

A Multi-Node Satellite System using Docker and Open vSwitch

Xuzhi Zhang

Ph.D. Candidate

Department of Electrical and Computer Engineering

University of Massachusetts Amherst

xuzhizhang@umass.edu

Narendra Prabhu

Graduate Student

Department of Electrical and Computer Engineering

University of Massachusetts Amherst

nprabhu@umass.edu

Reconfigurable Computing Group

July 10, 2019

Contents

Introduction	4
Design	6
Structure	7
Prerequisites	10
Installation and usage instructions	11
Install VirtualBox.....	11
Import Virtual Appliance	11
Using the Image	13
Connecting to SimulatorVM Image	16
Docker and OpenvSwitch Status	17
Starting the Simulator	21
Create a 4-node Simulator.....	21
Check Docker Containers.....	23
Check Interfaces	23
Check OpenvSwitch Configuration	24
Clearing the 4-node simulator	25
Scripts Overview	26
Internal Scripts.....	28
Running the Simulator	30
Running 4-node simulation	30
JSON-type graphs.....	36
Running 32-node simulation	37
Time slot and Varying simulation time.....	39

Output affirmation through log files	39
Further Usage	43
References	46

Introduction

The aim of this guide is to document the setup of an environment that can simulate a scalable network of satellite components and switches. For this experimentation, a satellite is formed from multiple components. A switch is used to interconnect these components and multiple satellites are combined together in the simulation via an additional switch. Docker [1], a container-based virtualization platform and Open vSwitch (OVS) [2], a virtualized switch, are used to generate this scalable simulation environment and ensure isolation between components and satellites. The environment will be used to simulate a mesh network of satellites/GPS systems connected to work in tandem. Docker containers are used to represent each satellite and OVS is used to interconnect them to form a communication network. Each satellite encloses multiple nested Docker containers designated to act as components and interconnected via OVS.

The simulation environment can be used to evaluate a scalable collection of Global Network Access Terminals (GNATs), a concept under investigation as part of the Space Combat Cloud Project [3]. The components in each satellite can be described as follows. The GNAT Brain (or GNAT) is the “brain” of the satellite. It controls satellite operation and configures routing tables. PHY devices are typically communication components of the satellite which act as the interfaces to other satellites in the network. The type of PHYs can vary according to its communication mode. For example, an RF-PHY connects to a ground station. An MSGEN is a message generator component inside the satellite that is responsible for the generation of messages that need to be sent to other satellites. An MSGEN PHY does not directly communicate with another satellite. The Ground Station is another type of a satellite but includes only RF-PHY, used to mimic a hardware station based in a physical location. A single satellite from the satellite node system is connected to the RF-PHY of this base station to receive commands about the traffic flow.

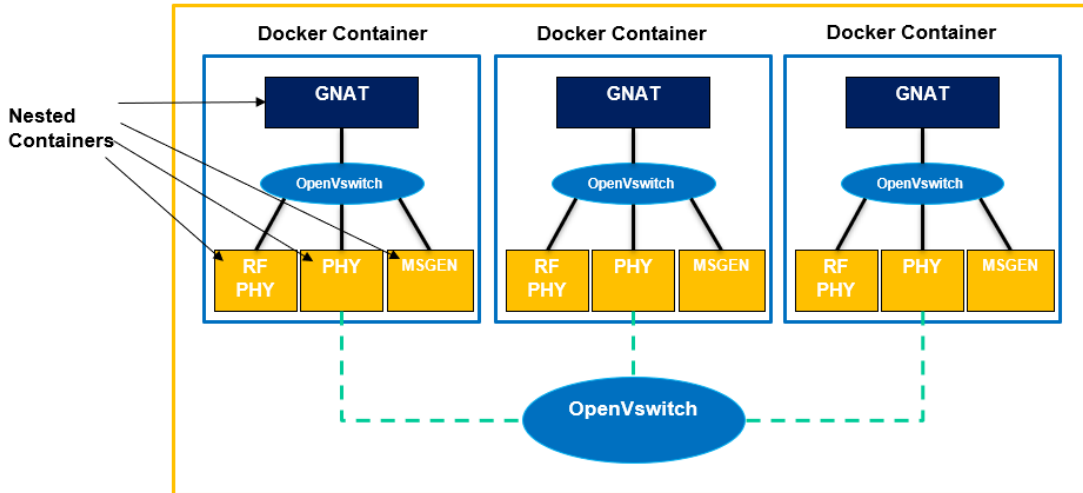


Figure 1: A 3 node system depicting the structure of the simulator

In a physical satellite network, PHYs communicate with each other directly (PHY-to-PHY). In the simulation environment, PHY-to-PHY connections are passed through a configurable OVS switch to facilitate rapid network reconfiguration. As shown in Figure 2, the links between the satellites are the responsibility of the Master Controller (MC), a control process that manages the simulation. The MC performs a series of actions including parsing visibility information between satellites (a JSON file), parsing traffic flow information, allowing simulation to proceed in time slots, and creating links between PHYs on distinct satellites so they can communicate with each other. The master controller process is written in python code developed locally at UMass, while the code for applications within the satellite i.e. the GNAT, PHYs and MSGEN is provided by Don Fronterhouse from PnP Innovations.

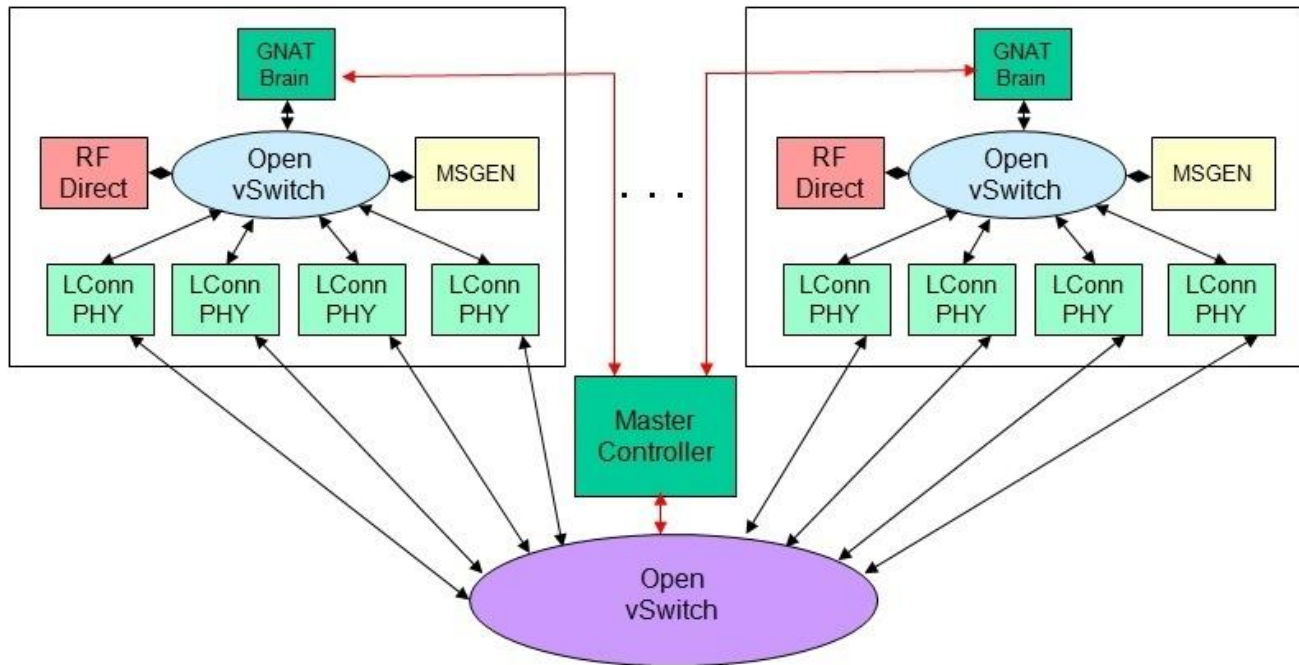


Figure 2: Illustrates the links between the components of a simple 2 satellite simulator

Design

The remainder of this document describes the steps related to building a scalable multi-node system. To support the creation of a network with nested containers, Open vSwitch and Docker need to be installed within a Docker container. This leads to the need for a nested container. Such functionality is supported by a ‘dind’ image provided by Docker [4], which allows for the nesting of one Docker container within another Docker container.

Linux-provided virtual Ethernet (Veth) connections (devices) are used to create inter- and intra-satellite connections. To provide isolation and accessibility to each Docker container we use network namespaces. A namespace provides a replica of the parent network stack but with its own routes and network devices. The Veth devices can act as tunnels between these network namespaces to create a bridge to a physical/virtual network device in another namespace. We place one end of a Veth pair in one network namespace and the other end in another network namespace, thus allowing communication between network namespaces. For intra-satellite communication, the GNAT, PHY, MSGEN and RFPHY containers need to be connected by individual Veth pairs to an Open vSwitch bridge. Each end of the Veth pair is added as a port on the bridge, thus ensuring that communication occurs over the bridge. The other ends are inserted into the network namespaces corresponding to each container. Apart from the veth interfaces available, we also use a MACVLAN type interface. With MACVLAN, we can create multiple interfaces with different Layer 2 (that is, Ethernet MAC) addresses on top of a single one. This allows isolation of the data and control planes of communication in the satellite.

The process ID (PID) of a Docker container changes every time a container is restarted, posing a challenge. As a result, the PID and container name are fixed to set a namespace. The PID namespace provides a separation of processes, removes the view of the system processes, and allows process IDs to be isolated. This step is required so that commands for the nested container can be run in the namespace from an external host. Hence, the PID and container name can be set using a symbolic link (also known as a soft link or symlink), a file that serves as a reference to another file or directory. This ensures the seclusion of each Docker container and enables access to the nested Docker containers from the host, which is essential to running the scripts on the host. Basically, each container can now be identified by its name that is linked with its PID. Inter-satellite communication is set up using the same method. Each PHY is connected to an external Open vSwitch using Veth links.

Structure

Figure 3 shows the basic internal satellite interfaces for a satellite. Each black line is a Veth link. The blue boxes signify the nested containers connected to an Open vSwitch bridge (in beige). A satellite with a PHY, an RFPHY, a GNAT and an MSGEN is shown. The GNAT communicates with its intersatellite container peers using one interface, the main physical interface “eth0” and its MACVLAN child “gnat”. MACVLAN allows us to host another interface with a different MAC address on the eth0 interface. The PHYs have two physical interfaces “eth0” and “eth1” and two virtual interfaces, “direct.1d”, “direct.1c”. The latter two are control and data MACVLAN interfaces which are employed to send and receive control and data messages from the GNAT. The control messages exchange the commands or rules for implementation while the data interfaces exchange data during the interaction. The eth1 interface is an outward facing interface as it provides the link to communicate to other PHY’s from a different satellite. The MSGEN has similar set of interfaces, although the “mgen.d” sends data outwards to the PHY’s and “mgen.c” exchanges control messages with the GNAT. The “eth0” and “eth1” interfaces are the main physical interfaces used to exchange initial communication messages. Depending on the components inside the satellite, we add MACVLAN interfaces on top of these host interfaces to allow for separation of communicating control and data messages to different interfaces. Each interface is assigned a different MAC address.

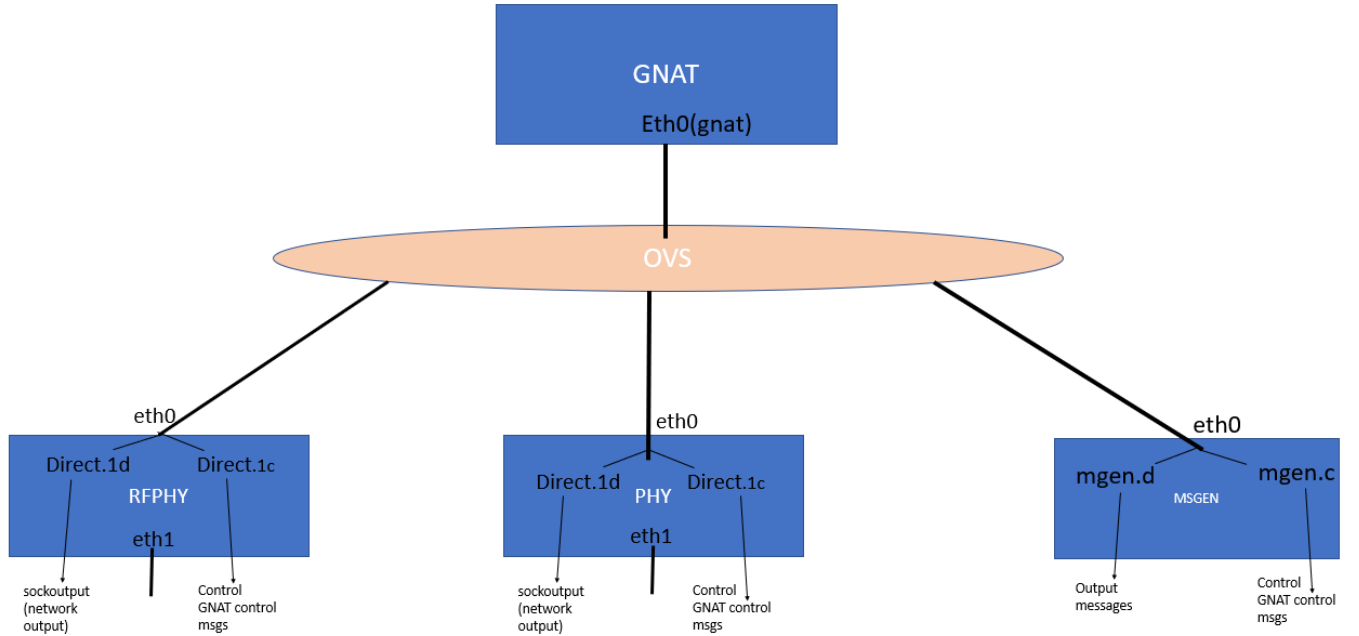


Figure 3: Interfaces inside a satellite node in the simulator

One satellite in the emulation environment, Satellite Node 1, can include additional components that are not shown in Figure 3, the Master Resinate Controller (MRC) and a Portal PHY (portphy). The MRC communicates with the Master Controller process to publish the simulation information using a channel to a backend satellite constellation imaging tool from Intelligent Fusion Technology, Inc (IFT). The Portal PHY is used to send and receive data from a physical ground station based in Albuquerque, NM. The ground station is part of the physical RESINATE testbed. None of these components (MRC, portal, backend, or RESINATE) are needed to perform the simulations described in this document. Figure 4 shows the overall connectivity for a multi-node satellite system without a ground station, MRC or Portal PHY:

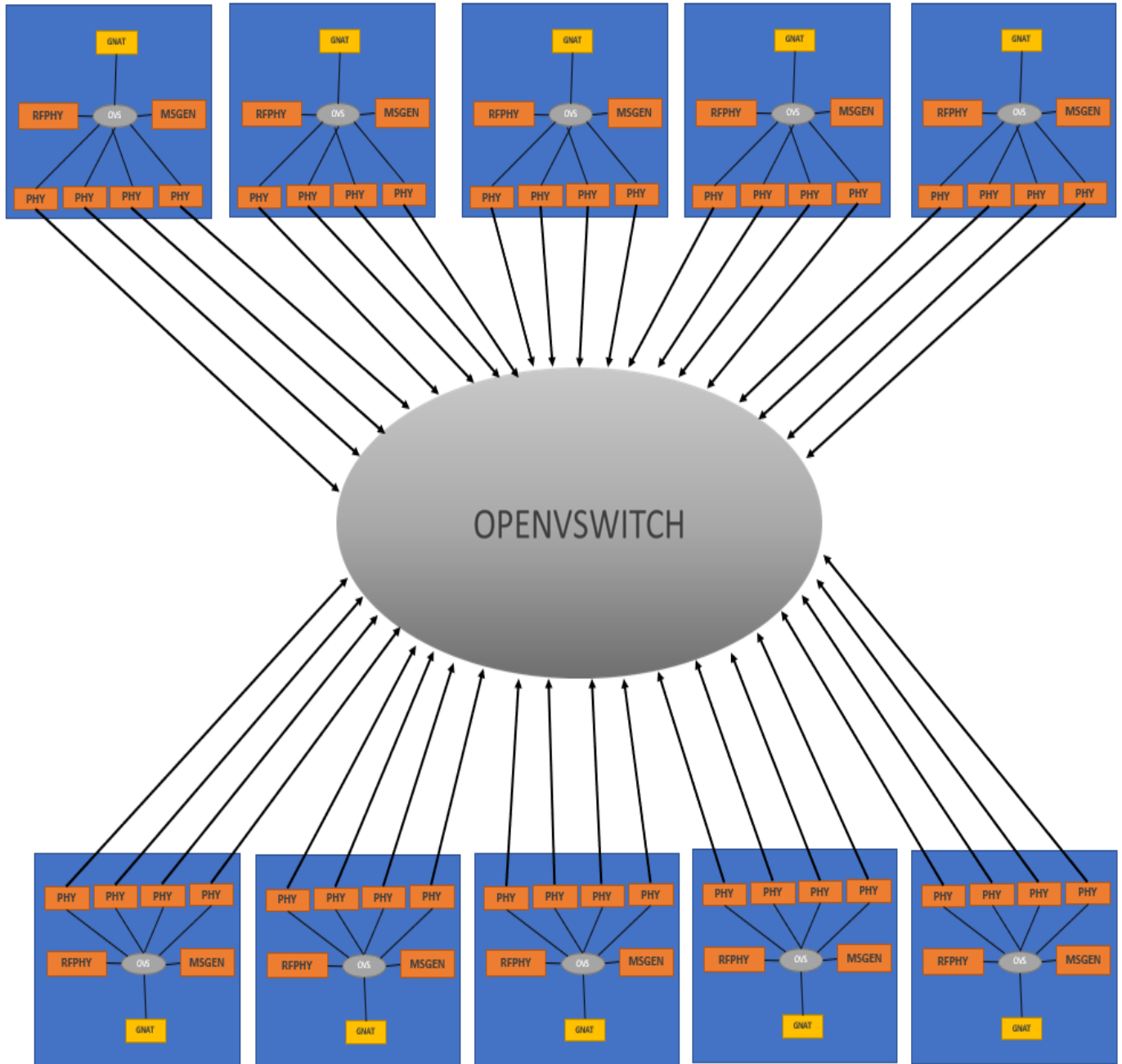


Figure 4: An example system that shows intersatellite links

Figure 5 shows the MRC, and its communication flow with the master controller. Figure 6 portrays how the Portal PHY is connected to the ground station hardware. Note: The portal, MRC, front end and back end are not needed for completely software simulation..

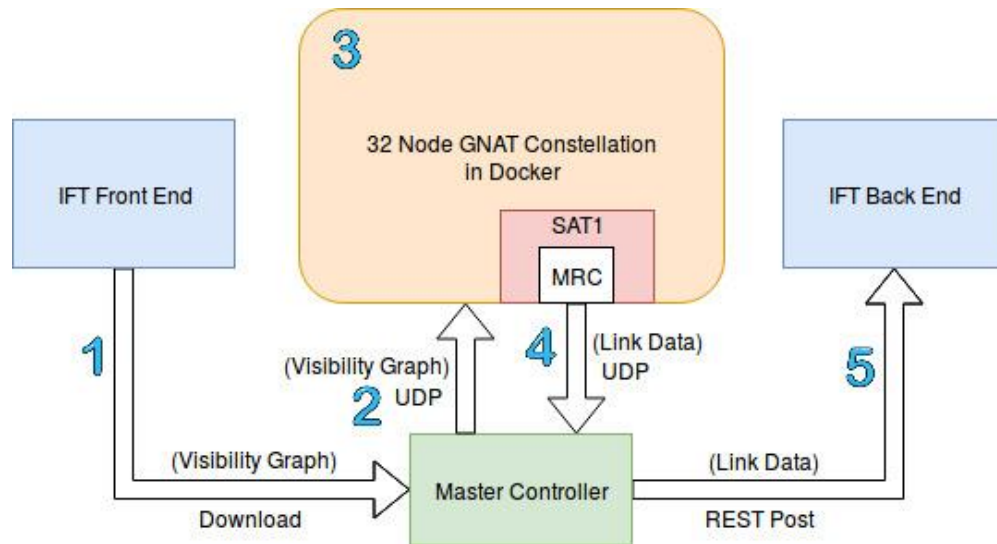


Figure 5: MRC integration in Satellite 1. The visibility graph distribution is made from the master controller to the 32 individual GNAT brains

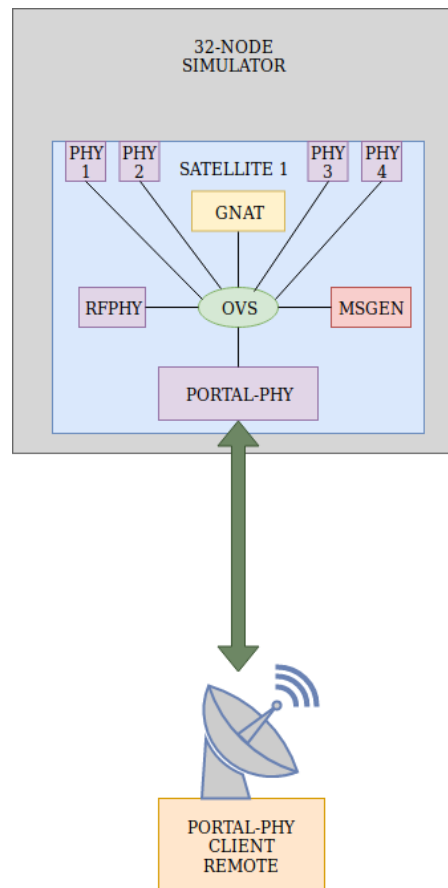


Figure 6: Portal PHY integration in Satellite 1

Prerequisites

An Ubuntu 16.04 operating system is necessary to run the simulator, as the newest version does not support the Linux kernel modules and Linux-headers dependencies needed for the Open vSwitch installation. Thus, the VirtualBox image provided is built using the Ubuntu 16.04 version. The kernel version used was 4.4.0.138. Older kernel versions might add arbitrary padding to UDP raw packets and cause data communication checksum issues. Each Docker container in our setup takes up 1.2GB of disk space along with an associated volume of 2GB. Each container consumes about 120MB of memory and an OVS instance takes up around 80 MB of memory while running. The scripts in the VM directory generate the network simulator, with Docker and Open vSwitch which have been installed in the virtual machine. The user must install VirtualBox on the system [5] to import the appliance, i.e. the VirtualBox image provided, and login through either a GUI or SSH using port forwarding. This issue is described in detail subsequently.

Installation and usage instructions

In this guide, we will provide instructions on setting up the network simulator inside the VirtualBox image and provide hints to customize the simulator as the user requires.

Install VirtualBox

Begin with downloading and installing the latest VirtualBox image [6] using the package suitable for your operating system (i.e. Windows/Linux/OS X). Once installed, run the VirtualBox. To run VirtualBox, open a terminal on your system and run the command **VirtualBox** as shown:

```
~]$ virtualbox
```

A GUI should pop up as shown in Figure 7.

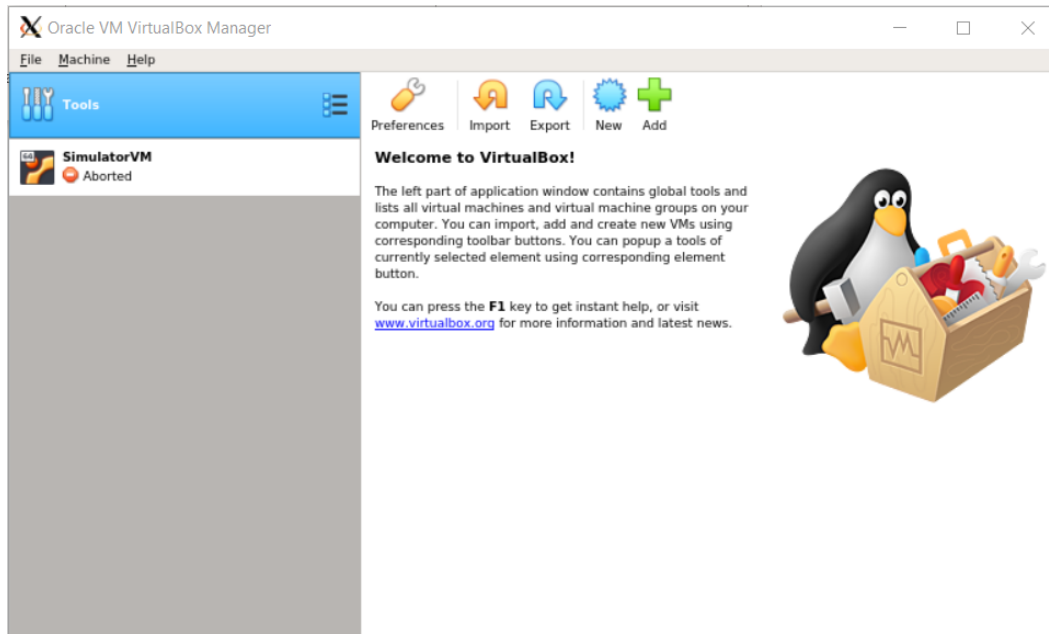


Figure 7: VirtualBox Manager screen

Importing the VirtualBox appliance:

The VirtualBox image provided i.e. SimulatorVM.Ova file can be downloaded to any directory you wish. This file is then imported into VirtualBox. In the VirtualBox GUI, Go to file-> Import appliance.

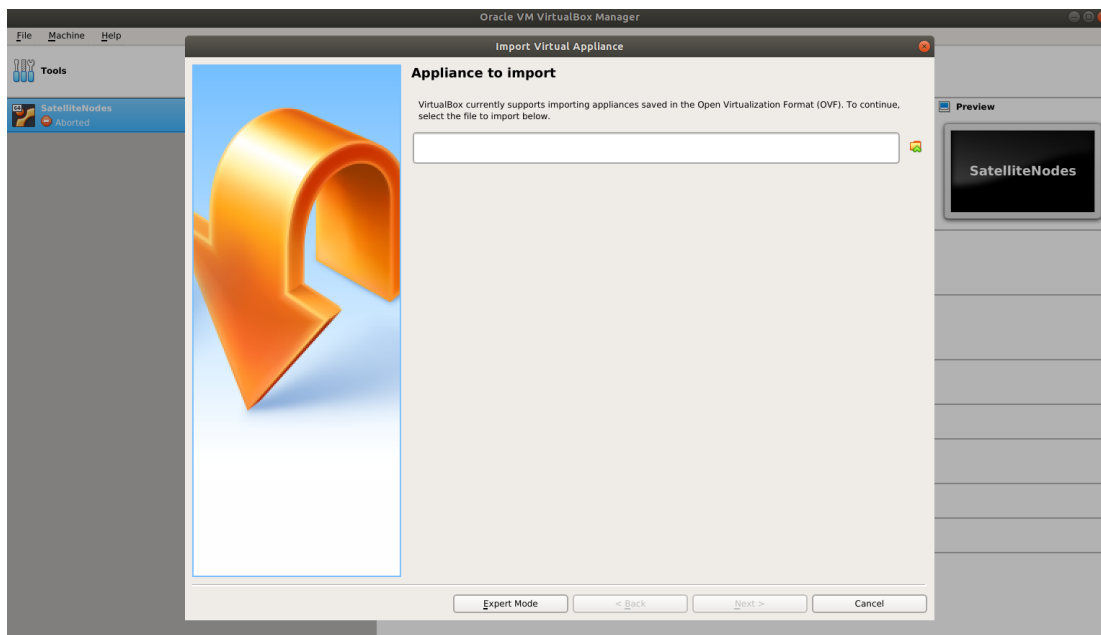


Figure 8: Appliance import screen

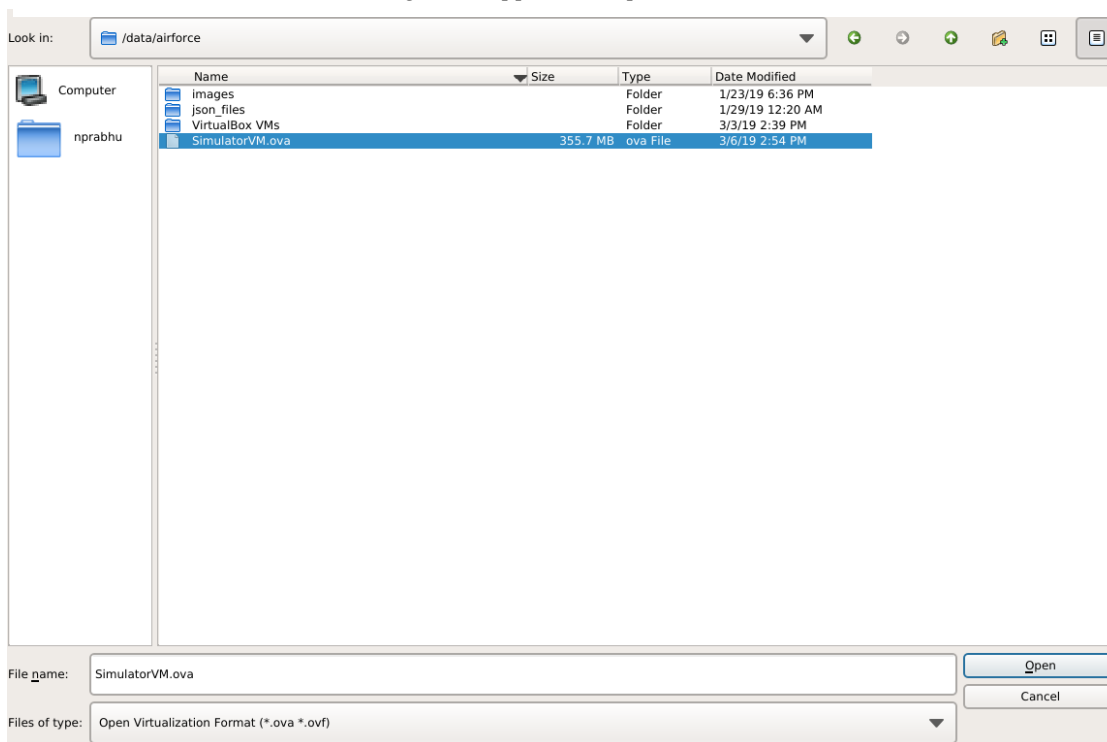


Figure 9: Loading the SimulatorVM.ova file

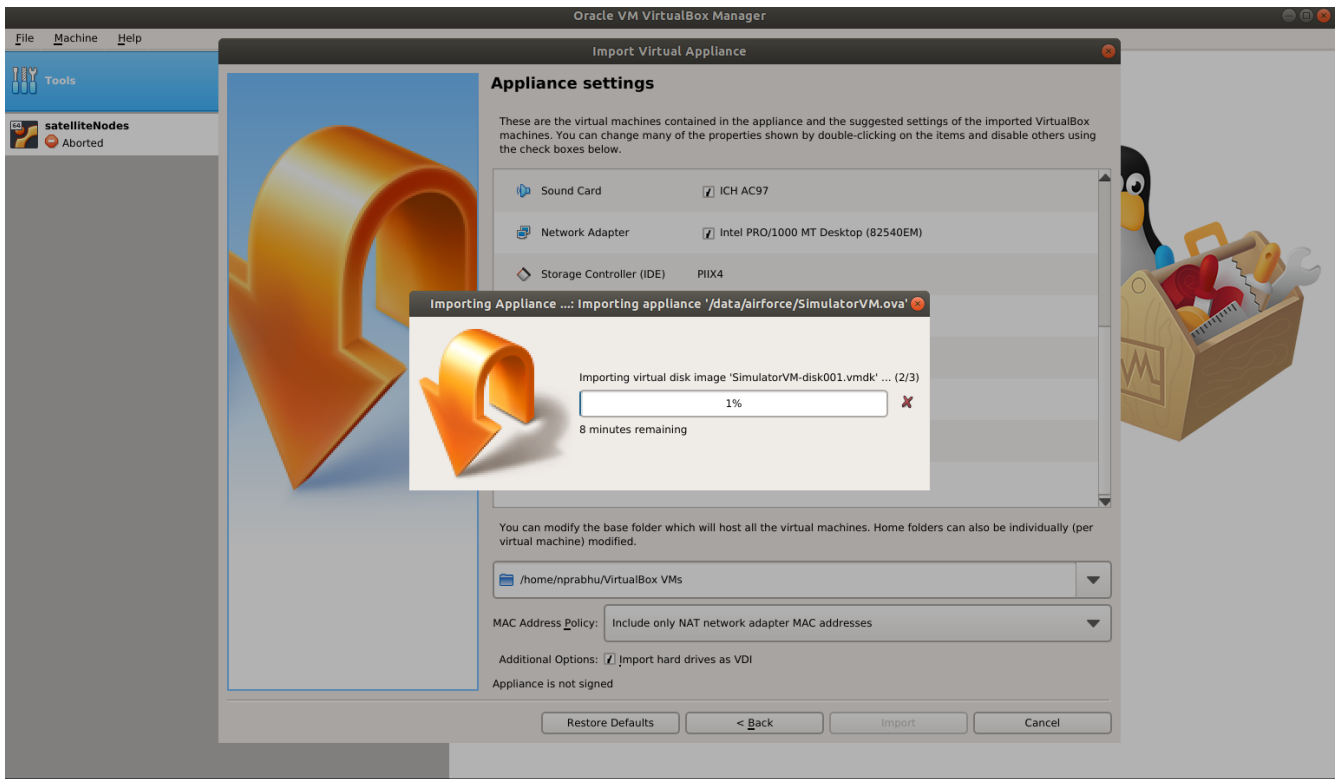


Figure 10: Importing the VM with configurations

Load the virtual machine image provided, SimulatorVM.Ova from the directory where you downloaded it to start up the VM.

Using the image:

Once the VM is loaded, it will show up in the GUI with the name SimulatorVM.ova. Start the image by using the 'START' button on the GUI. This action will open a terminal to interact with the VM. However, to be able to run any commands on the VM, the user will need an account.

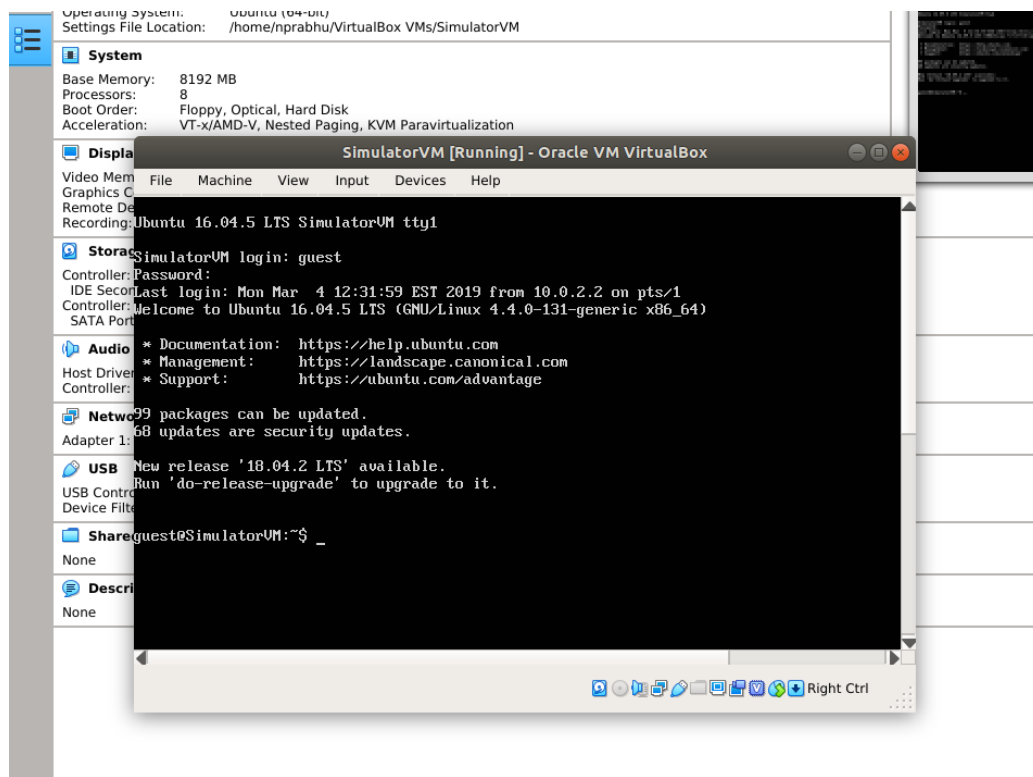


Figure 11: Logging into the VM

We have created a guest account on the VM with ‘sudo’ privileges. The username and password are **guest**. We will use this account to interact with the VM through an ssh terminal session, rather than through the VirtualBox GUI, which can be faster. Port forwarding will need to be enabled through the GUI. Open VirtualBox and select the SimulatorVMOva image in the left pane. Right click and go to -> Settings in the menu at the top.

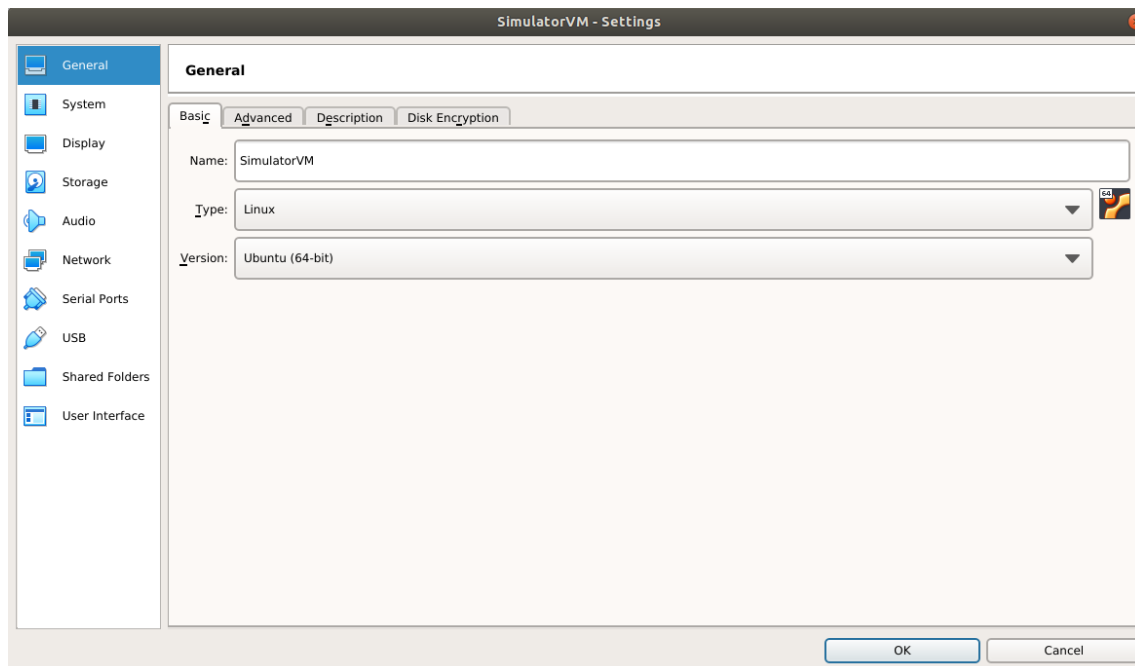


Figure 12: Changing network settings

Click the Network tab in the left pane. In this tab, click on Advanced and then on Port Forwarding.

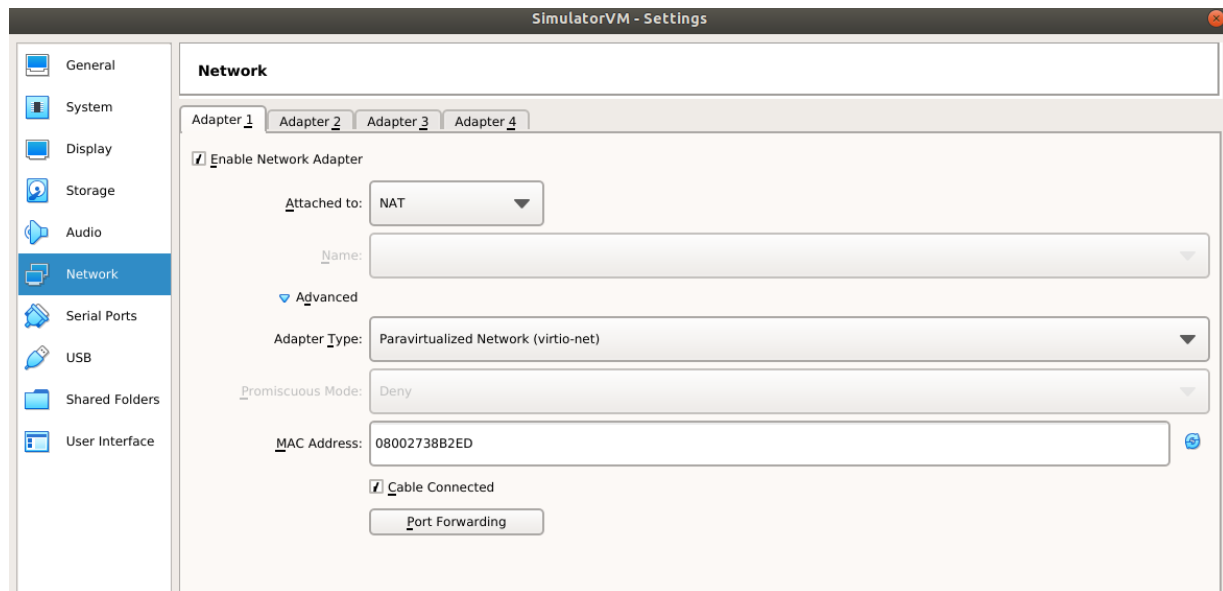


Figure 13: Adding Port Forwarding

If no rules exist for SSH, click the + sign in the upper right corner of the box. Type rule name as "SSH". The protocol should be "TCP", the Host IP should be empty, the Host Port can be "6022", the Guest IP should be empty, and the Guest Port should be "22". If the rule exists, leave it as it is.

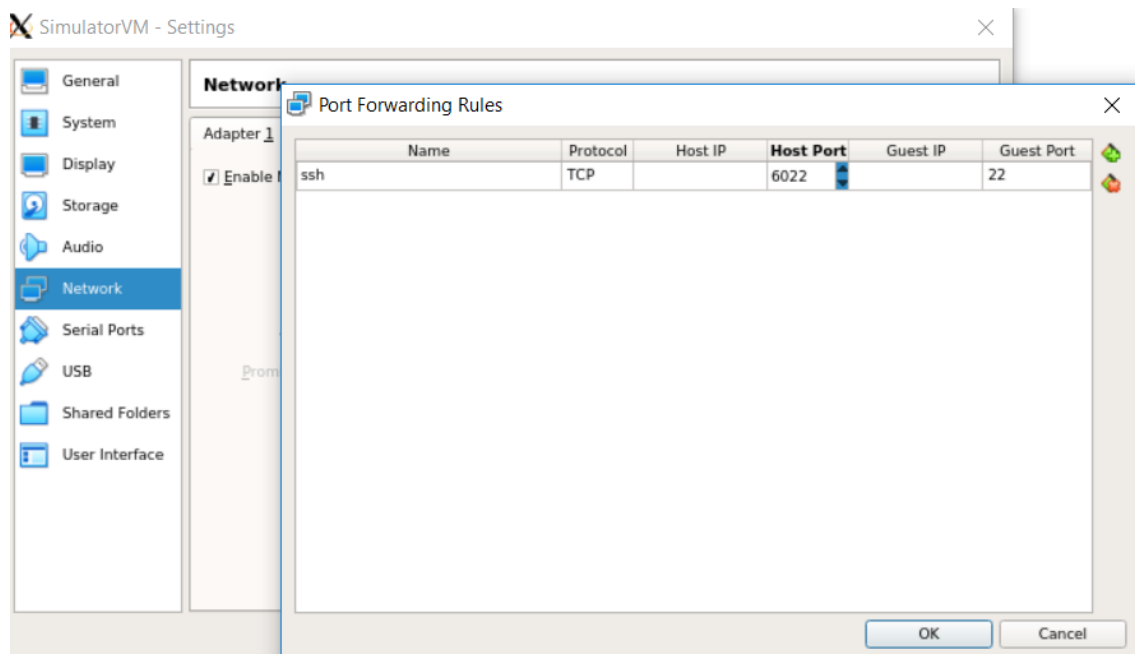


Figure 14: Adding port number and ssh forwarding rules

The GUI tends to be usually slow, so it is easier to use ssh connect in a terminal to interact with the VM. In an Ubuntu OS, open a terminal and use ssh as follows:

Connecting to SimulatorVM image

Open any terminal and use the following command to connect to the virtual machine.

ssh -p 6022 [guest@127.0.0.1](#)

```
[prabhu@prabhu-server ~]$ ssh -p 6022 guest@127.0.0.1
The authenticity of host '[127.0.0.1]:6022 ([127.0.0.1]:6022)' can't be established.
ECDSA key fingerprint is SHA256:0cZjZmciOUWi7tyQ1RS0/XW4EwKRDDnLA6No9eOkGgw.
ECDSA key fingerprint is MD5:8b:fa:76:e0:e8:43:f6:de:73:37:74:e6:a4:a4:86:98.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[127.0.0.1]:6022' (ECDSA) to the list of known hosts.
guest@127.0.0.1's password:
Welcome to Ubuntu 16.04.5 LTS (GNU/Linux 4.4.0-131-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

99 packages can be updated.
68 updates are security updates.

New release '18.04.2 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Wed Mar 13 13:46:29 2019
guest@SimulatorVM:~$
```

Figure 15: Logging into VM through host ssh terminal

If using Windows, you will need an SSH client program, such as Putty [7]. You will need to connect to localhost, change the default port from 22 to 6022 while creating the connection.

Once the ssh command is executed, the terminal will prompt for a password, enter “guest”. This will give access to the terminal through which a simulator can be built.

Now that a VM is connected, simulator building can start.

Docker and OpenvSwitch status

Before simulator is built, verify that Docker and OpenVswitch are functional.

Once logged in, a simple Docker command will print the current status of Docker containers on the VM:

sudo docker container ls -a

```
nprabhu@ubuntu:~$ sudo docker container ls -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
de0fb29ee4f2       jpetazzo/dind      "wrapdocker"       3 days ago         Exited (0) 2 days ago              sat1
306ce849fa0f       jpetazzo/dind      "wrapdocker"       3 days ago         Exited (0) 2 days ago              sat3
359431616a90       jpetazzo/dind      "wrapdocker"       3 days ago         Exited (0) 2 days ago              sat4
01ee3e8ae18e       jpetazzo/dind      "wrapdocker"       3 days ago         Exited (0) 2 days ago              sat2
```

Figure 16: Exited containers

This should list all the Docker containers (running or stopped because of -a flag). This shows all the stopped containers and their status.

sudo docker container ls

```
nprabhu@ubuntu:~$ sudo docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

Figure 17: No container running

The command shows all the running containers. If there are no running active containers, nothing is printed, as shown in Figure 17.

Now, the OpenvSwitch bridge is checked with the command:

sudo ovs-vsctl show

This command should print out the node structure of all the OpenVSwitch bridges and ports if they exist, otherwise it will just print the version of the Open vSwitch, as shown in Figure 18.

```
nprabhu@ubuntu:~$ sudo ovs-vsctl show
bc3692f1-9147-4bc5-9951-f71e42ddfd89
    ovs_version: "2.11.90"
```

Figure 18: OpenVSwitch bridge structure

The above commands confirm that the VM is successfully set and both software packages are interactive through the VM. Now scripts will be used to generate a satellite system.

On the VM terminal, you should be in the home directory currently. If not go to the home directory using “cd /home/guest/Airforce”

```
guest@SimulatorVM:~$ ls
Airforce
```

Figure 19: Folders in the VM

Using “ls” will list all the files. The folder ‘scripts’ in this directory contains all the files that will be used to generate the simulator. Folder master controller contains the master controller code and the GNAT_Docker folder contains C Code to run in the GNAT and PHYs.

```
guest@SimulatorVM:~/Airforce/airforce/scripts$ ls
build.sh          env_setup.sh      internal          sat_start.sh      testrunphy.sh
copy_files.sh     gen_gnat_gs.py    ovs_net_clear.sh sat_stop.sh        testrun.sh
Dockerfile        gen_phy_rf.py     ovs_net_setup.sh testkillphy.sh
env_clear.sh      get_logs.sh       reset_phy_ip.sh  testkill.sh
```

Figure 20: Scripts directory

The internal folder contains the scripts that will run inside each satellite Docker container and setup/clear intra-satellite connections.

```
guest@SimulatorVM:~/scripts$ cd internal
guest@SimulatorVM:~/scripts/internal$ ls
add_eth.sh  env_clear.sh  env_setup.sh
```

Figure 21: Internal Scripts Directory

The following table describes each script in brief. A more detailed explanation is provided later.

Script	Function
Sat_start.sh	Starts a container with a given satellite ID and No. of PHYs. If container doesn't exist, creates one.
Sat_stop.sh	Stops a container, given a satellite ID and No. of PHYs
Env_setup.sh	Sets up a multi-satellite simulator with inputs No. of satellites and No. of PHY's
Env_clear.sh	Clears the environment setup
Ovs_net_setup.sh	Setup links between the Master controller and Satellites
Ovs_net_clear.sh	Clear all the links between Master Controller and Satellites
Build.sh	Builds all C code from the GNAT_Docker into executables into the /target folder
Testrun.sh	Run all executable files in respective nested containers, given satnums,phys and gsnum
Testkill.sh	Kill all executable process in respective nested containers, given satnums,phys and gsnum
Getlogs.sh	Fetch the log files from the nested containers to the GNAT_Docker/logs directory
Dockerfile	Creates a Docker image to be used while generating containers
Internal/Add_eth.sh	Adds a virtual ethernet link given the Satellite ID, Container name of the container inside the satellite, name of the ethernet interface (custom) and the IP address (custom that the user needs to set)
Internal/Env_setup.sh	Set up the environment inside the satellite (Includes setting links between GNAT, PHYs, MSGEN and OVS) identified by satellite ID to be given as the input argument
Internal/env_clear.sh	Clear the environment inside the satellite

The GNAT_Docker folder contains the code run inside each component of a satellite. Figure 22 shows the seven main folders inside this folder. The first six folders contain code for each nested container. The GNAT, PHY, PHY-Direct and PHY-MGEN are common to all satellite nodes. The MRC and Portal folders are only applicable for the one satellite node that has these containers. The target folder contains all the executable files. This folder is mounted on each nested container volume to allow direct access/usability. This result can be seen in Figure 23.

```
guest@SimulatorVM:~/Airforce/airforce/GNAT_Docker$ ls
GNAT  PHY  PHY-Direct  PHY-MGEN  PHY-MRC  Portal  target
```

Figure 22: GNAT_Docker file contents

```

nprabhu@ubuntu:~$ docker container exec -it sat1 bash
Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.
sock: Get http://%2Fvar%2Frun%2Fdocker.sock/v1.39/containers/sat1/json: dial unix /var/run/docker.sock: connect: permission denied
nprabhu@ubuntu:~$ sudo docker container exec -it sat1 bash
root@2cd198168a73:/# sudo docker container exec -it gnat bash
root@25c56f916929:/# cd home/app/
root@25c56f916929:/home/app# ls
GNAT_GS  GNAT_SAT  GNAT_SAT_Debug  PHY-LCOMM  PHY-MGEN  PHY-MGEN_Debug  PHY-MRC  PHY-Portal  PHY-RF
root@25c56f916929:/home/app#

nprabhu@ubuntu:~$ sudo docker container exec -it sat1 bash
root@2cd198168a73:/# sudo docker container exec -it msgen bash
root@3453766fe3a8:/# cd home/app/
root@3453766fe3a8:/home/app# ls
GNAT_GS  GNAT_SAT_Debug  PHY-MGEN  PHY-MRC  PHY-RF
GNAT_SAT  PHY-LCOMM  PHY-MGEN_Debug  PHY-Portal
root@3453766fe3a8:/home/app#

```

Figure 23: Contents of target mounted to gnat container (left terminal) and msgen container (right terminal)

The master_controller folder contains the src directory that has the master_controller python code developed at UMass. It also includes some header/dependency files that we created for the master_controller.py file. The /json directory maintains the JSON visibility and traffic flow files provided by IFT and Leidos Corporation which are required for parsing the visibility information of the satellite to satellite communication. The /logs directory has the generated simulation log files, the reference table and the MC log file.

```

guest@SimulatorVM:~$ cd master_controller/
guest@SimulatorVM:~/master_controller$ ls
json  src
guest@SimulatorVM:~/master_controller$ cd src
guest@SimulatorVM:~/master_controller/src$ ls
includes.py  master_controller.log  master_controller.py  __pycache__  reference_table.py  set_link.sh

```

Figure 24: Mastercontroller folder contents

The JSON files in the /json directory are shown in Figure 25. The number at the end of each file represents the number of nodes the json file supports (e.g. 32 indicates a 32 node simulation). For instance, to simulate a 32-node system, ovsPortMap_32.json, VisibleBS_32.json, VisibleGPS_32.json and SCC_TrafficModel_1.json are needed. The traffic model file is common and does not influence the simulation. The JSON files without a suffix number are used for four-node simulation.

```
guest@SimulatorVM:~/Airforce/airforce/master_controller/json$ ls
ovsPortsMap_12.json      SCC_TrafficModel_2.json  VisibleBS_6.json
ovsPortsMap_16.json      VisibleBS_12.json        VisibleGPS_12.json
ovsPortsMap_24.json      VisibleBS_16.json        VisibleGPS_16.json
ovsPortsMap_32.json      VisibleBS1.json          VisibleGPS_24.json
ovsPortsMap_6.json       VisibleBS_24.json        VisibleGPS_32.json
ovsPortsMap.json         VisibleBS2.json          VisibleGPS_6.json
SCC_TrafficModel_1.json  VisibleBS_32.json        VisibleGPS.json
```

Figure 25: JSON files

Starting the simulator

A short example is presented here to demonstrate generation of a system using the scripts. A small case of four nodes is shown for brevity.

Starting containers:

The scripts folder is in the VirtualBox in the folder /home/guest/Airforce. Change the directory to make this the current directory. The Airforce folder contains the GNAT_Docker folder that has the code that will run inside the nested Docker containers. Now we can begin to build our simulator using the scripts.

To create a simulator with four satellites where each satellite has four PHYs we will use the script named 'env_setup.sh'. This script creates/starts the satellite system. It will start a network consisting of a number of satellites which need to be provided as an argument while running the script with a -s flag. The number of PHY's in each satellite is specified with a number following the -p flag. The ground station can be created by providing an input with -g flag. We take advantage of multiple processors to generate the satellite system. An integer following the command argument flag -j determines the number of parallel jobs that can be run, i.e. how many parallel processors can one use to create the satellites. Each Docker container is given a sat(ID) as a container_name in the scripts. In some cases the number of satellite container nodes required by the user might exceed the number of containers that have already been created. In such a case, the scripts will create new containers using the jpteazzo/dind image [8]. The scripts use a Docker file to build an image for the first time and uses this image to then generate additional containers. The following is an example of the generation of a four-node system with one ground station.

I) To create a 4-node system, run : **`./env_setup.sh -s 4 -p 2 -g 1`**

A sample output is show in Figure 26 (Note: Add -j *numberofcores* if using parallel processing)

```

guest@SimulatorVM:~/Airforce/airforce/scripts$ ./env_setup.sh -s 4 -p 2 -g 1
Start the openvswitch!
* Starting ovssdb-server
* system ID not configured, please use --system-id
* Configuring Open vSwitch system IDs
* Inserting openvswitch module
* Starting ovs-vswitchd
* Enabling remote OVSDDB managers
Creating a network with 4 satellites and 1 ground stations!
sat1
process_id=1918
mount: cgroup already mounted or cpu busy
mount: according to mtab, cgroup is already mounted on /sys/fs/cgroup/cpu,cpuacct
rmdir: failed to remove 'cpu': Not a directory
mount: cgroup already mounted or cpuacct busy
mount: according to mtab, cgroup is already mounted on /sys/fs/cgroup/cpu,cpuacct
rmdir: failed to remove 'cpuacct': Not a directory
mount: cgroup already mounted or net_cls busy
mount: according to mtab, cgroup is already mounted on /sys/fs/cgroup/net_cls,net_prio
rmdir: failed to remove 'net_cls': Not a directory
mount: cgroup already mounted or net_prio busy
mount: according to mtab, cgroup is already mounted on /sys/fs/cgroup/net_cls,net_prio
rmdir: failed to remove 'net_prio': Not a directory
* Starting Docker: docker
...done.
* Starting ovssdb-server
* system ID not configured, please use --system-id
* Configuring Open vSwitch system IDs
* Starting ovs-vswitchd
* Enabling remote OVSDDB managers

```

Figure 26: Starting a 4-node simulator

```

Starting satellite 4
sat4
process_id=31417
  54 pts/0    00:00:00 dockerd
* Starting ovssdb-server
* system ID not configured, please use --system-id
* Configuring Open vSwitch system IDs
* Starting ovs-vswitchd
* Enabling remote OVSDDB managers
gnat
msgen
rfphy
phy1
phy2
Setup the full connection for master controller with every gnat!
Create the ctrlbr bridge ...
Connection has been set!
Setup all PHY's with external OVS
Creating mainbridge connections
Create the main bridge ...
Creating links..
Turning it on..
Main bridge connection set!

```

Figure 27: Satellite 4 starting and creation of mainbridge and links

i) Once the containers have been created, we can check to see if the Docker containers are running:

sudo docker container ls

```
3b979940d449      jpetazzo/dind      "wrapdocker"      2 weeks ago
Up 6 minutes
3d60d9a9acad      jpetazzo/dind      "wrapdocker"      2 weeks ago
Up 6 minutes
c64fd4b9c35c      jpetazzo/dind      "wrapdocker"      2 weeks ago
Up 6 minutes
d76933dc82a5      jpetazzo/dind      "wrapdocker"      2 weeks ago
Up 6 minutes
sat1
```

Figure 28: Running satellite containers

ii) To check the internal connections of a container (for example sat1), use:

sudo docker exec sat1 /bin/sh -c "ifconfig"

```
~~~~~ ubuntu:/home/guest/AirForce/scripts$ sudo docker exec sat1 /bin/sh -c "sudo docker container exec phy2 /bin/sh -c 'ifconfig'"
[sudo] password for nprabhu:
direct.1c: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.0.102 netmask 255.255.255.0 broadcast 0.0.0.0
    ether 1e:2d:ca:72:29:12 txqueuelen 1000 (Ethernet)
    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

direct.1d: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.0.103 netmask 255.255.255.0 broadcast 0.0.0.0
    ether ce:be:4f:13:24:29 txqueuelen 1000 (Ethernet)
    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

eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.18.0.6 netmask 255.255.0.0 broadcast 172.18.255.255
    ether 02:42:ac:12:00:06 txqueuelen 0 (Ethernet)
    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

eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.2.1 netmask 255.255.255.0 broadcast 0.0.0.0
    ether 00:00:00:00:fe:ff txqueuelen 1000 (Ethernet)
    RX packets 16 bytes 1296 (1.2 KB)
    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

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    loop txqueuelen 1 (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
```

Figure 29: Interfaces of a nested container phy2

iii) To check the Open vSwitch configuration, use:

sudo ovs-vsctl show

```
root@ubuntu:/home/guest/AirForce/scripts$ sudo ovs-vsctl show
bc3692f1-9147-4bc5-9951-f71e42ddfd89
    Bridge ctrlbr
        Port "sat1.gnat"
            Interface "sat1.gnat"
        Port "sat3.gnat"
            Interface "sat3.gnat"
        Port ctrlbr
            Interface ctrlbr
                type: internal
        Port "sat2.gnat"
            Interface "sat2.gnat"
        Port "sat4.gnat"
            Interface "sat4.gnat"
    Bridge phybr
        Port phybr
            Interface phybr
                type: internal
        Port "sat4.phy1"
            Interface "sat4.phy1"
        Port "sat3.phy1"
            Interface "sat3.phy1"
        Port "sat3.phy2"
            Interface "sat3.phy2"
        Port "sat1.phy2"
            Interface "sat1.phy2"
        Port "sat2.phy1"
            Interface "sat2.phy1"
        Port "sat4.phy2"
            Interface "sat4.phy2"
        Port "sat2.phy2"
            Interface "sat2.phy2"
        Port "sat1.phy1"
            Interface "sat1.phy1"
```

Figure 30: OpenvSwitch bridge after four satellites created

II) To clear the four-node system, run:

`./env_clear.sh -s 4 -p 2 -g 1`

```
guest@SimulatorVM:~/Airforce/airforce/scripts$ ./env_clear.sh -s 4 -p 2 -g 1
Turn off all nodes! Clear the network setting!
gnat
msgen
rfphy
phy1
phy2
portphy
mrc
sat1
gnat
msgen
rfphy
phy1
phy2
sat2
gnat
msgen
rfphy
phy1
phy2
sat3
gnat
msgen
rfphy
phy1
phy2
sat4
gnat
msgen
rfphy
gs1
Delete the ctrlbr bridge
Delete the phybr bridge!
```

Figure 31: Clearing the environment

Clearing the environment will stop all the containers but will not remove them. However, all the links are deleted. Thus, when running `./env_setup.sh` again, the containers will be restarted. However the links will be recreated.

The number of satellites and PHYs in each satellite can be specified as arguments while running the `./env_setup.sh` script to generate a custom multi-node system. For example, for a 32-node system with four PHYs including one ground station run, `./env_setup.sh -s 31 -p 4 -g 1`. To clarify the functioning of the scripts we have included comments in the code.

Scripts overview

env_setup.sh

This script is used to create/start a network consisting of satellites and Open vSwitches. The inputs to this script are provided with flags -s, -p, -g and -j. The flagged inputs -s, -p, -g and -j indicate number of satellites, number of PHYs in each satellite, number of ground stations and job number for the number of parallel jobs that can be run, respectively. The job number is the number of cores on the VirtualBox that can be used to parallelly set up the satellites in the simulator. If not provided, the satellites will start sequentially. This script calls the sat_start.sh as many times as the number of satellites that is specified after the -s argument and another time if a GS value is specified. The script also sets up the Open vSwitch to connect all satellites and the master controller.

env_clear.sh

This script is used to clear the environment of existing connections and Docker containers. This script takes three arguments with flags -s, -p, -g and -j for number of satellites, number of PHYs in each satellite, number of ground stations and job number. If a job number is not provided, then it stops all satellites one by one. If there are several jobs, then it clears all the satellites in each job in parallel, followed by clearing the Open vSwitch structure.

sat_start.sh

This script creates/starts a container and is called by the env_setup.sh repeatedly to create the number of satellite containers required by the user. As seen from the example above, SatID and PHY_NUM input arguments, which correspond to the ID of the satellite and number of PHY's, are used to generate a satellite container. Each Docker container is given a sat(ID) as a container name, where ID is an integer from 1...N. If the user tries to create more nodes than the number that already exists in the VirtualBox, new containers are created using a custom image based on jptezzo/dind image [7], an image which enables creation of nested Docker containers. A dockerfile first generates the image with all dependencies and is subsequently used as a template for satellite containers.

Several dependencies are required for the execution and testing of the containers (net-tools, iproute2, ping, and tcpdump). These network related tools are used for setting links, viewing and editing network devices such as veth interfaces and communication testing. The tools are installed by the Docker file while building the image, followed by OVS installation.

Each container includes a GNAT Brain, MSGEN, RFPHY and several PHY containers. The number of PHYs are defined by the input parameter. An internal dev:ubuntu image is created using the dockerfile. All software packages and tools which are installed in the external container are replicated while creating this image (python3, vim, net-tools). Vim is a text editor that can be used for editing the code. This image then forms the base for the previously-mentioned device containers. For one satellite, two additional components, an MRC and a Portal, can be created, if desired. The Portal uses an IP and port pair to provide Internet access. When the portal-based container is created, a -p flag indicates the

presence of a port mapping between the host port forwarding Internet packets to the VM and the container. The nested container, portalphy has a mapping from the external container to the nested container. The MRC and portalphy are not used by the example simulations described subsequently in this document. The Open vSwitch bridge required to connect all components together is then created (br0) by the scripts.

The files from the internal folder that are required for internal setup of the container i.e. creating links between nested containers, are copied to the Docker satellite containers. The env_setup.sh file creates an internal Open vSwitch and sets up connections between the nested containers by adding veth interfaces. When the container is created for the first time, i.e. if the user needs to create more containers than already exist, new internal containers are created and the process ID of each is saved. If the containers exist, they are started and their process IDs are fetched. The script also periodically checks if the Docker daemon is running, since sometimes the daemon can switch off by itself.

sat_stop.sh

This script is called by the env_clear.sh to clear the environment. It calls the script env_clear.sh for the internal environment (a script in the internal folder). Finally, it stops the container.

ovs_net_setup.sh

This file inputs the number of satellites and creates connections between all satellites' GNATs and the master controller with the Open vSwitch. A ctrlbr(Open vSwitch) process is created for this setup. To create links, virtual ethernet links (veth) provided by Linux network tools are used. This arrangement requires an exclusive namespace for each container. A link is first created with two ends on the host. One end is inserted into the satellite container and then the add_eth.sh script from the internal directory is called to create the further connection to the nested container (GNAT). The other end is set as a port on the ctrlbr. The master controller is configured to have an interface connected to the ctrlbr. All IP addresses on the ctrlbr are in the subnetwork 192.169.0.x/24. All PHYs in each satellite are connected to a single OpenVswitch phybr bridge with a unique port name (e.g. sat1.phy1) using veth links. The interface end of this link in the satellite is labelled eth1 with IP 192.168.2.1/24.

ovs_net_clear.sh

This script clears the setup created by the ovs_net_setup.sh script. It deletes the ctrlbr, phybr and the virtual links in all satellites.

build.sh

This script builds and compiles all C files in the GNAT_Docker folder and stores the executables in the /target directory. This folder is mounted onto the containers while creating them such that the /home/app/ directory is linked to the /target on the host. This action helps avoid copying the files every time changes are made.

testrun.sh

This script is used to run all the executables in their respective containers. For example, it will use the /home/app directory of a 'gnat' container to run the GNAT_SAT file. Similarly, for 'msgen', it runs the

PHY-MGEN. The input parameters for this script are the number of satellites, number of PHYs per satellite and the number of ground stations.

testkill.sh

This script can be used to stop all the executables by issuing a ‘Kill’(9) signal to the running executable processes. A *ctrl+C* command does not affect all processes across 32 nodes. The input parameters have to be the same as the ones used for the *testrun.sh* script to ensure that all the components are stopped.

testrunphy.sh

This script is similar to *testrun.sh* but it can be used to only run the PHY-Direct executables in respective containers if only PHY code is run automatically. For example, the script will start the PHY-Direct executable file in the *phy1*, *phy2*, *phy3* and *phy4* containers of a satellite. It can also run the *rfphy* code in the *rfphy* containers when required. The input parameters are the number of satellites, number of PHYs per satellite and the number of ground stations.

testkillphy.sh

The script can be used to stop all PHY-Direct executables by issuing a ‘Kill’(9) signal in the PHY containers to stop the running PHY-Direct executable processes. The inputs have to be same as the ones used for the *testrun.sh* script to ensure that all components are stopped.

getlogs.sh

This script fetches the log files generated by the executing code. Log statements are added in the C code, which publishes the output into a log file. This script copies the log files from each container out to the host/VM. Note: Make sure to gracefully kill all the processes before getting the log files. This is an important issue to consider if the containers are running in a terminal.

Internal scripts

These scripts are responsible for the generation/starting or deletion of the connections between nested containers.

env_setup.sh

This script is used to create the links between the GNAT, RFPHY, MSGEN and the PHYs. It takes the arguments *SAT_ID* and *PHYNUM*. These arguments are supplied by *sat_start.sh* to complete the internal setup for each satellite container. The *SAT_ID* allows unique identification of each satellite while the *PHY_NUM* indicates the number of PHYs in the given satellite. The script ensures that the Open vSwitch and the Docker daemon are up and running in the container before starting the setup. All nested container interfaces (GNAT, MSGEN, PHYs, RFPHY etc) require an IP address. For the internal subnetwork, the IP address is in the range ‘192.168.0.x’, where x is defined uniquely for each container interface. In the script, we use .2 for GNAT .50/.51 for MSGEN control/data interface, .1xx for PHY control and 2xx for PHY data interfaces. For the satellite with a portal and MRC, .70 and .80 are assigned, respectively. The PID of each container is used to create symbolic links within the namespace of the Docker container. This allows identification of each satellite using its namespace to

add the virtual links. An internal bridge (br0) acts as the central switch for all the units within the satellite. The links between the GNAT and br0 are set using the veth links with one end on the Open vSwitch as a port and the other in the namespace. This action is repeated for the MSGEN and RFPHY containers. The same process is followed for connecting all the PHYs to this internal bridge.

env_clear.sh

This script removes all the links by removing the veth links for individual components of the satellite and deletes the br0. It requires PHY_NUM as input to clear all PHY container link settings. The nested containers are subsequently stopped.

add_eth.sh

This script adds eth1 interfaces required for external connections. It is called from the env_setup.sh script. It uses the SatID to generate a MAC address for the eth1 interface. The inputs to the script are satellite ID, container ID, container name, ethernet interface name, IP address for the interface. These arguments can be provided in order if needed to call the script for a custom link addition.

For example: **./add_eth.sh 1 0 gnat ctrl.gnat 192.169.0.1/24**

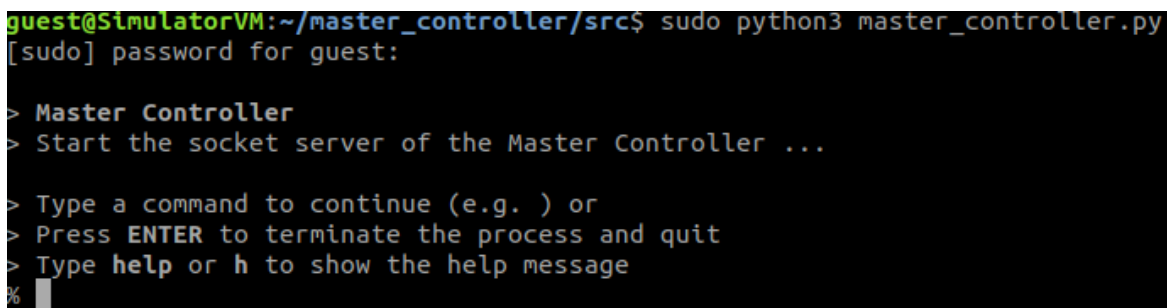
This command will add an ethernet link in the GNAT container of satellite 1 with the name ctrl.gnat and IP address 192.169.0.1/24.

Running the Simulator

1) Running a 4-node simulation

Using the container setup, we can execute application code designed for simulator-based testing. This code which models inter-satellite communication and was originally developed by Don Fronterhouse at PnP Innovations, is in the developmental stage. System setup by a user begins by opening a terminal on a host system and connecting to the VM using ssh. Once logged in, the first step involves running the master controller. In the airforce folder, change directory to the **master_controller** directory and then to the **src** directory. Then run the following command:

sudo python3 master_controller.py



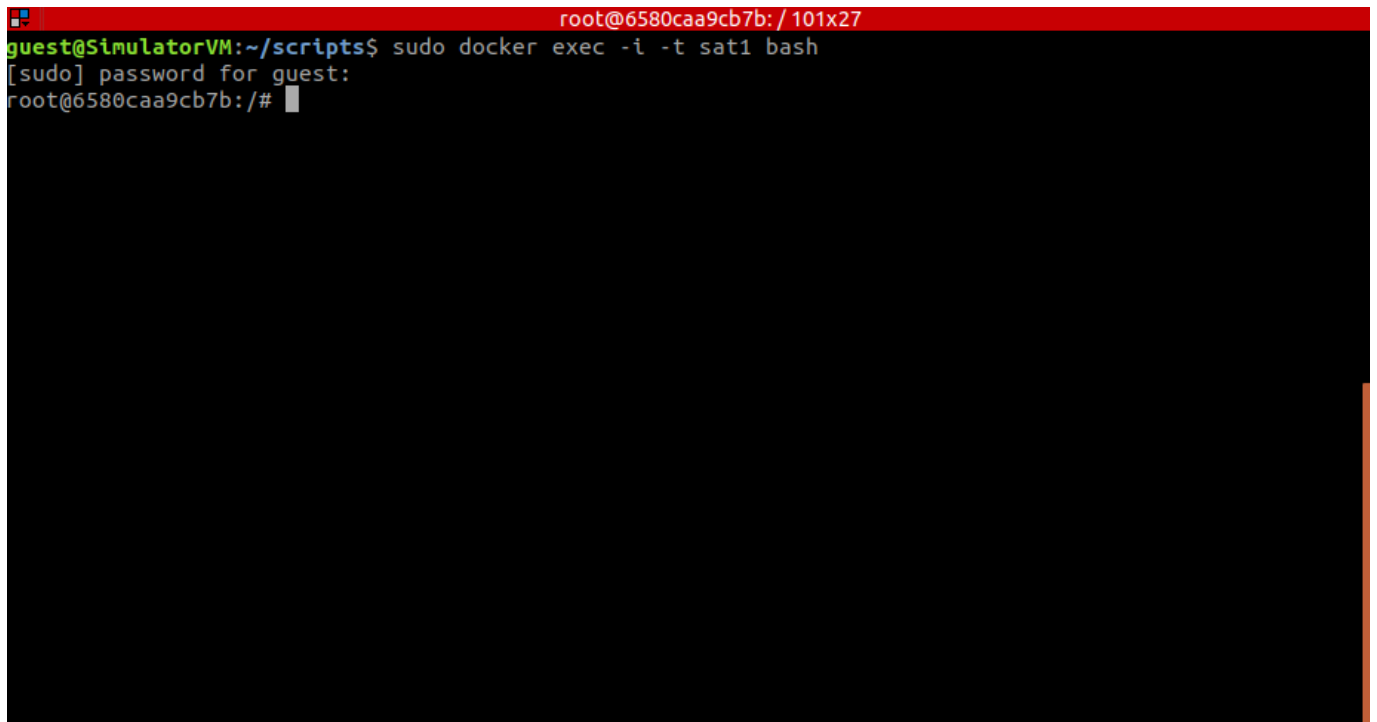
```
guest@SimulatorVM:~/master_controller/src$ sudo python3 master_controller.py
[sudo] password for guest:

> Master Controller
> Start the socket server of the Master Controller ...

> Type a command to continue (e.g. ) or
> Press ENTER to terminate the process and quit
> Type help or h to show the help message
% █
```

Figure 32: Starting the master controller

The next step is to attach the containers to separate terminals for testing. The following example demonstrates testing of a four-node simulator where the data is exchanged between all the satellites. As a start, a single container is attached to a bash terminal, as shown in Figure 33. This action provides access to the satellite Docker container, 'sat1'. If prompted enter 'guest' as the password:

A terminal window with a red title bar containing the text 'root@6580caa9cb7b: / 101x27'. The terminal shows a user 'guest' at 'SimulatorVM' in the directory '~/scripts' running the command 'sudo docker exec -i -t sat1 bash'. A prompt '[sudo] password for guest:' is shown, followed by the user becoming 'root' at '6580caa9cb7b' with a root shell prompt '#'.

```
root@6580caa9cb7b: / 101x27
guest@SimulatorVM:~/scripts$ sudo docker exec -i -t sat1 bash
[sudo] password for guest:
root@6580caa9cb7b:/#
```

Figure 33: Attach container sat1

In the satellite, other nested containers such as the GNAT, MSGEN and PHY can be attached.

To create this arrangement for multiple nested containers and satellites, multiple terminals are required. A user needs to open several terminals and log into the SimulatorVM on each. A tool such as terminator [9] can also be used to create multiple terminals on a single screen.

Using the same command as above, open two terminals for sat1 and then in each one, attach a different nested container as follows:

Sudo docker exec -it gnat bash

Sudo docker exec -it msgen bash

Repeat these actions for sat2 to sat4 by first attaching containers in two terminals and then attaching GNAT and MSGEN, respectively, in each individual one.

Figure 34 shows the terminals for all GNATs and MSGENs in four satellites. From left to right the GNATs are on the top row and the MSGENs are on the bottom row. Since there are several PHYs for each satellite, it is monotonous to launch several terminals for each PHY. For the purpose of testing, we used an edited version of testrun.sh, the testrunphy.sh script to run the PHYs. This allows us to observe the GNAT and MSGEN output on the screen, while the PHYs run in the background. The test is killed off by using testkillphy.sh (discussed in later sections).

Figure 34: Attaching a different terminal for gnat, msgen and phy test runs

After attaching eight different terminals for four satellites, attach two more for the ground station as shown next:


```

naren@naren-Inspiron-7370:~$ ./connecttovirtualbox
Last login: Fri Jun  7 20:32:37 2019 from 24-179-114-82.dhcp.oxfr.ma.charter.com
[nprabhu@rcglabserver ~]$ ssh -p 6022 -X guest@127.0.0.1
guest@127.0.0.1's password:
Welcome to Ubuntu 16.04.5 LTS (GNU/Linux 4.4.0-138-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

162 packages can be updated.
96 updates are security updates.

New release '18.04.2 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Fri Jun  7 20:32:53 2019 from 10.0.2.2
guest@SimulatorVM:~$ sudo docker exec -it gs1 bash
root@818b98417e0f:/home# sudo docker exec -it gnat bash
root@bffaefbaafae:/home/app#

naren@naren-Inspiron-7370:~$ ./connecttovirtualbox
Last login: Fri Jun  7 20:38:48 2019 from 24-179-114-82.dhcp.oxfr.ma.charter.com
[nprabhu@rcglabserver ~]$ ssh -p 6022 -X guest@127.0.0.1
guest@127.0.0.1's password:
Welcome to Ubuntu 16.04.5 LTS (GNU/Linux 4.4.0-138-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

162 packages can be updated.
96 updates are security updates.

New release '18.04.2 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Fri Jun  7 20:39:07 2019 from 10.0.2.2
guest@SimulatorVM:~$ sudo docker exec -it gs1 bash
root@818b98417e0f:/home# sudo docker exec -it msgen bash
root@d114e85f23e2:/home/app#

```

Figure 35:GS gnat and msgen containers

It may be necessary to change the directory in each container to run the respective executables. The C code for each component is compiled on the host and then copied to the directory /Gnat_Docker/target which is the default directory that is mounted while creating each Docker container. Docker allows for mounting between a directory on a host to one in the containers (i.e. /home/app/). The build.sh script compiles the code and copies it to the directory to run the executable. The right directory must be used for each component, i.e /home/app/. For instance, in the terminal in which the GNAT was attached, change directory, cd, to **/home/app/**. Repeat the steps for all sat terminals.

Next, the exe files in each of the containers are executed to test the code. These files were copied and then compiled by the scripts during setup. To run the code, use the exe files in each container. In both the GNAT containers, type **./GNAT_Debug**. Repeat this action for MSGEN containers using **./PHY-MGEN_Debug**. These two code versions allow the printing of all the necessary statements in the code. If print statements are not important, one can use the simple **./GNAT** and **./PHY-MGEN**. If running the PHY manually, this step should be done before PHY, as the code is designed in a way that the MSGEN should run before the PHYs. The PHY code is run in the PHY containers using, **./PHY-Direct**. Since scripts are used to run the simulator, open another tab, and create two windows. Log into the VM and then cd to /home/airforce/scripts. One script that should be executed is testrunphy.sh and the other is testrunkill.sh. After confirming that the GNAT and MSGEN containers are up, run ./testrunphy.sh as follows:

./testrunphy.sh 4 2 1

The following figures show the exchanged messages between the components when the C code is running. In Figure 36, the top row includes the GNAT of each satellite and the bottom row includes the MSGENs. The exchanged messages can be seen in each terminal.

[illegible]

Figure 36: *sat1* to *sat4* components communicating

Messages sent from the MSGEN of one satellite to another are seen. The NAT and routing table can be seen in the MSGEN terminals in Figure 37, indicating its own GNAT IP address (GA) and the other GNAT addresses. i.e the sat4 GNAT has its GA (10.10.1.4) and all the directly or indirectly connected GNATs 10.10.1.1, 10.10.1.2 and 10.10.1.3.

guest@SimulatorVM: ~/Airforce/airforce/scripts	root@af67fb7bafeaa: /home/app	guest@SimulatorVM: ~/Airforce/airforce/scripts	root@d114e85f23e2: /home/app
<pre> route #6, addr=0x2010a0a, cost=0x3e8, MAC=0x7e,0x90,0x62,0x4e,0xc0,0x60 route #7, addr=0x1010a0a, cost=0x3e8, MAC=0x7e,0x90,0x62,0x4e,0xc0,0x60 buildMessage...len=17 srcIP=0x200a8c0, dstIP=0x3200a8c0 outputMessage...sendto resp=59 buildMessage...len=17 srcIP=0x200a8c0, dstIP=0x6600a8c0 outputMessage...sendto resp=59 buildMessage...len=17 srcIP=0x200a8c0, dstIP=0x6400a8c0 outputMessage...sendto resp=59 buildMessage...len=17 srcIP=0x200a8c0, dstIP=0x6500a8c0 outputMessage...sendto resp=59 end of switch...6 wait for message from a PHY recv 25 bytes from PHY at 192.168.0.100 bytes=0,0,0,0,c0,a8,0,64,7,0,0,0,1,a,a,1,3,7e,90,62,4e,ce,60,e8,3, match=2 msgType=0x6 end of switch...6 wait for message from a PHY </pre>	<pre> route #6, addr=0x4010a0a, cost=0x3e8, MAC=0x26,0xa1,0x20,0xa0,0x87,0x1e route #7, addr=0x2010a0a, cost=0x3e8, MAC=0x26,0xa1,0x20,0xa0,0x87,0x1e buildMessage...len=17 srcIP=0x200a8c0, dstIP=0x3200a8c0 outputMessage...sendto resp=59 buildMessage...len=17 srcIP=0x200a8c0, dstIP=0x6600a8c0 outputMessage...sendto resp=59 buildMessage...len=17 srcIP=0x200a8c0, dstIP=0x6400a8c0 outputMessage...sendto resp=59 buildMessage...len=17 srcIP=0x200a8c0, dstIP=0x6500a8c0 outputMessage...sendto resp=59 end of switch...6 wait for message from a PHY recv 25 bytes from PHY at 192.168.0.101 bytes=0,0,0,0,c0,a8,0,65,7,0,0,0,1,a,a,1,1,26,a1,20,a0,87,1e,e8,3, match=3 msgType=0x6 end of switch...6 wait for message from a PHY </pre>	<pre> route #6, addr=0x1010a0a, cost=0x3e8, MAC=0x12,0x10,0x72,0xf,0xf5,0x71 route #7, addr=0x3010a0a, cost=0x3e8, MAC=0x12,0x10,0x72,0xf,0xf5,0x71 buildMessage...len=17 srcIP=0x200a8c0, dstIP=0x3200a8c0 outputMessage...sendto resp=59 buildMessage...len=17 srcIP=0x200a8c0, dstIP=0x6600a8c0 outputMessage...sendto resp=59 buildMessage...len=17 srcIP=0x200a8c0, dstIP=0x6400a8c0 outputMessage...sendto resp=59 buildMessage...len=17 srcIP=0x200a8c0, dstIP=0x6500a8c0 outputMessage...sendto resp=59 end of switch...6 wait for message from a PHY recv 25 bytes from PHY at 192.168.0.101 bytes=0,0,0,0,c0,a8,0,65,7,0,0,0,1,a,a,1,4,12,10,72,f,f5,71,e8,3, match=3 msgType=0x6 end of switch...6 wait for message from a PHY </pre>	<pre> srcIP=0x200a8c0, dstIP=0x6600a8c0 outputMessage...sendto resp=59 buildMessage...len=17 srcIP=0x200a8c0, dstIP=0x6400a8c0 outputMessage...sendto resp=59 buildMessage...len=17 srcIP=0x200a8c0, dstIP=0x6500a8c0 outputMessage...sendto resp=59 end of switch...6 wait for message from a PHY recv 25 bytes from PHY at 192.168.0.101 bytes=0,0,0,0,c0,a8,0,64,7,0,0,0,1,a,a,1,2,46,61,2,83,d8,3e,e8,3, match=2 msgType=0x6 end of switch...6 wait for message from a PHY </pre>
<pre> 0,0x0,0x1,0xa,0xa,0x1,0x2,0x7e,0x90,0x62,0x4e,0x ce,0x60,0xe8,0x3, added ROUTE entry #6, addr=0x2010a0a,MAC=0x7e,0x9 0,0x62,0x4e,0xc0,0x60 received GNAT control message..len=45, cmd=7 bytes=0x45,0x0,0x0,0x2d,0xf,0x79,0x40,0x0,0x40,0x 11,0xa9,0xc2,0xc0,0xa8,0x0,0x2,0xc0,0xa8,0x0,0x32 ,0xc0,0x65,0x75,0x3a,0x0,0x19,0xf1,0xf4,0x7,0x0,0 x0,0x0,0x1,0xa,0xa,0x1,0x1,0x7e,0x90,0x62,0x4e,0x ce,0x60,0xe8,0x3, added ROUTE entry #7, addr=0x1010a0a,MAC=0x7e,0x9 0,0x62,0x4e,0xc0,0x60 1 Enter Text: h Select Destination: NAT 0. 10.10.1.2 NAT 1. 10.10.1.1 NAT 2. 10.10.1.1 RTE 3. 10.10.1.2 RTE 4. 10.10.1.3 RTE 5. 10.10.1.4 RTE 6. 10.10.1.4 RTE 7. 10.10.1.3 RTE 8. 10.10.1.1 RTE 9. 10.10.1.2 RTE 10. 10.10.1.1 </pre>	<pre> 0,0x0,0x1,0xa,0xa,0x1,0x4,0x26,0xa1,0x20,0xa0,0x 87,0x1e,0xe8,0x3, added ROUTE entry #6, addr=0x4010a0a,MAC=0x26,0xa 1,0x20,0xa0,0x87,0x1e received GNAT control message..len=45, cmd=7 bytes=0x45,0x0,0x0,0x2d,0xf,0x79,0x40,0x0,0x40,0x 11,0xa9,0xc2,0xc0,0xa8,0x0,0x2,0xc0,0xa8,0x0,0x32 ,0xc0,0x65,0x75,0x3a,0x0,0x19,0xf0,0xd5,0x7,0x0,0 x0,0x0,0x1,0xa,0xa,0x1,0x2,0x26,0xa1,0x20,0xa0,0x 87,0x1e,0xe8,0x3, added ROUTE entry #7, addr=0x2010a0a,MAC=0x26,0xa 1,0x20,0xa0,0x87,0x1e 1 Enter Text: h Select Destination: NAT 0. 10.10.1.2 NAT 1. 10.10.1.2 NAT 2. 10.10.1.2 RTE 3. 10.10.1.1 RTE 4. 10.10.1.3 RTE 5. 10.10.1.4 RTE 6. 10.10.1.3 RTE 7. 10.10.1.1 RTE 8. 10.10.1.2 RTE 9. 10.10.1.4 RTE 10. 10.10.1.2 </pre>	<pre> 0,0x0,0x1,0xa,0xa,0x1,0x1,0x12,0x10,0x72,0xf,0xf 5,0x71,0xe8,0x3, added ROUTE entry #6, addr=0x1010a0a,MAC=0x12,0x1 0,0x72,0xf,0xf5,0x71 received GNAT control message..len=45, cmd=7 bytes=0x45,0x0,0x0,0x2d,0xf,0x79,0x40,0x0,0x40,0x 11,0xa9,0xc2,0xc0,0xa8,0x0,0x2,0xc0,0xa8,0x0,0x32 ,0xc0,0x65,0x75,0x3a,0x0,0x19,0xf0,0xd5,0x7,0x0,0 x0,0x0,0x1,0xa,0xa,0x1,0x3,0x12,0x10,0x72,0xf,0xf 5,0x71,0xe8,0x3, added ROUTE entry #7, addr=0x3010a0a,MAC=0x12,0x1 0,0x72,0xf,0xf5,0x71 1 Enter Text: h Select Destination: NAT 0. 10.10.1.3 NAT 1. 10.10.1.3 NAT 2. 10.10.1.3 RTE 3. 10.10.1.1 RTE 4. 10.10.1.2 RTE 5. 10.10.1.4 RTE 6. 10.10.1.4 RTE 7. 10.10.1.3 RTE 8. 10.10.1.2 RTE 9. 10.10.1.1 RTE 10. 10.10.1.3 </pre>	<pre> 0,0x0,0x1,0xa,0xa,0x1,0x4,0xd2,0xfc,0x40,0x7f,0x 31,0x77,0xe8,0x3, added ROUTE entry #6, addr=0x4010a0a,MAC=0xd2,0xf c,0x40,0x7f,0x31,0x77 received GNAT control message..len=45, cmd=7 bytes=0x45,0x0,0x0,0x2d,0xf,0x79,0x40,0x0,0x40,0x 11,0xa9,0xc2,0xc0,0xa8,0x0,0x2,0xc0,0xa8,0x0,0x32 ,0xc0,0x65,0x75,0x3a,0x0,0x19,0xb,0x83,0x7,0x0,0x 0,0x0,0x1,0xa,0xa,0x1,0x4,0x46,0x61,0x2,0x83,0xd8 ,0x3e,0xe8,0x3, added ROUTE entry #7, addr=0x4010a0a,MAC=0x46,0x6 1,0x2,0x83,0xd8,0x3e 1 Enter Text: h Select Destination: NAT 0. 10.10.1.4 NAT 1. 10.10.1.4 NAT 2. 10.10.1.4 RTE 3. 10.10.1.2 RTE 4. 10.10.1.1 RTE 5. 10.10.1.3 RTE 6. 10.10.1.3 RTE 7. 10.10.1.1 RTE 8. 10.10.1.2 RTE 9. 10.10.1.4 RTE 10. 10.10.1.4 </pre>

Figure 37: Sat1-sat4 msgen showing routing table

To verify the connections, examine the master controller log file, mastercontroller.log. This file stores the logs of the messages exchanged by the master controller with the GNAT. After the four-node simulation, we can observe the reference table to observe the connected satellites and the PHYs used. These can be found in the directory mastercontroller/logs. Figure 38 shows a sample reference table for a 10 second simulation of four nodes.

```

{
  "SimTime": 0,
  "SatDef": {
    "1": {
      "satID": 36585,
      "TrafficMode": {
        "MsgID": 1,
        "StartTime": 5,
        "Action": 1,
        "RepeatCount": 1000,
        "MsgModel": 1,
        "MsgsPerSec": 1,
        "MsgSize": 125,
        "DstID": 1
      },
      "phyNums": 3,
      "isActive": true,
      "phyStats": {
        "1": {
          "phyID": 2,
          "dstIdx": 1,
          "phyType": 3,
          "isOccupied": true,
          "dstID": 3,
          "phyStatus": "up"
        },
        "2": {
          "phyID": 3,
          "dstIdx": 2,
          "phyType": 3,
          "isOccupied": true,
          "dstID": 2,
          "phyStatus": "up"
        },
        "3": {
          "phyID": 1,
          "dstIdx": 0,
          "phyType": 1,
          "isOccupied": false,
          "dstID": 0,
          "phyStatus": "up"
        }
      },
      "connectedSATS": {
        "1": 0,
        "2": 1,
        "3": 1,
        "4": -1
      }
    }
  }
}

```

Figure 38:Reference table for 4 node system, showing connections of sat 1

2) JSON input files

Satellite communication needs directives for simulation and configuration. The visibility files provided by IFT dictate how the satellites will be connected with each other, how long the simulation will run and how the visibility of the satellites changes with respect to each other over time. In folder /master_controller/json/ there are several versions of the visibility files, VisibleGPS and VisibleBS.

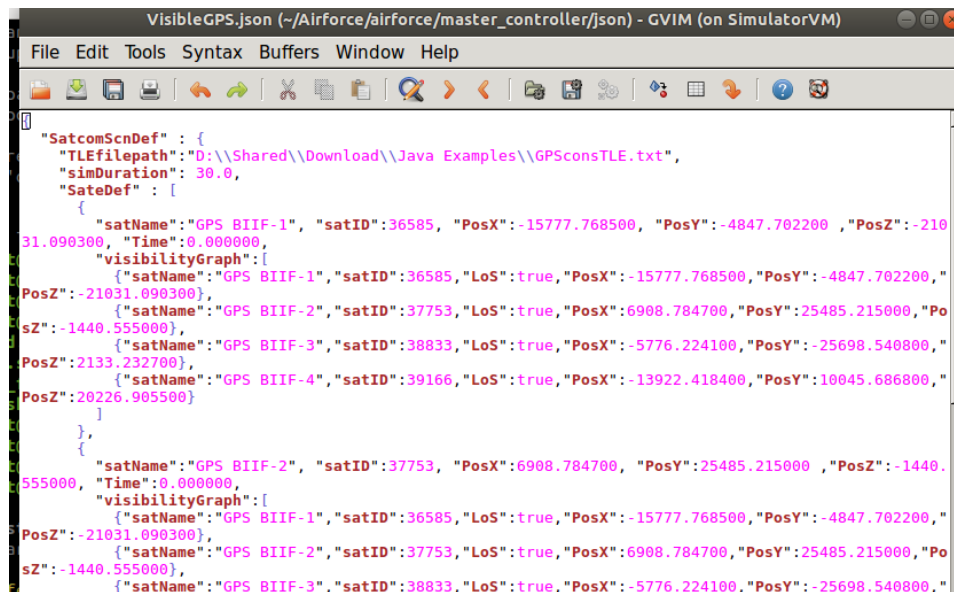


Figure 39:GPS 4 node visibility json file

```

{
  "SatcomScnDef" : {
    "TLEfilepath": "D:\\Shared\\Download\\Java Examples\\GPSconsTLE.txt",
    "simTime": "2019-02-05T00:00:00.000Z",
    "simDuration": 30.0,
    "SateDef" : [
      {
        "BSName": "MC Colorado Spring", "BSID": 1, "PosX": 4324.694434, "PosY": 2471.735847, "PosZ": 3970.158822, "Time": 0.000000,
        "visibilityGraph": [
          { "satName": "GPS BIIF-1", "satID": 36585, "LoS": false, "PosX": -15777.768500, "PosY": -4847.702200, "PosZ": -21031.090300 },
          { "satName": "GPS BIIF-2", "satID": 37753, "LoS": true, "PosX": 6908.784700, "PosY": 25485.215000, "PosZ": -1440.555000 },
          { "satName": "GPS BIIF-3", "satID": 38833, "LoS": false, "PosX": -5776.224100, "PosY": -25698.540800, "PosZ": 2133.232700 },
          { "satName": "GPS BIIF-4", "satID": 39166, "LoS": true, "PosX": -13922.418400, "PosY": 10045.686800, "PosZ": 20226.905500 }
        ]
      },
      {
        "BSName": "MC Colorado Spring", "BSID": 1, "PosX": 4324.694434, "PosY": 2471.735847, "PosZ": 3970.158822, "Time": 10.000000,
        "visibilityGraph": [
          { "satName": "GPS BIIF-1", "satID": 36585, "LoS": false, "PosX": -15755.507400, "PosY": -4877.559200, "PosZ": -21040.792500 },
          { "satName": "GPS BIIF-2", "satID": 37753, "LoS": true, "PosX": 6888.037600, "PosY": 25492.399300, "PosZ": -1408.430900 },
          { "satName": "GPS BIIF-3", "satID": 38833, "LoS": false, "PosX": -5753.119300, "PosY": -25700.974500, "PosZ": 2164.479100 },
          { "satName": "GPS BIIF-4", "satID": 39166, "LoS": true, "PosX": -13954.229500, "PosY": 10027.450300, "PosZ": 20214.276800 }
        ]
      }
    ]
  }
}

```

Figure 40: Visible BS 4 node json file

The former file specifies the visibility of the satellites with respect to each other over time. The latter file provides this information for the ground station. The JSON files play a crucial role in the simulation since the master controller determines how long intersatellite communication will remain static before link reconfiguration from these JSON files. This duration is indicated by the `simDuration` parameter. The files also indicate the inter-satellite link cost. For error-free simulation, the number of satellites running in the simulator should be the same as the number present in the visibility JSON file. The master controller, upon receiving the hello messages from the GNAT, assigns each one a GA IP address and sends visibility data. If an incorrect number of satellites is found, the master controller will throw an error. Also, the master controller checks the number of satellites in the JSON files and makes sure it receives a hello message from each satellite. Thus, each JSON file version indicates the number of satellites it can support for simulation. For example, `VisibleGPS_32.json` can support 32 satellites.

3) Running 32-node simulation

To run the 32-node simulation, the same procedure used for a four-node simulation can be followed with the only difference being the quantity of GNAT and MSGEN terminals that must be generated. However, to simplify simulation versus the previously-described four-node case, scripts can be used to run the simulator without the need for terminals per individual satellite. Please be sure to build the needed files before you create or start a simulation. To build the files if any changes have been made, use the `build.sh` script from the scripts directory. To set up a simulator with 32 nodes (31 satellites and 1 GS) with four PHYs each, change directory in a terminal to `/scripts` and run the command `./env_setup -s 31 -p 4 -g 1` as shown in Figure 41. Note: Please make sure you delete the older four containers and start these containers afresh. (Use *`docker system prune`*).

```

Delete the phybr bridge!
guest@SimulatorVM:~/Airforce/airforce/scripts$ ./env_setup.sh -s 31 -p 4 -g 1

```

Figure 41: Setup 32 nodes simulator

One can clear the node configurations as shown in Figure 42:

```

guest@SimulatorVM:~/Airforce/airforce/scripts$ ./env_clear.sh -s 4 -p 2 -g 1
Turn off all nodes! Clear the network setting!
gnat
msgen
rfphy
phy1
phy2
portphy
mrc
sat1
gnat
msgen
rfphy
phy1
phy2
sat2
gnat
msgen
rfphy
phy1
phy2
sat3
gnat
msgen
rfphy
phy1
phy2
sat4
gnat
msgen
rfphy
gs1
Delete the ctrlbr bridge
Delete the phybr bridge!

```

Figure 42: Clear 32 nodes system

A few changes are required to be able to use the master controller with 32 nodes. First, the user needs to change the number of PHYs per node defined in the /src/mastercontroller.py. This step can be seen in Figure 43, where we change the parameter self._phys_nums to 5 (For 4 PHYs and 1 RFPHY).

```

class MasterController(object):
    def __init__(self):
        self._host = includes.HOST
        self._server_port = includes.SERVER_PORT
        self._client_port = includes.CLIENT_PORT
        self._server = None
        self._enable_server = False
        self._server_closed = True
        self._sats_num = 0
        self._bs_num = 0
        self._phys_num = 5
        self._hello_num = 0
        self._sim_start_time = 0
        self._sim_duration_time = 0
        self._sim_current_time = 0
        self._sim_next_time = 0
        self._json_vis_gps = None
        self._json_vis_bs = None
        self._json_loaded = False
        self._phy_def_cnt = 0
        self._phy_ready = False

```

Figure 43: Change number of PHYs for 32 node

To change the JSON file that signifies the number of nodes that are used for simulation, the user needs to add the appropriate JSON files that are to be read. The changes can be seen in Figure 44. All the JSON files used have the postfix of _32 for a 32 node system.

```
def main():
    # Instantiate the MasterController object
    master_ctrl = MasterController()

    print("\n> \033[1mMaster Controller\033[0m")
    print("> Start the socket server of the Master Controller ...")

    # Start the socket server of the Master Controller
    master_ctrl.control_server()

    # Loading default json file
    master_ctrl.load_json("../json/VisibleGPS_32.json",
                          "../json/VisibleBS_32.json",
                          "../json/SCC_TrafficModel_2.json",
                          "../json/ovsPortsMap_32.json")
```

Figure 44: Change the json files to be used for simulation

Using `testrun.sh`, the number of satellites, PHYs per satellite, and ground stations are specified as arguments to perform a simulation for a stipulated time. For example, for a 32 node system, *use* `./testrun.sh 31 4 1`. The time duration, as previously mentioned, is determined by the JSON files. This script starts all the executables in the containers in a given order. The simulation can be stopped by killing all the processes in the satellite Docker containers. `Testkill.sh` achieves goal this by sending termination signals to the container processes. Similar to the four-node simulation, `./testkill.sh 31 4 1` is used to stop the simulation.

4) Time slots and varying simulation time

As discussed previously in this section, the duration of simulation time is determined by reading a visibility JSON file. Depending on the *simDuration* value of the JSON file in `VisibleGPS` and `VisibleBS`, the master controller will extract simulation duration information from the JSON file as seen in Figure 39 and 40. Effectively, a “time slot” indicates how long the visibility information remains valid before a potential change in visibility between satellites. Currently we use 10 second time slots (e.g. the visibility is guaranteed to remain stable for at least 10 seconds). This value can be seen in the “Time:” value for each satellite JSON item. Specifying a *simDuration* of 10 will lead to a run of just one time slot from 0-10s. The position-related data for each time slot is specified for the satellites in the JSON file. The user can increase the time duration of the simulation, assuming that positioning information for an appropriate number of time slots have been specified for each satellite in the JSON file. At the end of each simulation time slot, the master controller outputs a JSON file named ‘`refTable_X.json`’, with X being the timeslot for which the JSON file corresponds. This file illustrates the links and status of each PHY and satellite from the master controller’s point of view at the end of each time slot. This file is quite useful for debugging and can be used to check the link settings indicating the satellite in focus and the related PHYs used for achieving the connection. An example file is shown in Figure 46.

5) Affirmation using log files

Running a simulation without terminals (e.g. using a script for execution rather than opening lots of windows) prevents output observation in the terminals. To address this issue, logging has been added to

GNAT and PHY code so the results of the simulation can be observed. Once the simulation is killed the `./getlogs.sh` script can be executed. This script saves log files in `airforce/logs`, as seen in Figure 47.

Overall, simulation be performed using the following steps:

- i) Start `mastercontroller.py` in a terminal.
- ii) Run `./testrun.sh 31 4 1` in another terminal to start the simulation.
- iii) Wait for simulation to complete.
- iv) Run `./testkill.sh 31 4 1` in another terminal to kill the simulation.
- v) After all processes have been killed, use `./getlogs 31 4 1` to write GNAT and PHY log files to `airforce/logs`.
- vi) Check `ref_Table_X.json` in `/logs` to check satellite activity at time slot X..

The following figures further elaborate the procedure for a simple simulation without opening any satellite terminals.

```
nprabhu@ubuntu:~/airforce/master_controller/src$ python3 master_controller.py

> Master Controller
> Start the socket server of the Master Controller ...

> Reference table has not been initialized!
> Please load the json file first!

> Type a command to continue (e.g. ) or
> Press ENTER to terminate the process and quit
> Type help or h to show the help message
% > Required link between SAT 40105 and 40730 has been set! Trying to use current PHY to connect to other SAT!
> Required link between SAT 28129 and 27663 has been set! Trying to use current PHY to connect to other SAT!
> Required link between SAT 37753 and 25933 has been set! Trying to use current PHY to connect to other SAT!
> Required link between SAT 29486 and 40534 has been set! Trying to use current PHY to connect to other SAT!
> Required link between SAT 41328 and 36585 has been set! Trying to use current PHY to connect to other SAT!
> Required link between SAT 26605 and 36585 has been set! Trying to use current PHY to connect to other SAT!
> Required link between SAT 24876 and 35752 has been set! Trying to use current PHY to connect to other SAT!
> Required link between SAT 26605 and 29486 has been set! Trying to use current PHY to connect to other SAT!

:: Operation now in progress
connect to remote server.. res=0x0, errno=115
SRV_REMOTE initiate connect on DC1
connect protocol...connect DC1
connect to remote server.. res=0x0, errno=115
SRV_REMOTE initiate connect on DC1
connect protocol...connect DC1
connect to remote server.. res=0x0, errno=115
SRV_REMOTE initiate connect on DC1
connect protocol...connect DC1
connect to remote server.. res=0x0, errno=115
SRV_REMOTE initiate connect on DC1
connect protocol...connect DC1
remote connect..res=-1, errno=115
:: Operation now in progress
remote address = 192.168.2.2
Direct accept for interface 0 from 2.0.216.194
auto=1, opMode=3
SRV_LOCAL initiate connect on DC1
connect protocol...connect DC1
connect to remote server.. res=0x0, errno=115
SRV_REMOTE initiate connect on DC1
connect protocol...connect DC1
connect to remote server.. res=0x0, errno=115
SRV_REMOTE initiate connect on DC1
connect protocol...connect DC1
connect to remote server.. res=0x0, errno=115
SRV_REMOTE initiate connect on DC1
connect protocol...connect DC1
```

Figure 45 :Start the master controller


```

{
  "SatDef": {
    "1": {
      "TrafficMode": {
        "MsgsPerSec": 1,
        "DstID": 1,
        "Action": 1,
        "MsgID": 1,
        "MsgModel": 1,
        "StartTime": 5,
        "MsgSize": 125,
        "RepeatCount": 1000
      },
      "phyStats": {
        "1": {
          "phyID": 4,
          "phyType": 3,
          "phyStatus": "up",
          "dstIdx": 4,
          "dstID": 21,
          "isOccupied": true
        },
        "2": {
          "phyID": 5,
          "phyType": 3,
          "phyStatus": "up",
          "dstIdx": 4,
          "dstID": 6,
          "isOccupied": true
        },
        "3": {
          "phyID": 2,
          "phyType": 3,
          "phyStatus": "up",
          "dstIdx": 1,
          "dstID": 22,
          "isOccupied": true
        },
        "4": {
          "phyID": 3,
          "phyType": 3,
          "phyStatus": "up",
          "dstIdx": 2,
          "dstID": 29,
          "isOccupied": true
        },
        "5": {
          "phyID": 1,
          "phyType": 1,
          "phyStatus": "up",
          "dstIdx": 0,
          "dstID": 0,
          "isOccupied": false
        }
      }
    }
  }
}

```

Figure 46: Reference table for 32 nodes after 10 seconds of simulation

In the reference table, we can observe the connected satellites and the PHYs through which they are connected. For example, in Figure 46, Satellite 1, PHY 1 is connected with Satellite 21 (dstID) through its PHY 4 (dstIdx).

```

nprabhu@ubuntu:~/airforce/logs$ ls
gnat_gs1.log  gnat_sat4.log  phy_sat14_2.log  phy_sat20_3.log  phy_sat26_1.log  phy_sat3_3.log
gnat_sat10.log  gnat_sat5.log  phy_sat14_3.log  phy_sat20_4.log  phy_sat26_2.log  phy_sat3_4.log
gnat_sat11.log  gnat_sat6.log  phy_sat14_4.log  phy_sat21_1.log  phy_sat26_3.log  phy_sat4_1.log
gnat_sat12.log  gnat_sat7.log  phy_sat1_4.log  phy_sat21_2.log  phy_sat26_4.log  phy_sat4_2.log
gnat_sat13.log  gnat_sat8.log  phy_sat15_1.log  phy_sat21_3.log  phy_sat27_1.log  phy_sat4_3.log
gnat_sat14.log  gnat_sat9.log  phy_sat15_2.log  phy_sat21_4.log  phy_sat27_2.log  phy_sat4_4.log
gnat_sat15.log  phy_sat10_1.log  phy_sat15_3.log  phy_sat2_1.log  phy_sat27_3.log  phy_sat5_1.log
gnat_sat16.log  phy_sat10_2.log  phy_sat15_4.log  phy_sat22_1.log  phy_sat27_4.log  phy_sat5_2.log
gnat_sat17.log  phy_sat10_3.log  phy_sat16_1.log  phy_sat22_2.log  phy_sat28_1.log  phy_sat5_3.log
gnat_sat18.log  phy_sat10_4.log  phy_sat16_2.log  phy_sat22_3.log  phy_sat28_2.log  phy_sat5_4.log
gnat_sat19.log  phy_sat11_1.log  phy_sat16_3.log  phy_sat22_4.log  phy_sat28_3.log  phy_sat6_1.log
gnat_sat1.log  phy_sat11_2.log  phy_sat16_4.log  phy_sat2_2.log  phy_sat28_4.log  phy_sat6_2.log
gnat_sat20.log  phy_sat11_3.log  phy_sat17_1.log  phy_sat23_1.log  phy_sat29_1.log  phy_sat6_3.log
gnat_sat21.log  phy_sat11_4.log  phy_sat17_2.log  phy_sat23_2.log  phy_sat29_2.log  phy_sat6_4.log
gnat_sat22.log  phy_sat1_1.log  phy_sat17_3.log  phy_sat23_3.log  phy_sat29_3.log  phy_sat7_1.log
gnat_sat23.log  phy_sat12_1.log  phy_sat17_4.log  phy_sat23_4.log  phy_sat29_4.log  phy_sat7_2.log
gnat_sat24.log  phy_sat12_2.log  phy_sat18_1.log  phy_sat2_3.log  phy_sat30_1.log  phy_sat7_3.log
gnat_sat25.log  phy_sat12_3.log  phy_sat18_2.log  phy_sat24_1.log  phy_sat30_2.log  phy_sat7_4.log
gnat_sat26.log  phy_sat12_4.log  phy_sat18_3.log  phy_sat24_2.log  phy_sat30_3.log  phy_sat8_1.log
gnat_sat27.log  phy_sat1_2.log  phy_sat18_4.log  phy_sat24_3.log  phy_sat30_4.log  phy_sat8_2.log
gnat_sat28.log  phy_sat13_1.log  phy_sat19_1.log  phy_sat24_4.log  phy_sat31_1.log  phy_sat8_3.log
gnat_sat29.log  phy_sat13_2.log  phy_sat19_2.log  phy_sat2_4.log  phy_sat31_2.log  phy_sat8_4.log
gnat_sat2.log  phy_sat13_3.log  phy_sat19_3.log  phy_sat25_1.log  phy_sat31_3.log  phy_sat9_1.log
gnat_sat20.log  phy_sat13_4.log  phy_sat19_4.log  phy_sat25_2.log  phy_sat31_4.log  phy_sat9_2.log

```

Figure 47: log files

```

master_controller.log (~/airforce/master_controller/log) - GVIM1 (on ubuntu)
File Edit Tools Syntax Buffers Window Help
06/08/2019 20:01:55.598 INFO: > Received data from ('192.169.0.21', 55100):
06/08/2019 20:01:55.598 INFO: > 3C 00 00 67 ED 00 0C 67 00 A8 C0 00 00 00 05 00 00 00 03
06/08/2019 20:01:55.598 INFO: > Sending 15 bytes data to 192.169.0.21:
06/08/2019 20:01:55.598 INFO: > 3D 00 00 67 ED 00 08 00 00 00 00 05 00 00 00 01
06/08/2019 20:01:55.605 INFO: > Received data from ('192.169.0.30', 55100):
06/08/2019 20:01:55.605 INFO: > 32 00 00 7F C7 00 10 00 00 7F C7 00 00 65 4D 00 00 00 05 00 00 00 00

06/08/2019 20:01:55.629 INFO: > Add link between SAT 32 PHY 1 and SAT 2 PHY 5
06/08/2019 20:01:55.649 INFO: > Sending 23 bytes data to 192.169.0.32:
06/08/2019 20:01:55.649 INFO: > 33 00 00 00 01 00 10 00 00 00 01 00 00 9F 1A 00 00 00 01 00 00 00 02

06/08/2019 20:01:55.649 INFO: > Sending 23 bytes data to 192.169.0.2:
06/08/2019 20:01:55.649 INFO: > 33 00 00 9F 1A 00 10 00 00 9F 1A 00 00 00 01 00 00 00 01 00 00 00 02

06/08/2019 20:01:55.672 INFO: > Add link between SAT 18 PHY 1 and SAT 5 PHY 1
06/08/2019 20:01:55.693 INFO: > Sending 23 bytes data to 192.169.0.18:
06/08/2019 20:01:55.693 INFO: > 33 00 00 65 4D 00 10 00 00 65 4D 00 00 93 79 00 00 00 02 00 00 00 02

06/08/2019 20:01:55.694 INFO: > Sending 23 bytes data to 192.169.0.5:
06/08/2019 20:01:55.694 INFO: > 33 00 00 93 79 00 10 00 00 93 79 00 00 65 4D 00 00 00 02 00 00 00 02

06/08/2019 20:01:55.719 INFO: > Add link between SAT 31 PHY 1 and SAT 28 PHY 1
06/08/2019 20:01:55.739 INFO: > Sending 23 bytes data to 192.169.0.31:
06/08/2019 20:01:55.739 INFO: > 33 00 00 8B A8 00 10 00 00 8B A8 00 00 7E 04 00 00 00 02 00 00 00 02

06/08/2019 20:01:55.740 INFO: > Sending 23 bytes data to 192.169.0.28:
1151,34 59%

```

Figure 48: Master controller log after running simulation for 10 seconds

For better debugging it is advisable to run the `testrun.sh` script starting from `sat2` by changing the start of the `for` loop for GNAT and MSGEN to '2'. Then two terminals can be used to open the GNAT and MSGEN of satellite 1 by attaching `sat1` and then GNAT and MSGEN containers as performed for the four node simulation. Then run the `./GNAT_SAT_Debug` and `./PHY-MGEN_Debug` scripts, respectively. Next, run the simulation in the other terminal and in the MSGEN terminal of satellite 1 type in '1'. At the prompt enter a message. This action will print all the route entries that support communication (i.e. all connected satellites) from satellite 1. A message can be sent to a selected satellite.

```

added ROUTE entry #127, addr=0x18010a0a,MAC=0xa2,0x39,0xe0,0xa0,0xd,0xe7
1
Enter Text: h
Select Destination:
NAT 0. 10.10.1.1
NAT 1. 10.10.1.1
NAT 2. 10.10.1.1
RTE 3. 10.10.1.4
RTE 4. 10.10.1.3
RTE 5. 10.10.1.21
RTE 6. 10.10.1.2
RTE 7. 10.10.1.32
RTE 8. 10.10.1.15
RTE 9. 10.10.1.23
RTE 10. 10.10.1.13
RTE 11. 10.10.1.30
RTE 12. 10.10.1.8
RTE 13. 10.10.1.25
RTE 14. 10.10.1.26
RTE 15. 10.10.1.12
RTE 16. 10.10.1.10
RTE 17. 10.10.1.22
RTE 18. 10.10.1.18
RTE 19. 10.10.1.5
RTE 20. 10.10.1.1
RTE 21. 10.10.1.22
RTE 22. 10.10.1.29
RTE 23. 10.10.1.19
RTE 24. 10.10.1.31
RTE 25. 10.10.1.9
RTE 26. 10.10.1.28
RTE 27. 10.10.1.27
RTE 28. 10.10.1.17
RTE 29. 10.10.1.16
RTE 30. 10.10.1.14
RTE 31. 10.10.1.20
RTE 32. 10.10.1.6
RTE 33. 10.10.1.20
RTE 34. 10.10.1.28
RTE 35. 10.10.1.8
RTE 36. 10.10.1.19
RTE 37. 10.10.1.30
RTE 38. 10.10.1.24
RTE 39. 10.10.1.21
RTE 40. 10.10.1.15
RTE 41. 10.10.1.3
RTE 42. 10.10.1.2
RTE 43. 10.10.1.4
RTE 44. 10.10.1.2
RTE 45. 10.10.1.32
RTE 46. 10.10.1.15
RTE 47. 10.10.1.23
RTE 48. 10.10.1.13
RTE 49. 10.10.1.30

```

Figure 49: Route entries in MSGEN of sat1 in a 32 node simulation

Further usage

After generation of a custom multi-node system, it can be used for testing or to create custom connections.

Using custom code:

Presently, default code written to support inter-satellite communication is executed in Docker and transferred into the containers by default by scripts. This code can be replaced by custom code to create a customized simulation environment.

Using **sudo docker cp [options] src_path|- container:dest_path**, custom code can be transferred and run within each container. Figure 50 shows how to transfer a directory with files into a container “sat1”. The container must be running to be able to run this command.

```
nprabhu@ubuntu:/home/guest/AirForce$ ls
CustomCode  GNAT_Docker_Latest  scripts
nprabhu@ubuntu:/home/guest/AirForce$ cd scripts
nprabhu@ubuntu:/home/guest/AirForce/scripts$ sudo docker cp ../CustomCode sat1:/home
nprabhu@ubuntu:/home/guest/AirForce/scripts$ sudo docker exec sat1 /bin/sh -c "cd home; ls"
CustomCode
GNAT_Docker
add_eth.sh
env_clear.sh
env_setup.sh
```

Figure 50: Copying files to satellite containers

Once transferred the code can be executed using the terminal i.e ./code.

Using a container as a terminal:

A terminal for each container can be started using the following command:

Sudo docker exec -it "id of running container" bash

```
__@ubuntu:/home/guest/AirForce$ sudo docker exec -it sat1 bash
[sudo] password for nprabhu:
root@de0fb29ee4f2:/#
```

Figure 51: Starting a bash shell for satellite container

This provides a terminal for the specific container and can be used for running code or creating new connections.

Setting custom IP address:

The custom IP address can be set to any of the interfaces in the system.

For example: To change the IP of a GNAT interface in satellite 1, run the following command:

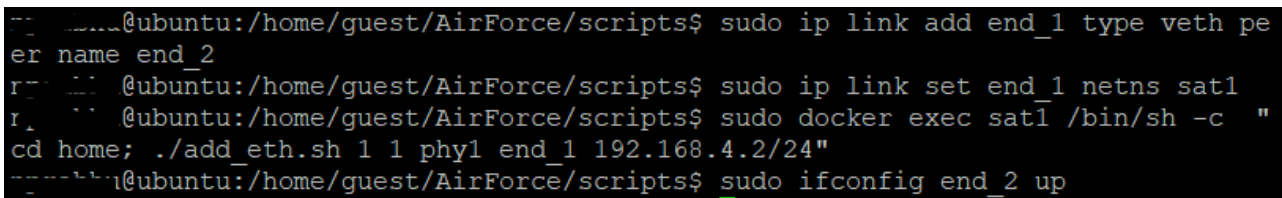
```
sudo docker exec sat1 /bin/sh -c "cd home; ./add_eth.sh 1 0 gnat ctrl.gnat 192.169.0.1/24"
```

This command utilizes the `add_eth.sh` script to change the IP address.

Creating a custom veth link to a nested container, run the following commands from the scripts directory :

For example, to set a custom link to `phy1` in `sat1`,

```
sudo ip link add end_1 type veth peer name end_2 // create veth pair  
sudo ip link set end_1 netns sat1 // set one end in sat1 namespace  
sudo docker exec sat1 /bin/sh -c "cd home; ./add_eth.sh 1 1 phy1 end_1 192.168.4.2/24" // set one  
end in phy1 namespace and set IP 192.168.4.2  
sudo ifconfig end_2 up // turn up the end on host
```

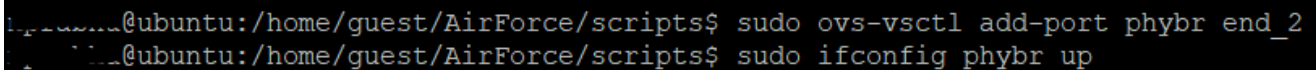


```
@ubuntu:/home/guest/AirForce/scripts$ sudo ip link add end_1 type veth pe  
er name end_2  
r@ubuntu:/home/guest/AirForce/scripts$ sudo ip link set end_1 netns sat1  
r@ubuntu:/home/guest/AirForce/scripts$ sudo docker exec sat1 /bin/sh -c "  
cd home; ./add_eth.sh 1 1 phy1 end_1 192.168.4.2/24"  
@ubuntu:/home/guest/AirForce/scripts$ sudo ifconfig end_2 up
```

Figure 52: Setting a new link interface inside container

This end can be connected to an OVS bridge (e.g. `phybr`):

```
sudo ovs-vsctl add-port phybr end_2  
sudo ifconfig phybr up
```



```
@ubuntu:/home/guest/AirForce/scripts$ sudo ovs-vsctl add-port phybr end_2  
@ubuntu:/home/guest/AirForce/scripts$ sudo ifconfig phybr up
```

Figure 53: Adding other link end on external ovs

Use the `exec` command to check the new interface configuration in `phy1` as shown:

```
sudo docker exec sat1 /bin/sh -c "sudo docker container exec phy1 /bin/sh -c "ifconfig""
```

```

root@ubuntu:/home/guest/AirForce/scripts$ sudo docker exec sat1 /bin/sh -c "sudo docker container exec phy1 /bin/sh -c "ifconfig"
direct.1c: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.0.100 netmask 255.255.255.0 broadcast 0.0.0.0
    ether ba:f8:2d:49:45:b8 txqueuelen 1000 (Ethernet)
    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

direct.1d: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.0.101 netmask 255.255.255.0 broadcast 0.0.0.0
    ether ba:0c:ee:36:8a:ea txqueuelen 1000 (Ethernet)
    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

end_1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.4.2 netmask 255.255.255.0 broadcast 0.0.0.0
    ether 12:57:af:1c:84:29 txqueuelen 1000 (Ethernet)
    RX packets 8 bytes 648 (648.0 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

```

Figure 54: Interface added in the nested container

We can see that our end_1 interface is set in phy1 with the designated IP address. The other end on the Open vSwitch can be identified using the following command:
sudo ovs-vsctl show

Figure 55 shows end_2 added as a port on the phybr bridge.

```

Bridge phybr
  Port phybr
    Interface phybr
      type: internal
  Port "sat4.phy1"
    Interface "sat4.phy1"
  Port "sat3.phy1"
    Interface "sat3.phy1"
  Port "sat3.phy2"
    Interface "sat3.phy2"
  Port "sat1.phy2"
    Interface "sat1.phy2"
  Port "sat2.phy1"
    Interface "sat2.phy1"
  Port "sat4.phy2"
    Interface "sat4.phy2"
  Port "end_2"
    Interface "end_2"
  Port "sat2.phy2"
    Interface "sat2.phy2"
  Port "sat1.phy1"
    Interface "sat1.phy1"
ovs_version: "2.11.90"

```

Figure 55: Port added to the external ovs

There are several other customization possibilities which can be explored by examining the Docker documents page [9]. Our scripts cover almost every aspect of network creation. However, since the scripts assume a certain system structure, add/delete can be used to create other formations or set up a multi-node virtual network as required.

References

- [1] <https://docs.docker.com/install/linux/docker-ce/ubuntu/>
- [2] <http://docs.openvswitch.org/en/latest/intro/install/general/>
- [3] <https://aviationweek.com/technology/pentagon-s-combat-cloud>
- [4] https://hub.docker.com/_/docker
- [5] <https://www.virtualbox.org/manual/ch02.html>
- [6] <https://www.virtualbox.org/wiki/Downloads>
- [7] <https://www.putty.org/>
- [8] <https://github.com/jpetazzo/dind>
- [9] <https://docs.docker.com/get-started/part2/>
- [10] <https://blog.arturofm.com/install-terminator-terminal-emulator-in-ubuntu/>