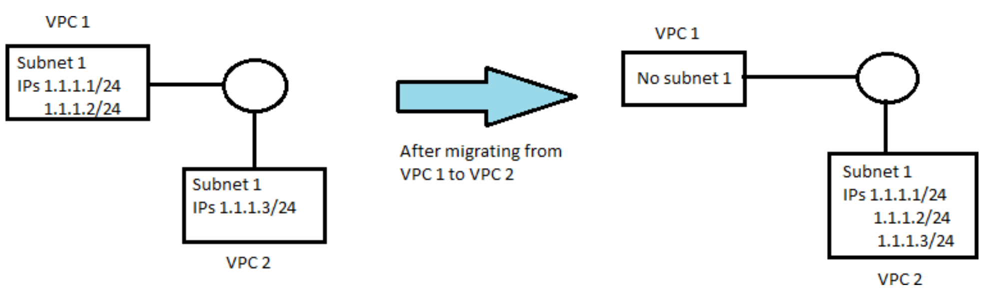


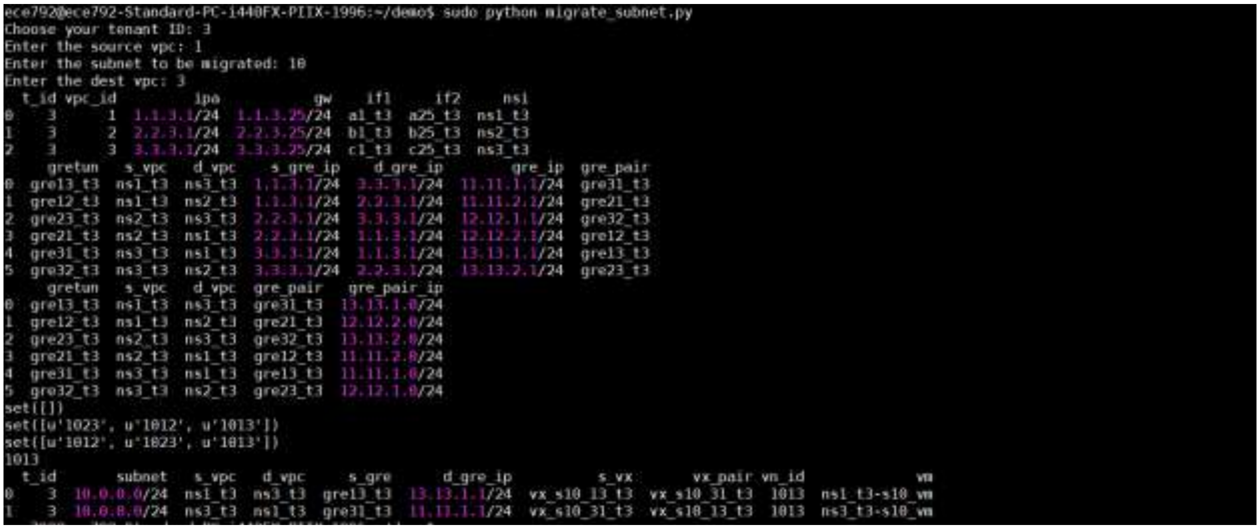
3) <u>Different Subnet on Same VPC</u>

In this scenario user on Subnet(S1) is communicating with user on Subnet (2) on the same VPC. The data path will include the Bridge SVI (L3) in this case.



Case for Migration of Subnets





```
ece792@ece792-Standard-PC-i440FX-PIIX-1996:~/demo$ sudo python migrate_subnet.py
Choose your tenant ID: 3
Enter the source vpc: 1
Enter the subnet to be migrated: 10
Enter the dest vpc: 3
  t_id vpc_id          ipa           gw     if1     if2     nsi
0    3      1  1.1.3.1/24  1.1.3.25/24  a1_t3  a25_t3  ns1_t3
1    3      2  2.2.3.1/24  2.2.3.25/24  b1_t3  b25_t3  ns2_t3
2    3      3  3.3.3.1/24  3.3.3.25/24  c1_t3  c25_t3  ns3_t3
     gretun   s_vpc   d_vpc    s_gre_ip    d_gre_ip        gre_ip  gre_pair
0  gre13_t3  ns1_t3  ns3_t3  1.1.3.1/24  3.3.3.1/24  11.11.1.1/24  gre31_t3
1  gre12_t3  ns1_t3  ns2_t3  1.1.3.1/24  2.2.3.1/24  11.11.2.1/24  gre21_t3
2  gre23_t3  ns2_t3  ns3_t3  2.2.3.1/24  3.3.3.1/24  12.12.1.1/24  gre32_t3
3  gre21_t3  ns2_t3  ns1_t3  2.2.3.1/24  1.1.3.1/24  12.12.2.1/24  gre12_t3
4  gre31_t3  ns3_t3  ns1_t3  3.3.3.1/24  1.1.3.1/24  13.13.1.1/24  gre13_t3
5  gre32_t3  ns3_t3  ns2_t3  3.3.3.1/24  2.2.3.1/24  13.13.2.1/24  gre23_t3
     gretun   s_vpc   d_vpc  gre_pair   gre_pair_ip
0  gre13_t3  ns1_t3  ns3_t3  gre31_t3  13.13.1.0/24
1  gre12_t3  ns1_t3  ns2_t3  gre21_t3  12.12.2.0/24
2  gre23_t3  ns2_t3  ns3_t3  gre32_t3  13.13.2.0/24
3  gre21_t3  ns2_t3  ns1_t3  gre12_t3  11.11.2.0/24
4  gre31_t3  ns3_t3  ns1_t3  gre13_t3  11.11.1.0/24
5  gre32_t3  ns3_t3  ns2_t3  gre23_t3  12.12.1.0/24
set([])
set([u'1023', u'1012', u'1013'])
set([u'1012', u'1023', u'1013'])
1013
  t_id        subnet   s_vpc   d_vpc     s_gre      d_gre_ip          s_vx       vx_pair vn_id              vm
0    3  10.0.0.0/24  ns1_t3  ns3_t3  gre13_t3  13.13.1.1/24  vx_s10_13_t3  vx_s10_31_t3  1013  ns1_t3-s10_vm
1    3  10.0.0.0/24  ns3_t3  ns1_t3  gre31_t3  11.11.1.1/24  vx_s10_31_t3  vx_s10_13_t3  1013  ns3_t3-s10_vm
```

Future scope:

Overlay networks are a front-runner in the network virtualization space. Overlays enable communication between private networks over a public network. There is total address space isolation. Once proper encapsulation is done, underlay and overlay become independent of each other. This feature is extremely beneficial for big firms who have multiple VPCs in different locations, but still wish to operate them as if they are directly connected in network.

But an actual industrial solution needs flexibility in the number of hypervisors as well. So, it can be such that a tenant's presence is in multiple hypervisors and needs to communicate cross-hypervisors to reach his subnets.

For this scenario, we can have a Kubernetes-based solution. Kubernetes helps in deploying clusters which have a master and several nodes. Based on the user's preference, one hypervisor can be made the master and the rest as worker nodes. Each node has the facility to host pods. Each pod can have single or multiple containers. Now we can proceed in a similar fashion as the current containerized approach.