

# *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](mailto:xuzhizhang@umass.edu)

Narendra Prabhu

Graduate Student

Department of Electrical and Computer Engineering

University of Massachusetts Amherst

[nprabhu@umass.edu](mailto:nprabhu@umass.edu)

Reconfigurable Computing Group  
Version 1.0

March 15, 2019

---

# Contents

---

<b>Introduction .....</b>	<b>3</b>
<b>Design .....</b>	<b>5</b>
<b>Structure .....</b>	<b>6</b>
<b>Prerequisites .....</b>	<b>8</b>
<b>Installation and usage instructions .....</b>	<b>9</b>
Install VirtualBox.....	9
Import Virtual Appliance .....	9
Using the Image .....	11
Connecting to SimulatorVM Image .....	14
Docker and OpenvSwitch Status .....	15
<b>Starting the Simulator .....</b>	<b>18</b>
Create a 4-node Simulator.....	18
Check Docker Containers.....	19
Check Interfaces .....	19
Check OpenvSwitch Configuration .....	20
Clearing the 4-node simulator .....	22
<b>Running the Simulator .....</b>	<b>23</b>
<b>Scripts Overview .....</b>	<b>29</b>
Internal Scripts.....	30
<b>Further Usage .....</b>	<b>31</b>
<b>References .....</b>	<b>35</b>
<b>Appendices .....</b>	<b>..</b>
Appendix A.....	36

---

## Introduction

---

The aim of this guide is to document how to set up an environment that can simulate a scalable network consisting of satellite components and switches. For this experimentation, a satellite is formed from multiple components. A switch is used to interconnect these components. Multiple satellites can be combined together in the simulation via an additional switch. Docker [1], a container-based virtualization platform and OpenvSwitch (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 scalable network of satellites/GPS systems connected to work in tandem. Docker containers are used to represent the satellite component and OVS is used to interconnect them to form a satellite. Each satellite is a nested Docker container including the component containers and the 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 [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.

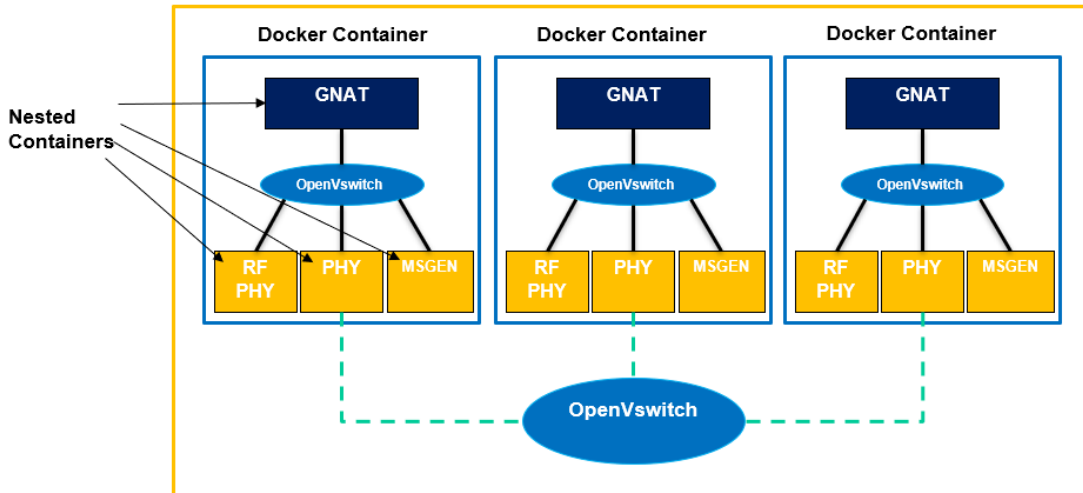


Figure 1: A 3 node system depicting the structure of the simulator

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 is 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 provided by IFT), parsing traffic flow information from Leidos, 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 a python code developed locally at UMass in the Reconfigurable Computing Laboratory, while the code for applications within the satellite i.e. the GNAT, PHY's and MSGEN is provided by Don Fronterhouse from PnP innovations.

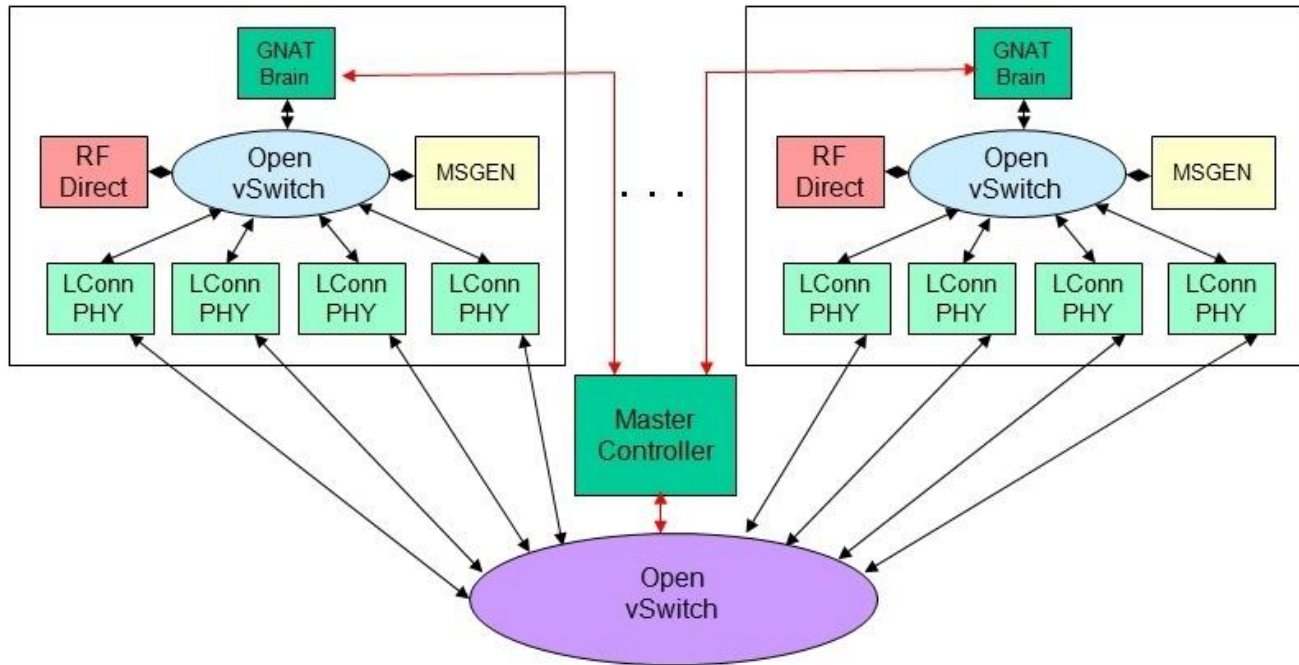


Figure 2: Illustrates the links between the components of a simple 2 satellite simulator

---

## *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, OpenvSwitch and Docker need to be installed within a Docker container itself. This raises 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 OpenVSwitch 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 plane 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 OpenVswitch using Veth links.

---

## Structure

---

Figure 3 shows internal satellite interfaces. Each black line is a Veth link. The blue boxes signify the nested container connected to an OpenvSwitch bridge (in beige). Here we demonstrate a satellite with two PHY's, an RFPHY and a PHY, a GNAT and an MSGEN. The GNAT communicates with its intra-satellite 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 PHY's 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, though 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 the host interfaces to allow for separation of control and data messages onto different interfaces each of which is assigned a different MAC address.

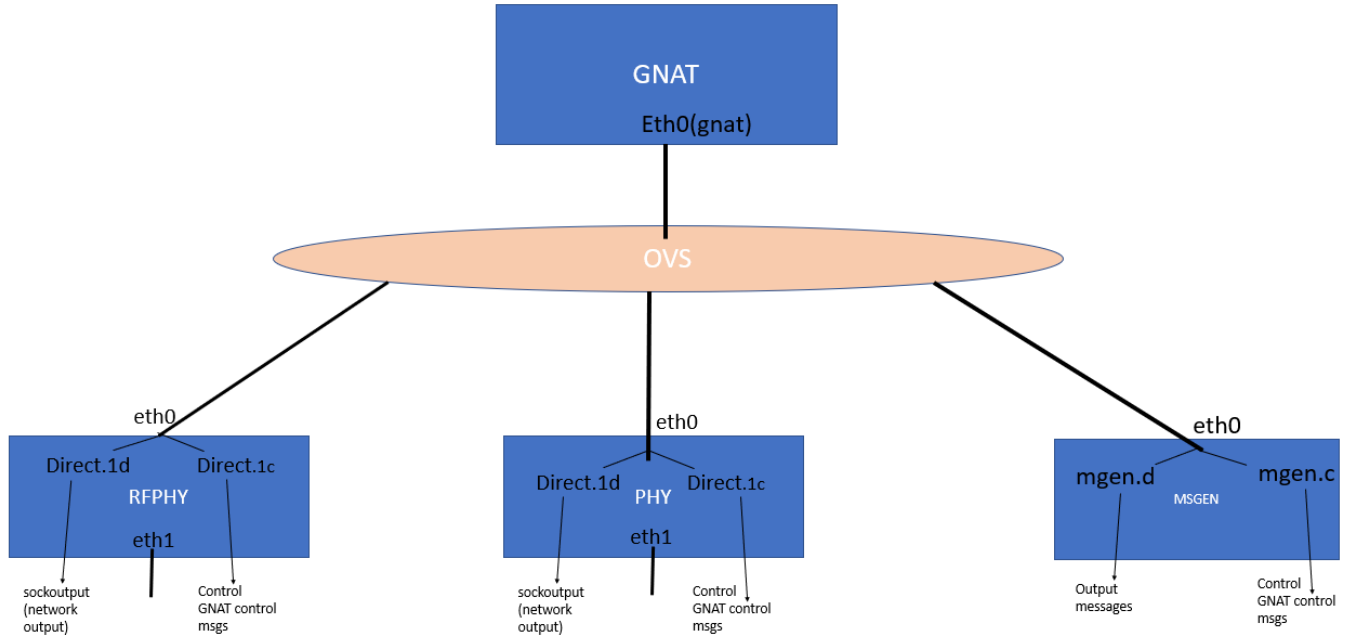


Figure 3: Interfaces inside a atellite node in the simulator

As an example, Figure 4 shows the overall connectivity for a multi-node satellite system:

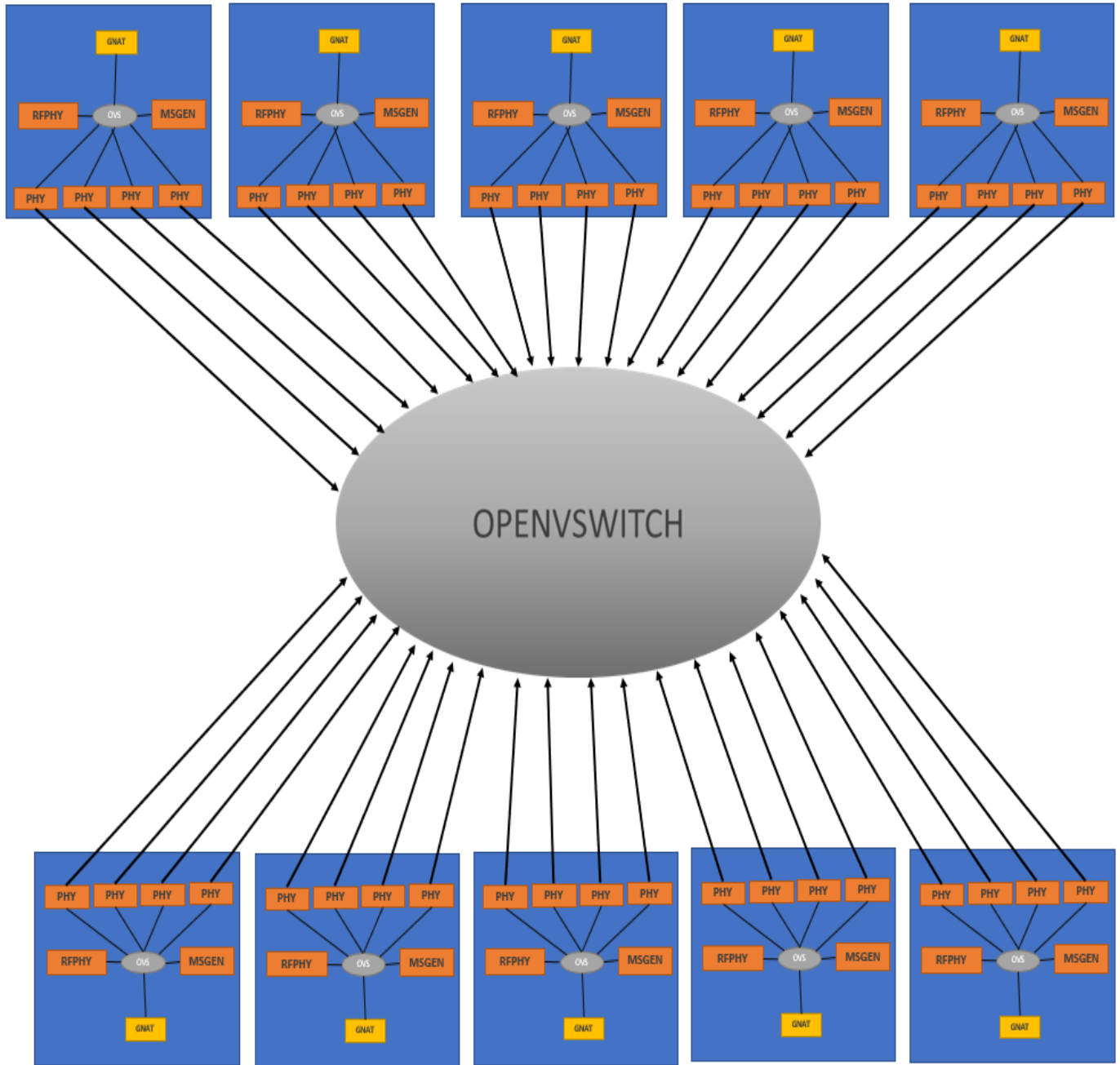


Figure 4: An example system that shows intersatellite links

---

## *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 OpenvSwitch installation. Thus, the VirtualBox image provided is built using the Ubuntu 16.04 version. 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 attached scripts (See Appendix A) generate the network simulator including Docker and Open vSwitch instances in the VirtualBox. The user must install VirtualBox on the system [5] to import the appliance, i.e. the provided VirtualBox image , and login through either a GUI or SSH using port forwarding, as described in the next section.



---

## *Installation and usage instructions*

---

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

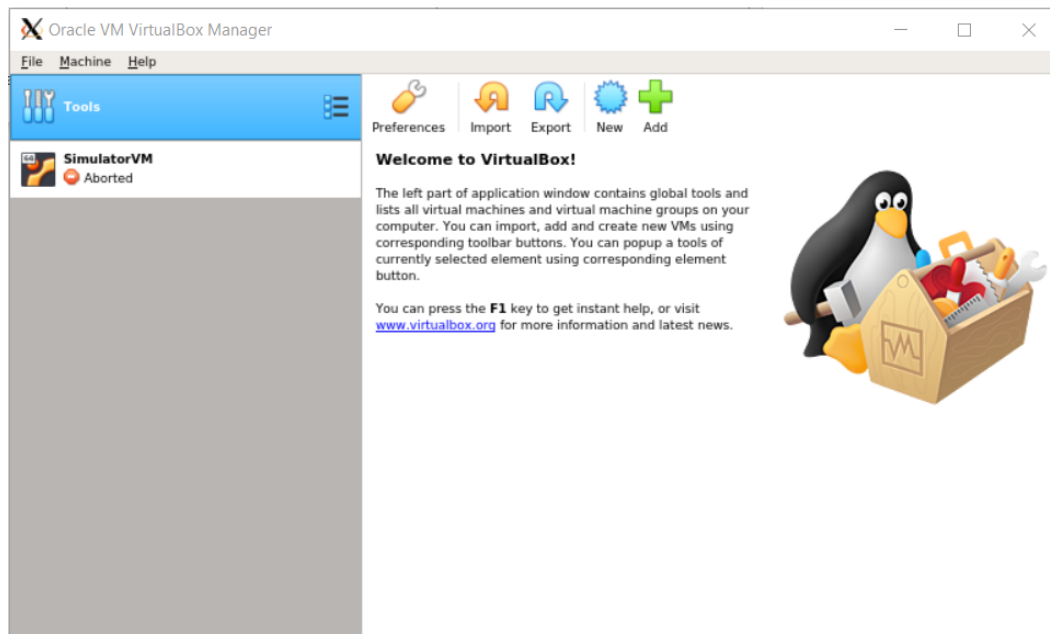
### 1) Install VirtualBox:

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 out as shown in figure 5.



*Figure 5: VirtualBox Manager screen*

### 2) Importing the VirtualBox appliance:

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

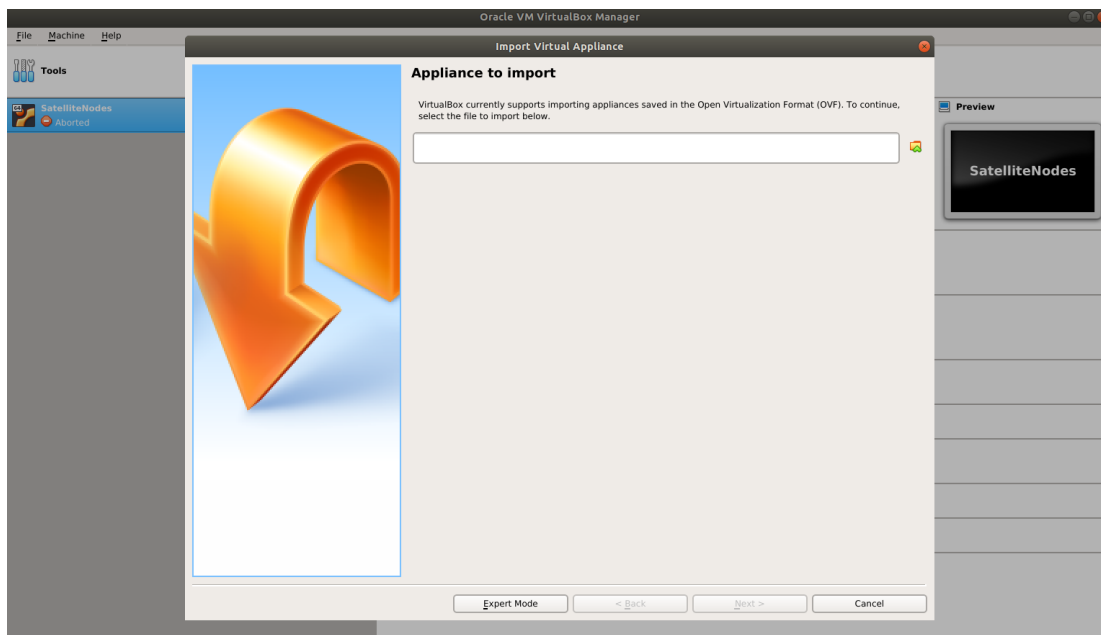


Figure 6: Appliance import screen

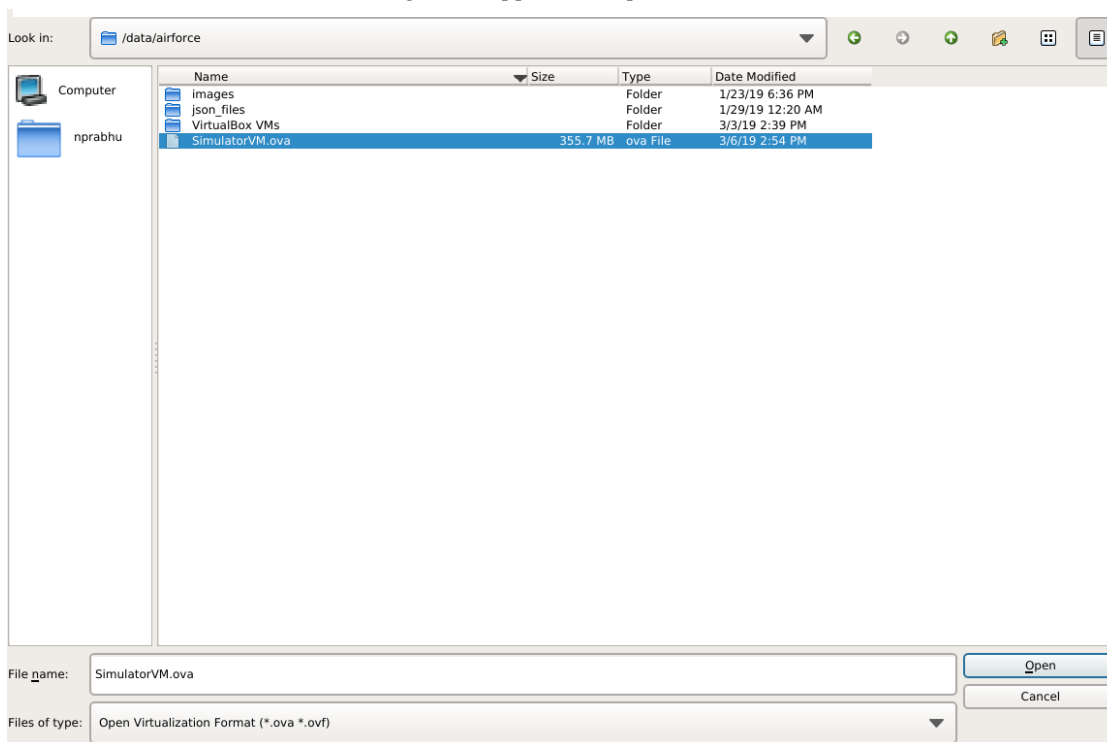
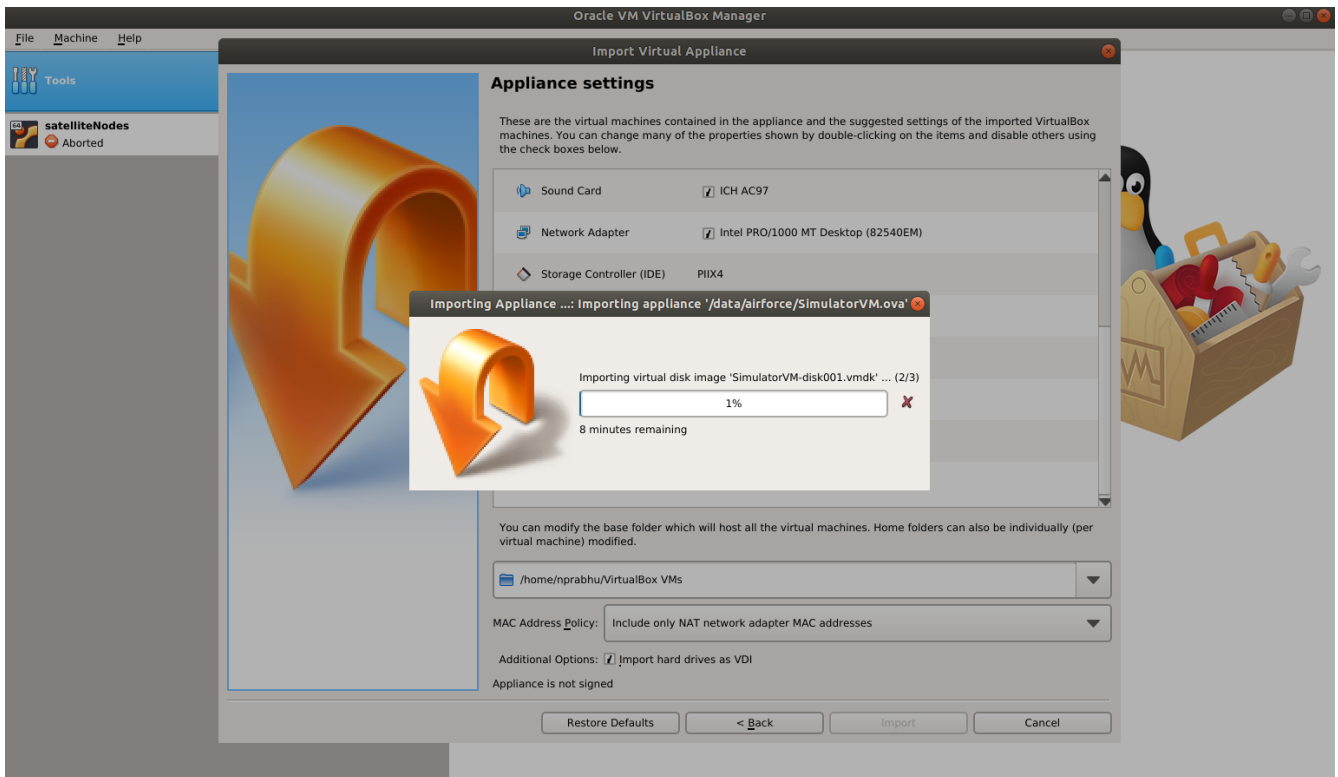


Figure 7: Loading the SimulatorVM.ova file



*Figure 8: 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.

### 3) 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 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.

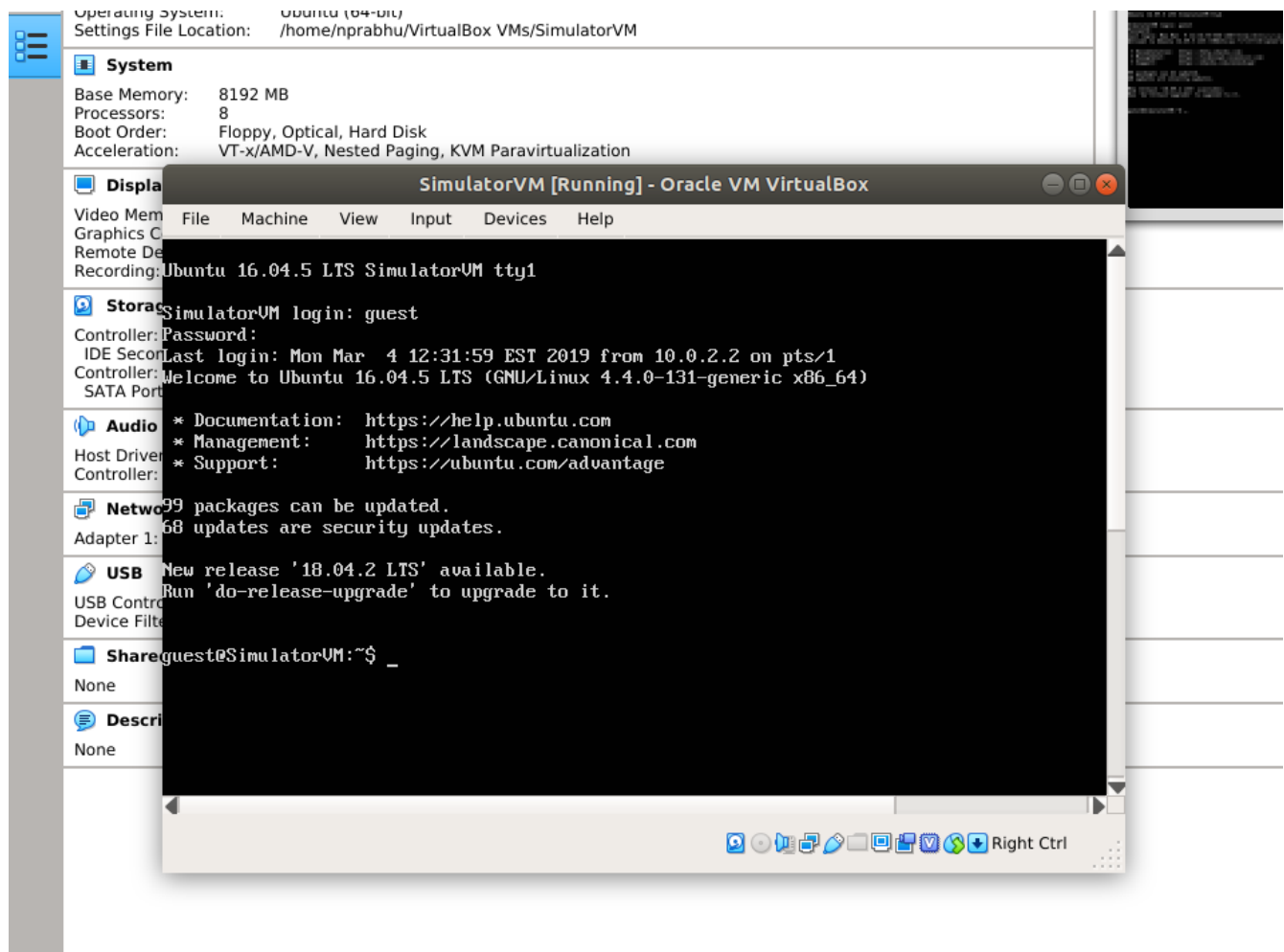


Figure 9: Logging into the VM

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 a 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 SimulatorVM.Ova image in the left pane. Right click and go to -> Settings in the menu at the top.

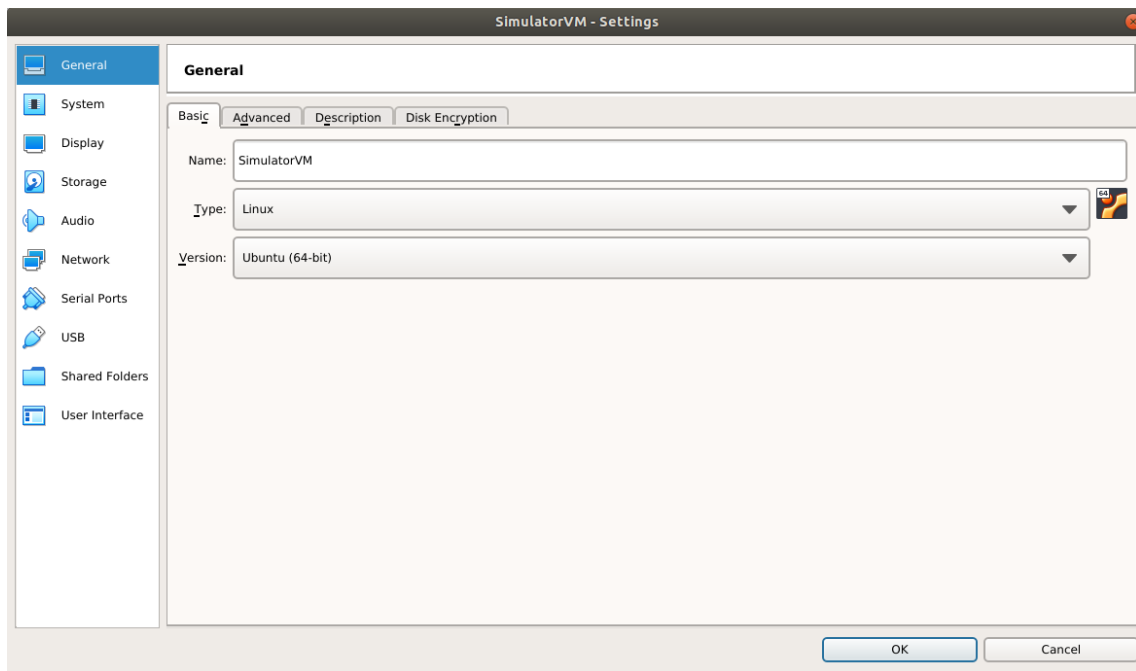


Figure 10: Changing network settings

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

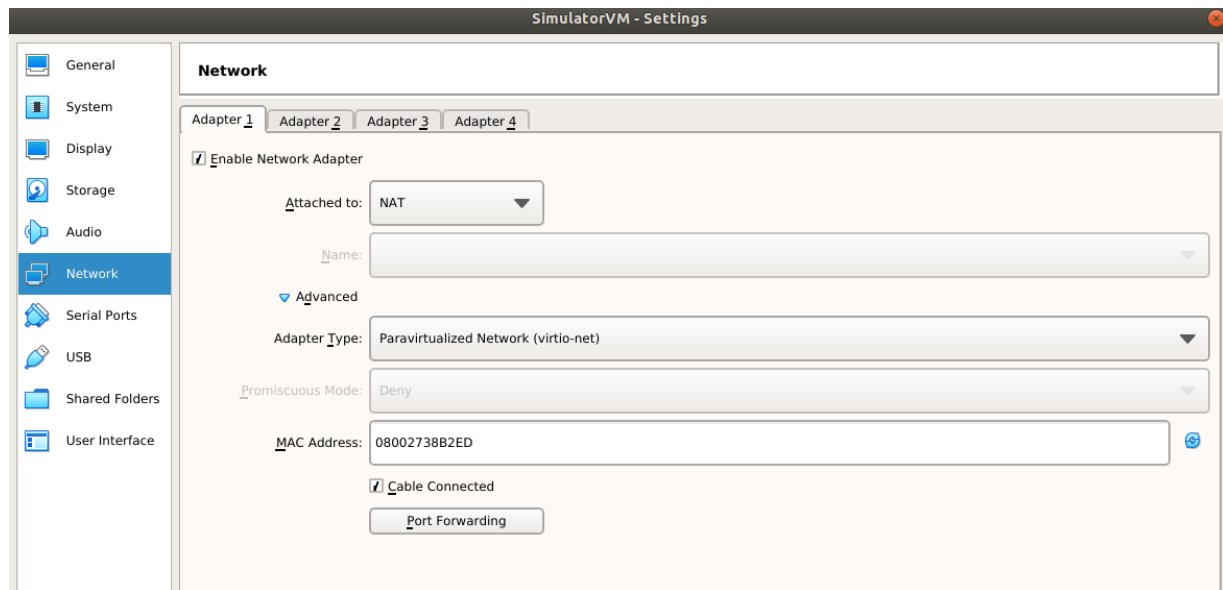


Figure 11: Adding Port Forwarding

If no existing rules 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 rule exists, leave it as it is.

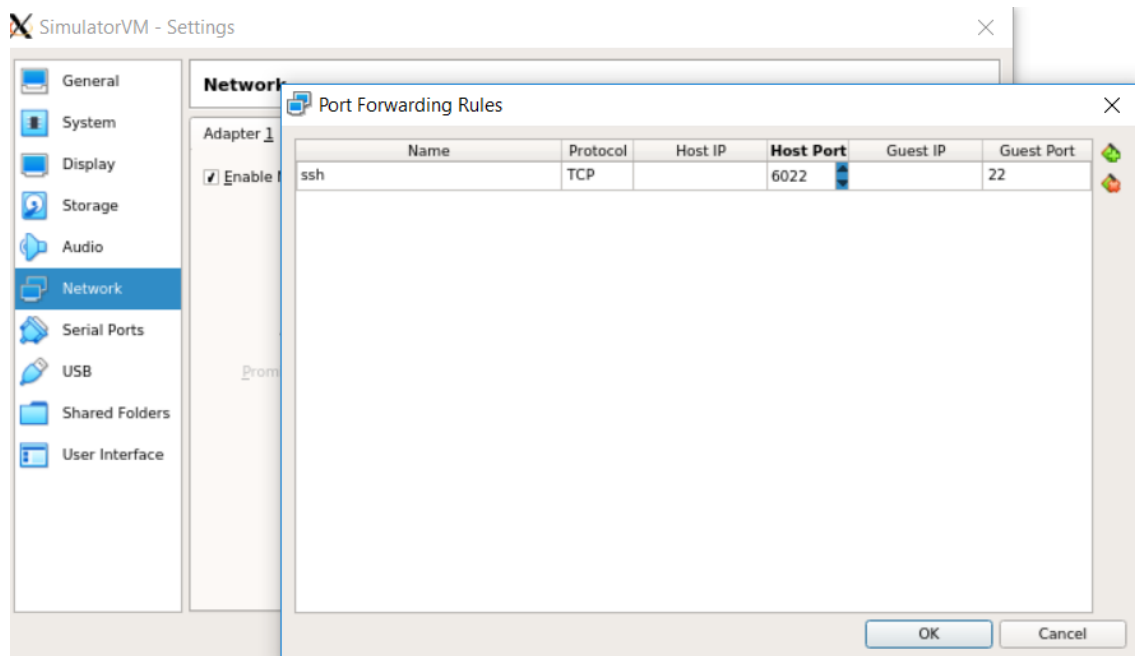


Figure 12: Adding port number and ssh forwarding rules

The GUI tends to be usually slow, so it is easier to use ssh connect in a terminal to interact with the VM. On a Ubuntu OS, open a terminal and use ssh as described in the next section.

#### 4) 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](#)**

```
[root@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/XW4EwKRDDnLA6No9e0kGgw.
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 13: 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 action will provide access to the terminal which can be used to build the simulator.

Now that a connection is made to the VM, it is possible to start building the simulator

## 5) Docker and OpenvSwitch status

Before building the simulator, it is necessary to verify that Docker and OpenVswitch are functional.

Once logged in, a simple Docker command is used to print the current status of the 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 14: Exited containers

This should list all the Docker containers (running or stopped because of -a flag). It 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 15: No container running

The command shows all the running containers. If there are no running containers, it will print nothing.

Now, the OpenvSwitch bridge is checked with the command:

**sudo ovs-vsctl show**

This should print out the node structure of all the OpenVSwitch bridges and ports if they exist. Otherwise it just prints out the version of the OpenvSwitch, as shown in Figure 16.

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

Figure 16: OpenVSwitch bridge structure

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

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

```
guest@SimulatorVM:~$ ls
GNAT_Docker  master_controller  scripts
```

Figure 17: Folders in the VM

Using “ls” will list all the files. The folder ‘scripts’ that we have added in this directory contains all the files that will be used to generate the simulator. Folder ‘master controller’ contains the master controller code and ‘GNAT\_Docker’ contains Don Fronterhouse’s code for the GNAT and PHYs.

```
guest@SimulatorVM:~/scripts$ ls
env_clear.sh  env_setup.sh  internal  ovs_net_clear.sh  ovs_net_setup.sh  sat_start.sh  sat_stop.sh
```

Figure 18: Scripts directory

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

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

Figure 19: Internal Scripts Directory

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

Script	Function
Sat_start.sh	Starts a container with a given satellite ID and No. of PHY’s. If container doesn’t exist, creates one.
Sat_stop.sh	Stops a container, given a satellite ID and No. of PHY’s
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
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 C code for each component of a satellite. Figure 20 shows the four main folders inside this folder. Each folder contains the respective code for each nested container.

```
guest@SimulatorVM:~$ ls
GNAT_Docker  master_controller  scripts
guest@SimulatorVM:~$ cd GNAT_Docker/
guest@SimulatorVM:~/GNAT_Docker$ ls
GNAT  PHY  PHY-Direct  PHY-MGEN  PHY-MRC
```

Figure 20: GNAT\_Docker file contents

The ‘master\_controller’ folder contains the Master Controller Python code developed at UMass. It also maintains the json visibility and traffic flow files provided by Leidos which are required for parsing the visibility information of the satellite to satellite communication. It also includes header/dependency files created for the master\_controller.py file. This is the main file which includes the master\_controller code.

```
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 21: Master\_controller folder contents

---

## Starting the simulator

---

A short example is presented here to demonstrate generation of a system using the scripts. A small case of 4 nodes is shown for brevity.

Starting containers: The scripts folder is in the VirtualBox in the folder /home/guest/Airforce. Change the directory to switch to this as the current directory. The Airforce folder also 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. In our case, to create a simulator with four satellites, where each satellite has four PHYs we will use the script named 'env\_setup.sh' that will setup our simulator. This script creates/starts the satellite system. It will start a network consisting of a specific number of satellites. The number needs to be provided as an argument to the script using a -s flag. The number of PHY's in each satellite is specified with a number following a -p flag. We take advantage of multiple processors to generate the satellite system. An integer argument following the flag -j determines 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. There might be a case that 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].

I)To create a 4-node system, run : **./env\_setup.sh -s 4 -p 2 -j 2**

A sample output is show below in Figure 22 (Note: Order is not important as we use parallel processing)

```
nprabhu@ubuntu:~/scripts$ ./env_setup.sh -s 4 -p 2 -j 2
Creating a network with 4 nodes!
Starting satellite 2
sat2
process_id=28949
  53 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 OVSDb managers
gnat
msgen
rfphy
phy1
phy2
Starting satellite 1
sat1
process_id=28948
  53 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 OVSDb managers
gnat
msgen
rfphy
phy1
phy2
Starting satellite 3
sat3
process_id=31418
  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 OVSDb managers
gnat
msgen
rfphy
phy1
phy2
```

Figure 22: Starting a 4 node simulator

```

Starting satellite 4
sat4
process_id=31417
  54 pts/0    00:00:00 dockerd
  * Starting ovsdb-server
  * system ID not configured, please use --system-id
  * Configuring Open vSwitch system IDs
  * Starting ovs-vswitchd
  * Enabling remote OVSDb 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 23: Creation and start of mainbridge and links on Satellite 4

i) Once we have created the containers, we can check if the Dockers 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 24: Running satellite containers

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

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

```

root@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 25: Interfaces of a nested container phy2

iii) To check the OpenvSwitch 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 26: OpenvSwitch bridge after four satellites created

II) If we need to clear the 4-node system in the future, run:  
**`./env_clear.sh -s 4 -p 2 -j 2`**

```
nprabhu@ubuntu:~/scripts$ ./env_clear.sh -s 4 -p 2 -j 2
Turn off all nodes! Clear the network setting!
gnat
msgen
rfphy
phy1
phy2
sat2
gnat
msgen
rfphy
phy1
phy2
sat1
gnat
msgen
rfphy
phy1
phy2
sat4
gnat
msgen
rfphy
phy1
phy2
sat3
Delete the ctrlbr bridge!
```

*Figure 27: Clearing the environment*

Clearing the environment will just stop all the containers and not remove them. However, all the links are deleted. Thus, when running `./env_setup.sh` again, the containers will be started and the links will be created again.

The number of satellites and PHYs in each satellite can be specified as arguments while running the `./env_setup.sh` to generate a custom multi-node system. For example, for a 32-node system with 4 PHYs in each run, use `./env_setup.sh -s 32 -p 4`. To clarify the functioning of the scripts we have documented their use with comments. A brief overview of general usage is provided below.

---

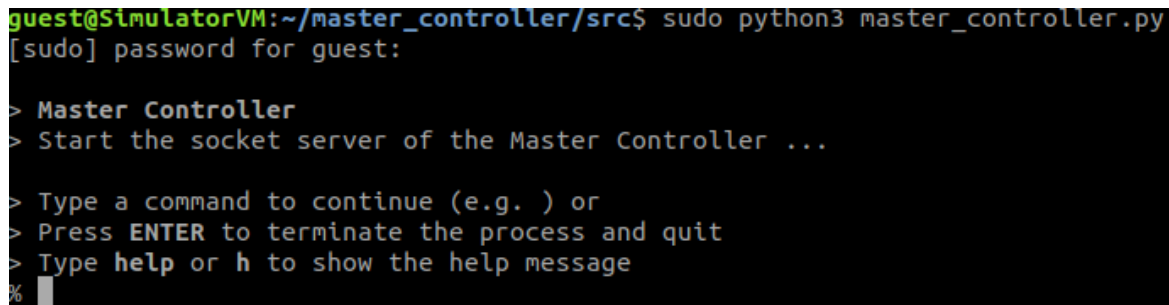
## Running the Simulator

---

Now that we have a running simulator, it can be used to run the code that we have designed for simulator testing. This code is in a developmental stage and represents the functionality of the satellite system in the Space Combat Cloud project. We begin by opening a terminal on the host system and connecting to the VM using ssh, as shown in previous steps. Once logged in, the master controller is started.

Change directory to the **master\_controller** directory and then to the **src** directory. Then run the following command:

**sudo python3 master\_controller.py**



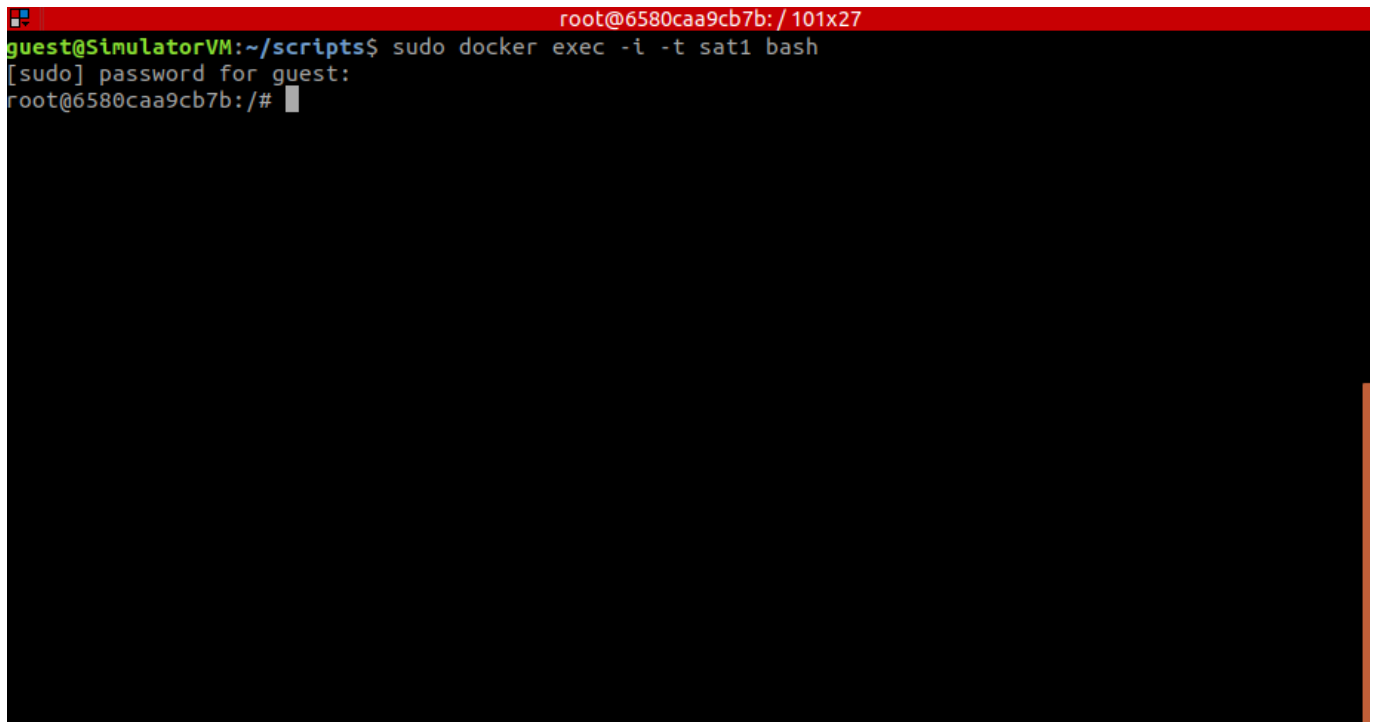
```
guest@SimulatorVM:~/master_controller/src$ sudo python3 master_controller.py
[sudo] password for guest:

> 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 28: Starting the master controller*

The next step is to attach the containers to separate the terminals for testing. We will test just a 2-node simulator where the data is exchanged between the satellites, sat1 and sat4. To start with we can attach a sat1 container to a terminal as shown in Figure 29. This action gives us access to the bash terminal of the satellite Docker container. If prompted, enter 'guest' as 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's password being entered. The prompt then changes to 'root@6580caa9cb7b: /#', indicating a successful root login into the 'sat1' container.

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

*Figure 29: Attach container sat1*

In this satellite another nested container, like GNAT, MSGEM or PHY can be attached.

For this, multiple terminals are required. Several terminals can be opened to perform this action. It is possible to login to the SimulatorVM on each terminal or a tool like terminator [9] can be used to create multiple terminals on a single screen.

Open 3 terminals for sat1 and then in each individual one, attach a different nested container.



The following figure shows the terminals for all the components in two satellites, sat1 and sat4. The top left corner terminal is used for the master controller, and the second from left is the GNAT of sat1. The top right corner terminal is the sat4 GNAT. The bottom terminals are phy1 and MSGEN of sat1 on the left and phy1 and MSGEN of sat4 on the right.

```

root@9b7b5f80e91: /
guest@SimulatorVM:~$ ls -l
total 12
-rwxr-xr-x 7 guest guest 4096 Mar  4 12:31 GNAT_D
-rwxr-xr-x 4 guest root  4096 Mar  4 17:51 master
controller
-rwxr-xr-x 3 guest root  4096 Mar 11 13:00 script
...
guest@SimulatorVM:~$ cd master_controller/
guest@SimulatorVM:~/master_controller$ cd src/
guest@SimulatorVM:~/master_controller/src$ ls
includes.py  master_controller.py  __pycache__  r
reference_table.py  set_link.sh
guest@SimulatorVM:~/master_controller/src$ python
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
[]

root@d76d470d93dd: /49x27
* 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 13:52:43 2019 from 10.0.2.2
guest@SimulatorVM:~$ cd scripts
guest@SimulatorVM:~/scripts$ clear
guest@SimulatorVM:~/scripts$ sudo docker exec -i -t sat1 bash
[sudo] password for guest:
root@6580caa9cb7b:/# sudo docker exec -i -t phy1
bash
root@d76d470d93dd:/#

root@9b7b5f80e91: /50x27
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 13:47:44 2019 from 10.0.2.2
guest@SimulatorVM:~$ ^
guest@SimulatorVM:~$ cd scripts
guest@SimulatorVM:~/scripts$ ls
env_clear.sh  env_setup.sh  internal  ovs_net_clea
r.sh  ovs_net_setup.sh  sat_start.sh  sat_stop.sh
guest@SimulatorVM:~/scripts$ clear
guest@SimulatorVM:~/scripts$ sudo docker exec -i -t sat1 bash
[sudo] password for guest:
root@6580caa9cb7b:/# sudo docker exec -i -t gnat bash
root@9b7b5f80e91:/#

root@9b7b5f80e91: /50x27
type: internal
Port "sat4.gnat"
Interface "sat4.gnat"
Port "sat2.gnat"
Interface "sat2.gnat"
Bridge phybr
Port "sat4.phy2"
Interface "sat4.phy2"
Port "sat3.phy1"
Interface "sat3.phy1"
Port "sat4.phy1"
Interface "sat4.phy1"
Port "sat1.phy2"
Interface "sat1.phy2"
Port "sat2.phy2"
Interface "sat2.phy2"
Port phybr
Interface phybr
type: internal
Port "sat2.phy1"
Interface "sat2.phy1"
Port "sat1.phy1"
Interface "sat1.phy1"
Port "sat3.phy2"
Interface "sat3.phy2"
ovs_version: "2.11.90"
guest@SimulatorVM:~/scripts$

root@ecfcd7d7a20: /49x27
* 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 13:52:54 2019 from 10.0.2.2
guest@SimulatorVM:~$ cd scripts
guest@SimulatorVM:~/scripts$ clear
guest@SimulatorVM:~/scripts$ sudo docker exec -i -t sat1 bash
[sudo] password for guest:
root@6580caa9cb7b:/# sudo docker exec -i -t msgen
bash
root@ecfcd7d7a20:/#

root@ecfcd7d7a20: /49x27
* 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 13:54:23 2019 from 10.0.2.2
guest@SimulatorVM:~$ clear
guest@SimulatorVM:~$ sudo docker exec -i -t sat4
bash
[sudo] password for guest:
root@54cea1363950:/# sudo docker exec -i -t phy1
bash
root@ecfcd7d7a20:/#

root@ed9038aec586: /50x27
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 13:54:05 2019 from 10.0.2.2
guest@SimulatorVM:~$ clear
guest@SimulatorVM:~$ sudo docker exec -i -t sat4 bash
[sudo] password for guest:
root@54cea1363950:/# sudo docker exec -i -t msgen
bash
root@ed9038aec586:/#

```

Figure 30: Attaching a different terminal for gnat, msgen and phy

Repeat these steps for sat4, by first attaching sat4 containers in three terminals and then attaching GNAT, MSGEN and PHY respectively in each individual container as follows:

**Sudo docker exec -it gnat bash**  
**Sudo docker exec -it msgen bash**  
**Sudo docker exec -it phy bash**

After attaching the six different terminals, change directory to each to run the respective codes. Since the compiled C code is copied to the respective directories, it is necessary to be in the appropriate directory to run the code, i.e. the GNAT folder for a GNAT, PHY-MGEN folder for MSGEN code and PHY-Direct for PHY code. For instance, in the terminal in which the GNAT was attached, change directory to **/home/GNAT\_Docker/GNAT**.

```

root@ab0af4d09f08:/home/GNAT_Docker# cd GNAT/
root@ab0af4d09f08:/home/GNAT_Docker/GNAT# ls
DockerSim.h  GNAT  GNAT.c  GNAT.h
root@ab0af4d09f08:/home/GNAT_Docker/GNAT# ./GNAT

```

Figure 31: cd to GNAT code folder

For Msgen, **/home/GNAT\_Docker/PHY-MGEN**.

```

root@9b77b5f80e91:/# cd /home/GNAT_Docker/PHY-MGEN/
root@9b77b5f80e91:/home/GNAT_Docker/PHY-MGEN# ls
MessageGenerator.c  PHY-MGEN  PHY_MGEN.h
root@9b77b5f80e91:/home/GNAT_Docker/PHY-MGEN# ./PHY-MGEN

```

Figure 32: cd to MSGEN folder

For PHY, **/home/GNAT\_Docker/PHY-Direct**.

```

root@d76d470d93dd:/# cd /home/GNAT_Docker/PHY-Direct/
root@d76d470d93dd:/home/GNAT_Docker/PHY-Direct# ls
DirectConnect.c  PHY_Direct.h
PHY-Direct      PHY_Direct_Init.sh
root@d76d470d93dd:/home/GNAT_Docker/PHY-Direct# ./PHY-Direct

```

Figure 33: cd to PHY folder

Repeat the steps for sat4. Now, the exe files in each of the containers are run to test the code.

These files were compiled and copied by scripts during the setup. To run the code, use the exe files in each container as shown in the above figures. In both GNAT containers, type **./GNAT**.

Next, execute the MSGEN containers, **./PHY-MGEN**. This step should be performed before PHY, as the code is designed so that the MSGEN should run before the PHYs. Lastly, run the PHY code in both PHY containers, **./PHY-Direct**.

The following figures show the messages that are exchanged when the C code is running between all the components. In Figure 34, the top left corner is the GNAT of sat1, the top right corner is MSGEN, and the bottom two are the PHY's. The exchanged messages can be seen in each terminal.

[illegible]

Figure 34: *sat1* components communicating

In Figure 35, the top left corner is the GNAT of sat4, the top right corner is MSGEN, and the bottom two figures are the PHYs. We can also see the messages sent across from the MSGEN of one satellite to another. The NAT and routing table can be seen in the MSGEN code, indicating its own GNAT IP address and the other GNAT address. i.e, the sat4 GNAT has address 10.10.1.4 and the other end has address 10.10.1.1.

<pre> root@6805edd204bd: /home/GNAT_Docker/GNAT 57x32 MC IP address=0xfe00a9c0 linkCost..X(Mbps) = 100.000000, Y = 1000.900901 added ROUTE entry #0, addr=0xa0a0101,MAC=0x2,0x66,0xf7,0x 61,0x49,0xab route #0, addr=0xa0a0101, cost=0x3e8, MAC=0x2,0x66,0xf7,0 x61,0x49,0xab buildMessage...len=17 srcIP=0x0, dstIP=0x3200a8c0 hdr bytes=8e c0 c6 2f 3c ce 0 0 0 0 0 8 0 45 0 0 2d f 7 9 40 0 40 11 6a 6d 0 0 0 0 c0 a8 0 32 c0 65 75 3a 0 19 7f b 7 0 0 0 1 1 1 a outputMessage...resp=59 buildMessage...len=17 srcIP=0x0, dstIP=0x6400a8c0 hdr bytes=a6 f7 eb df b8 33 0 0 0 0 0 8 0 45 0 0 2d f 7 9 40 0 40 11 6a 3b 0 0 0 0 c0 a8 0 64 c0 65 75 3a 0 19 7e d9 7 0 0 0 1 1 1 a outputMessage...resp=59 added route for remGA=0xa0a0101 through PHYcommIP=0x6500a 8c0 received status from 192.168.0.100, commIP = 192.168.0.10 1, commMAC=2:66:f7:61:49:ab, bitRate=100000000, devOP=2, remote GA=1.1.10.10, device status=1 end of switch...2 wait for message from a PHY GNAT network message...msgLen=0x4b, vlanID=0x0 Dest MAC=0:0:0:0:0:0...Source MAC=5e:d3:3e:60:2c:cc Src IP 0.0.0.0 Dst IP 4.1.10.10 Layer-4 protocol 17 Source, Dest ports 51100,30020 </pre>	<pre> root@ed9038aec586: /home/GNAT_Docker/PHY-MGEN 57x32 0x2,0x66,0xf7,0x61,0x49,0xab,0xe8,0x3,0x33,0x2,0x0,0x0,0x 0,0x0,0x0,0x0,0x0,0x1,0x0,0x0,0x0,0x0, added ROUTE entry #0, addr=0xa0a0101,MAC=0x2,0x66,0xf7,0x 61,0x49,0xab 1 Enter Text: Hello Select Destination: NAT 0. 4.1.10.10 RTE 1. 1.1.10.10 1 ....i=1 destination IP=0xa0a0101, port=30020 srcIP=0x0 dstIP=0xa0a0101 initial destIP=0xa0a0101, destPort=17525, destMAC=0:0:0:0 :0:0 i,NATtable[i].addr=0xa0a0104,NATtable[i].port=0,NATtable[i ].hopAddr=0x0,NATtable[i].hopPort=30020 nROUTE=1, destIP=0xa0a0101 i=0,ROUTEtable[i].addr=0xa0a0101,ROUTEtable[i].hopCost=10 00,ROUTEtable[i].hopMAC=2:66:f7:61:49:ab etherframe=2 66 f7 61 49 ab d6 36 98 f9 22 63 8 0 45 0 0 21 27 d9 40 0 40 11 7 e9 0 0 0 0 1 1 a a c7 9c 75 44 0 d 94 16 48 65 6c 6c 6f before sendto outputMessage...resp=47 message sent.. Enter message type: 1. Text 2. Binary data 3. Repeated binary </pre>
<pre> root@c7ecfd7d7a20: /home/GNAT_Docker/PHY-Direct 57x32 x0,0x0,0x0,0x0,0x0,0x1,0x0,0x0,0x0,0x0, added ROUTE entry #0, addr=0xa0a0101,MAC=0x2,0x66,0xf7,0x 61,0x49,0xab hopMAC == myMAC...don't send to the other end...  read header from direct iface=0, bytes=61,bIn=0x1, 0x0, 0x0, 0x3d len=61, data = aa,2a,61,c3,a4,bf,5e,d3,3e,60,2c,cc,8,0,45 ,0,0,21,27,d9,40,0,40,11,4,e9,0,0,0,0,4,1,a,a,c7,9c,75,44 ,0,d,91,16,48,65,6c,6c,6f,0,0,0,18,0,0,0,0,0,0,0,0,0,0,0, received 61 bytes on DC1..msgType=1 initial destIP=0xa0a0104, destPort=17525, destMAC=aa:2a:6 1:c3:a4:bf i,NATtable[i].addr=0xa0a0104,NATtable[i].port=0,NATtable[i ].hopAddr=0x0,NATtable[i].hopPort=30020 2 final destIP=0xa0a0104, destPort=0x4475, destMAC=0:0:0:0: 0:0 outputMessage..iface=1, len=61 bytes=0 0 0 0 0 5e d3 3e 60 2c cc 8 0 45 0 0 21 27 d9 40 0 40 11 e9 4 0 0 0 0 4 1 a a c7 9c 75 44 0 d 16 79 48 65 6c 6c 6f 0 0 0 18 0 0 0 0 0 0 0 0 0 Dst MAC: 02:66:f7:61:49:ab Src MAC: d6:36:98:f9:22:63 Src IP: 00:00:00:00, Dst IP: 01:01:0a:0a i=1, mac=ab, ip=65 sendDirect..iface=0,len=61,bOut=0x1,0x0,0x0,0x3d data = 2,66,f7,61,49,ab,d6,36,98,f9,22,63,8,0,45,0,0,21,2 7,d9,40,0,40,11,7,e9,0,0,0,0,1,1,a,a,c7,9c,75,44,0,d,94,1 6,48,65,6c,6c,6f,67,0,0,44,0,6,0,8,0,0,0,0,0,0,0,0,0,0,0, sent 61 bytes using socket=c send direct network message..len=61 </pre>	<pre> root@92668c6783bd: /home/GNAT_Docker/PHY-Direct 57x32 added ROUTE entry #0, addr=0xa0a0103,MAC=0x9e,0x53,0x11,0 x7,0xad,0x85 ^C root@75c3029e8032: /home/GNAT_Docker/PHY-MGEN# ^C root@75c3029e8032: /home/GNAT_Docker/PHY-MGEN# root@9c8449 2650e4:/# + src git:(develop_mc_ovs) x ssh rcglabserver xuzhi@keb302.ecs.umass.edu's password: Last login: Wed Mar 13 15:08:45 2019 from natp-128-119-20 2-134.wireless.umass.edu + ~ ssh -p 6022 guest@127.0.0.1 guest@127.0.0.1's password: Welcome to Ubuntu 16.04.5 LTS (GNU/Linux 4.4.0-131-generi c 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 15:09:32 2019 from 10.0.2.2 guest@SimulatorVM:~\$ sudo docker exec -i -t sat4 bash [sudo] password for guest: root@54ceal363950:/# docker exec -it phy2 bash root@92668c6783bd:/# cd /home/GNAT_Docker/PHY-Direct/ root@92668c6783bd:/home/GNAT_Docker/PHY-Direct# </pre>

Figure 35: Sat4 components communicating

---

## *Scripts overview*

---

### **env\_setup.sh**

This script is used to create/start the network consisting of satellites and OpenVswitches. Inputs are provided to this script with flags -s, -p and -j. The flagged inputs -s, -p and -j are the number of satellites (-s), number of PHYs in each satellite (-p), and number of parallel jobs that can be run (-j), respectively. The number of jobs can match the number of cores that the VirtualBox can use to parallelly set up the satellites in the simulator. If not provided, the satellites will start sequentially on a single core. The script calls sat\_start.sh as many times as the number of satellites that is specified after the -s argument. The script also sets up an OpenVswitch to connect all satellites using the Master Controller.

### **env\_clear.sh**

We use this script to clear the environment of existing connections and Docker containers. The script takes three arguments -s, -p and -j for number of satellites (-s), number of PHYs in each satellite (-p) and job number (-j). If job number is not provided, then all satellites are stopped one by one. If there are several jobs, then each clears all the satellites in each job in parallel, followed by the clearing of the OpenVswitch structure.

### **sat\_start.sh**

This script creates/starts a container and is repeatedly called by the env\_setup.sh to implement the number of satellite containers specified by the user. Input arguments SatID (satellite ID) and PHY\_NUM (number of PHYs) must be provided to generate a satellite container. Each Docker container is given a sat(ID) as a name, where ID is an integer from 1...N. If the user tries to create more nodes than the number that already exist in the VirtualBox, new containers are created using the jpetazzo/dind image [7], an image which enables the creation of nested Docker containers.

Several dependencies are required for the execution and testing of the containers. Networking tools net-tools, iproute2, ping, and tcpdump are used for setting links, viewing and editing network devices (e.g., Veth interfaces), and communication testing. These tools are installed by the scripts in each container, followed by Open vSwitch installation. Each container needs a GNAT Brain, MSGEN, RFPHY and several PHY containers. The number of PHYs is defined by an input parameter. All software packages and tools installed in a satellite container are also installed in each nested container (e.g, python3, vim, net-tools). Vim is a text editor that can be used for code editing. The OpenVswitch bridge required to connect all components together (br0) is then created by the scripts.



The files required for internal setup of the container (i.e. creating links between nested containers), are copied to the Docker satellite containers. Script `env_setup.sh` creates an internal OpenVswitch and sets up connections between the nested containers by adding Veth interfaces. If the container is created for the first time, i.e. if the user needs to create more containers than already exist, then new internal containers are created and the process ID of each is saved. If the containers exist, they are started, and the process ID is fetched. The script also periodically checks to see if the Docker daemon is running, since we faced a few bugs which caused the daemon to switch off by itself.

### **sat\_stop.sh**

This script is called by `env_clear.sh` to clear the execution environment for a container. It calls the script `env_clear.sh` for any internal containers if the container is nested. Finally, the script stops execution of the container.

### **ovs\_net\_setup.sh**

This script takes the number of satellites as input to create a connection between the GNAT in each satellite and the Master Controller. It also creates connections between the Master Controller and the OpenVswitch which forms connections between the PHYs on the satellites. A `ctrlbr` (OpenVswitch) is created for this setup. To create links, virtual ethernet links provided by the default Linux network tools are used. This requires an exclusive namespace for each container. First, a link with two ends is created. One end is inserted into the satellite container and then the `add_eth.sh` script is called upon to connect the nested GNAT container. The other end is set as a port on the `ctrlbr`. The PHYs of each satellite are then connected to a single master OpenVswitch (`phybr`) bridge with distinct port number (e.g., `sat1.phy1`) using Veth links. Each link in a satellite is `eth1` with IP `192.168.2.1/24`.

### **ovs\_net\_clear**

Used to clear the setup created by `ovs_net_setup.sh`, by deleting the `ctrlbr`, `phybr` and the virtual link in all satellites.

## **Internal scripts**

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

### **env\_setup.sh**

This script is used to create the links between the GNAT, RFPHY, MSGEN and the PHYs. It takes arguments for `SAT_ID` and `PHYNUM`. This script is called by `sat_start.sh` to complete the internal setup for each satellite container. OpenVswitch and Docker daemon must be up and running in the container for this script to operate. The GNAT containers must have an IP addresses, which is assigned to be `10.8.2.100` for all satellites. Using the PID, symbolic links are created with the namespace so that virtual links can be added. The links between the GNAT and `br0` are set using the Veth links with one

end on the OVS and one end in the namespace. This operation is repeated for MSGEN and RFPHY. The same process is also performed to connect several PHYs to the bridge.

### **env\_clear.sh**

This script removes all the links by removing the Veth links for individual components of the satellite and deleting br0. The script takes PHY\_NUM as input and performs operations for all PHYs. It also stops all nested containers.

### **add\_eth.sh**

This script adds eth1 interfaces and other required interfaces when called by env\_setup.sh. It uses the SatID to generate 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 are provided in order if they are needed by the script.

For ex: **./add\_eth.sh 1 0 gnat ctrl.gnat 192.169.0.1/24**

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

---

## Further usage

---

A custom multi-node system can be created and used for testing.

Using custom code: In the current setup we provide code that is transferred by default by the scripts into the containers. This code could be replaced with customized code created by a user.

Using **sudo docker cp [options] src\_path|- container:dest\_path**, custom code can be transferred for subsequent execution within a container. Figure 36 shows how to transfer a directory with files into an environment for container “sat1” execution. The container must be running to execute 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 36: Copying files to satellite containers

Once transferred the code can be run in normal terminal mode (i.e ./code).

Using a container as a terminal: a terminal can be started for each container 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 37: Starting a bash for a satellite container

This command provides a terminal for a specific container and can be used to run code or create new connections.

Setting a custom IP address: a custom IP address can be set to any interface in the system.

For example, to change the IP address of a GNAT interface in satellite 1, the following command can be used:

**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.

To creating a custom Veth link to a nested container, commands from the scripts directory can be used: For example, set a custom link to phyl1 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

```

```

root@ubuntu:/home/guest/AirForce/scripts$ sudo ip link add end_1 type veth pe
er name end_2
root@ubuntu:/home/guest/AirForce/scripts$ sudo ip link set end_1 netns sat1
root@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"
root@ubuntu:/home/guest/AirForce/scripts$ sudo ifconfig end_2 up

```

Figure 38: Setting a new link interface inside container

This end can be connected to an OVS bridge for ex phybr:

```

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

```

```

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

```

Figure 39: 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 40: 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 OpenvSwitch can be seen as follows:

```

sudo ovs-vsctl show

```

The end\_2 is 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 41: Port added to the external ovs*

There are several other customization possibilities which can be explored using examples from the Docker documents page [9].

---

## *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](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/>

---

# APPENDIX

---

## APPENDIX A

### **sat\_start.sh**

```
#!/bin/bash
# ID of the Satellite
SAT_IDX=$1
# Number of PHY's in the Satellite
PHY_NUM=$2

# The name of the generated nested container
CT_NAME=sat$SAT_IDX

# Check whether the container has been created
if ! sudo docker container ls -a | grep -q -w $CT_NAME; then
    # Create host container in detach mode using the docker-in-docker image provided, see [7]
    sudo docker run -it -d --privileged --name=$CT_NAME jpetazzo/dind
    # Install necessary dependencies
    sudo docker exec $CT_NAME /bin/sh -c "apt-get update"
    # iproute2 for ping, net-tools for ifconfig, manpages for help, iputils, iperf is a testing tool, vim is an
    # editing tool, git is a content management tool.
    sudo docker exec $CT_NAME /bin/sh -c "apt-get -y install net-tools iproute2 build-essential manpages-
    dev iputils-ping iperf vim git apt-utils"
    # Install python, automake for installation of OpenvSwitch, tcpdump for testing of ovs, linux-headers to
    # load correct kernel modules for OpenvSwitch
    sudo docker exec $CT_NAME /bin/sh -c "apt-get -y install python-simplejson python-qt4 python-twisted-
    conch automake autoconf gcc uml-utilities libtool pkg-config libssl-dev tcpdump linux-headers-$(uname
    -r)"
    # Enables reinstallation of linux-image and kernel headers for Linux(OpenvSwitch installation manual)
    # non-interactively
    sudo docker exec $CT_NAME /bin/sh -c "export DEBIAN_FRONTEND=noninteractive && apt-get -y -
    o Dpkg::Options::="--force-confdef" -o Dpkg::Options::="--force-confold" install --reinstall linux-
    image-$(uname -r)"
    # Install openvswitch in the container, pull from git
    sudo docker exec $CT_NAME /bin/sh -c "git clone https://github.com/openvswitch/ovs.git"
    sudo docker exec $CT_NAME /bin/sh -c "cd /ovs; git checkout master; ./boot.sh; ./configure; make;
    make install"
    # Load kernel modules for OpenvSwitch to be installed on
    sudo docker exec $CT_NAME /bin/sh -c "/sbin/modprobe openvswitch"
    sudo docker exec $CT_NAME /bin/sh -c "/sbin/lsmmod | grep openvswitch"

    # Create nested containers, first verify that docker daemon is running
    if ! sudo docker exec $CT_NAME /bin/sh -c "ps -A | grep -q dockerd"; then
```

```

        sudo docker exec $CT_NAME /bin/sh -c "service docker start"
    fi
    # Create 3 containers, corresponding to gnat,msgen,rfphy
    sudo docker exec $CT_NAME /bin/sh -c "docker pull ubuntu"
    sudo docker exec $CT_NAME /bin/sh -c "docker run -it -d --privileged --name=gnat ubuntu /bin/bash"
    sudo docker exec $CT_NAME /bin/sh -c "docker run -it -d --privileged --name=msgen ubuntu /bin/bash"
    sudo docker exec $CT_NAME /bin/sh -c "docker run -it -d --privileged --name=rfphy ubuntu /bin/bash"
    # Create PHY_NUM PHY's, PHY_NUM is taken as input argument
    for i in $(seq 1 $PHY_NUM)
    do
        sudo docker exec $CT_NAME /bin/sh -c "docker run -it -d --privileged --name=phy$i ubuntu
        /bin/bash"
    done

    # Install same dependencies in all nested containers
    sudo docker exec $CT_NAME /bin/sh -c "docker exec gnat /bin/sh -c \"apt-get update && apt-get -y
    install net-tools iproute2 build-essential manpages-dev iputils-ping iperf tcpdump vim python3 apt-
    utils\""
    sudo docker exec $CT_NAME /bin/sh -c "docker exec msgen /bin/sh -c \"apt-get update && apt-get -y
    install net-tools iproute2 build-essential manpages-dev iputils-ping iperf tcpdump vim python3 apt-
    utils\""
    sudo docker exec $CT_NAME /bin/sh -c "docker exec rfphy /bin/sh -c \"apt-get update && apt-get -y
    install net-tools iproute2 build-essential manpages-dev iputils-ping iperf tcpdump vim python3 apt-
    utils\""
    for i in $(seq 1 $PHY_NUM)
    do
        sudo docker exec $CT_NAME /bin/sh -c "docker exec phy$i /bin/sh -c \"apt-get update && apt-
        get -y install net-tools iproute2 build-essential manpages-dev iputils-ping iperf tcpdump vim python3
        apt-utils\""
    done

    # Create a bridge in host container to connect gnat, phy's, msgen and rfphy
    sudo docker exec $CT_NAME /bin/sh -c "export PATH=$PATH:/usr/local/share/openvswitch/scripts;
    ovs-ctl start"
    sudo docker exec $CT_NAME /bin/sh -c "ovs-vsctl add-br br0"

    # Copy the files required to create the internal connections to each the container
    sudo docker cp internal/env_setup.sh $CT_NAME:/home/
    sudo docker cp internal/env_clear.sh $CT_NAME:/home/
    sudo docker cp internal/add_eth.sh $CT_NAME:/home/
    # Copy the C++ project folder to the corresponding nested containers
    sudo docker cp ../GNAT_Docker $CT_NAME:/home/
    sudo docker exec $CT_NAME /bin/sh -c "cd home; docker cp GNAT_Docker gnat:/home/"
    sudo docker exec $CT_NAME /bin/sh -c "cd home; docker cp GNAT_Docker msgen:/home/"
    for i in $(seq 1 $PHY_NUM)
    do
        sudo docker exec $CT_NAME /bin/sh -c "cd home; docker cp GNAT_Docker phy$i:/home/"
    done

    # Build C code in nested containers
    sudo docker exec $CT_NAME /bin/sh -c "docker exec gnat /bin/sh -c \"cd home/GNAT_Docker/GNAT;
    gcc -w -o GNAT GNAT.c -lpthread -lm\""

```

```

sudo docker exec $CT_NAME /bin/sh -c "docker exec msgen /bin/sh -c \"cd home/GNAT_Docker/PHY-
MGEN; gcc -w -o PHY-MGEN MessageGenerator.c -lpthread -lm\""
for i in $(seq 1 $PHY_NUM)
do
    sudo docker exec $CT_NAME /bin/sh -c "docker exec phy$i /bin/sh -c \"cd
    home/GNAT_Docker/PHY-Direct; gcc -w -o PHY-Direct DirectConnect.c -lpthread -lm\""
done

# Set container internal environment, i.e connecting all the nested containers with br0
sudo docker exec $CT_NAME /bin/sh -c "cd home; ./env_setup.sh $SAT_IDX $PHY_NUM"

# Check whether the container has been started already
elif ! sudo docker container ls | grep -q $CT_NAME; then
    # Start host container
    sudo docker container start $CT_NAME &
    process_id=$!
    wait $!
    echo "process_id=$!"

    # Check whether docker daemon is running
    if ! sudo docker exec $CT_NAME /bin/sh -c "ps -A | grep dockerd"; then
        sudo docker exec $CT_NAME /bin/sh -c "service docker start"
    fi

    # Set container internal environment
    sudo docker exec $CT_NAME /bin/sh -c "cd home; ./env_setup.sh $SAT_IDX $PHY_NUM"

else
    # Check whether docker daemon is running
    if ! sudo docker exec $CT_NAME /bin/sh -c "ps -A | grep -q dockerd"; then
        sudo docker exec $CT_NAME /bin/sh -c "service docker start"
    fi
fi

sat_stop.sh
#!/bin/bash
#Id of the Satellite
SAT_IDX=$1
# Number of PHY's
PHY_NUM=$2

# The name of the generated nested container
CT_NAME=sat$SAT_IDX

# Clear the internal setup, delete links and bridge ports
sudo docker exec $CT_NAME /bin/sh -c "cd home; ./env_clear.sh $PHY_NUM"
sudo docker container stop $CT_NAME

```

## **env\_setup.sh**

```

#!/bin/bash

```

```

# Prompt user to enter Number of Satellites, Number of PHY's, and number of cores to be used to setup the
simulator
while getopts ":s:p:j:" opt; do
    case $opt in
        s) SAT_NUM="$OPTARG"
        ;;
        p) PHY_NUM="$OPTARG"
        ;;
        j) JOB_NUM="$OPTARG"
        ;;
        \?) echo "Invalid option -$OPTARG" >&2
        ;;
    esac
done
#
if ! ps -A | grep -q -w ovs-vswitchd; then
    echo "Start the openvswitch!"
    sudo /usr/local/share/openvswitch/scripts/ovs-ctl start
fi

echo "Creating a network with $SAT_NUM nodes!"

# Using GNU Parallel to setup satellites parallelly, using multiple cores, to mitigate time taken for each satellite
to setup
sudo apt-get -qq -y install parallel
if [ -z ${JOB_NUM+x} ]; then
    seq 1 $SAT_NUM | parallel --no-notice ./sat_start.sh {} $PHY_NUM
    for i in $(seq 1 $SAT_NUM)
    do
        ./sat_start.sh $i $PHY_NUM
    done
else
    seq 1 $SAT_NUM | parallel --no-notice -j $JOB_NUM ./sat_start.sh {} $PHY_NUM
fi

echo "Setup the full connection for master controller with every gnat!"
# Setup the connection between mastercontroller and the gnats through an OpenvSwitch bridge
./ovs_net_setup.sh $SAT_NUM $PHY_NUM

```

## env\_clear.sh

```

#!/bin/bash
# Clear the above setup
while getopts ":s:p:j:" opt; do
    case $opt in
        s) SAT_NUM="$OPTARG"
        ;;
        p) PHY_NUM="$OPTARG"
        ;;
        j) JOB_NUM="$OPTARG"
        ;;
        \?) echo "Invalid option -$OPTARG" >&2
    esac
done

```

```

;;
esac
done

echo "Turn off all nodes! Clear the network setting!"

if [ -z ${JOB_NUM+x} ]; then
    #seq 1 $SAT_NUM | parallel --no-notice ./sat_stop.sh {} $PHY_NUM
    for i in $(seq 1 $SAT_NUM)
    do
        ./sat_stop.sh $i $PHY_NUM
    done
else
    #repeatedly call sat_stop.sh to clear the internal setup of each satellite
    seq 1 $SAT_NUM | parallel --no-notice -j $JOB_NUM ./sat_stop.sh {} $PHY_NUM
fi

./ovs_net_clear.sh $SAT_NUM $PHY_NUM

```

## **ovs\_net\_setup.sh**

```

#!/bin/bash
#No. of Satellites as input argument
SAT_NUM=$1
#No. of PHY's in each satellite as input argument
PHY_NUM=$2

# Create a bridge for connecting the master controller with all GNATs
sudo ovs-vsctl add-br ctrlbr
echo "Create the ctrlbr bridge ..."

# Create network namespace for isolation of each container
sudo mkdir -p /var/run/netns
for i in $(seq 1 $SAT_NUM)
do
    pid=$(sudo docker inspect --format '{{ .State.Pid }}' "sat$i")
    name=$(sudo docker inspect --format '{{ .Name }}' "sat$i")
    sudo ln -sf /proc/$pid/ns/net /var/run/netns/$name #Create a symbolic link between Pid and Name of
    namespace
done

# Create virtual link for connecting ovs and nested containers
for i in $(seq 1 $SAT_NUM)
do
    # Create connection between ovs and gnat
    #Add a veth link with ctrl.gnat port inside gnat and sat$i.gnat on the ovs bridge
    sudo ip link add ctrl.gnat type veth peer name sat$i.gnat
    #Set the ctrl.gnat link inside the satellite
    sudo ip link set ctrl.gnat netns sat$i
    sudo ip link set sat$i.gnat nomaster
    #Set the link inside gnat of sat1 with the IP address
    sudo docker exec sat$i /bin/sh -c "cd home; ./add_eth.sh $i 0 gnat ctrl.gnat 192.169.0.$i/24"
    #Add the port on the openvswitch
    sudo ovs-vsctl add-port ctrlbr sat$i.gnat

```



```

        sudo ifconfig sat$i.gnat up
done

# Turn up the ctrlbr
sudo ifconfig ctrlbr up
sudo ifconfig ctrlbr 192.169.0.254 netmask 255.255.255.0

echo "Connection for ctrlbr has been set!"

# ----- cutting line -----

# Create a bridge for connecting all PHYs
sudo ovs-vsctl add-br phybr
echo "Create the phybr bridge ..."

# Create virtual link for connecting ovs and nested containers(PHY's)
for i in $(seq 1 $SAT_NUM)
do
    for j in $(seq 1 $PHY_NUM)
    do
        # Create connection between ovs and phys with veth links
        #Links inside the PHY's is the eth1 interface, on the bridge is sati.phy$j
        sudo ip link add eth1 type veth peer name sat$i.phy$j
        sudo ip link set eth1 netns sat$i
        sudo ip link set sat$i.phy$j nomaster
        #All eth1 have the same IP address
        sudo docker exec sat$i /bin/sh -c "cd home; ./add_eth.sh $i $j phy$j eth1 192.168.2.1/24"
        sudo ovs-vsctl add-port phybr sat$i.phy$j
        sudo ifconfig sat$i.phy$j up
    done
done

# Turn up the phybr
sudo ifconfig phybr up
sudo ifconfig phybr 192.168.2.254 netmask 255.255.255.0

# Delete the default rule in phybr
sudo ovs-ofctl del-flows phybr

echo "Connection for phybr has been set!"

```

## **ovs\_net\_clear.sh**

```

#!/bin/bash
SAT_NUM=$1
PHY_NUM=$2

# Turn down virtual eth interface, delete the ctrlbr and phybr bridges
sudo ifconfig ctrlbr down
for i in $(seq 1 $SAT_NUM)
do
    sudo ovs-vsctl del-port ctrlbr sat$i.gnat
done
sudo ovs-vsctl del-br ctrlbr

```

```
echo "Delete the ctrlbr bridge"
```

```
sudo ifconfig phybr down
for i in $(seq 1 $SAT_NUM)
do
    for j in $(seq 1 $PHY_NUM)
    do
        sudo ovs-vsctl del-port phybr sat$i.phy$j
    done
done
sudo ovs-vsctl del-br phybr
```

```
echo "Delete the phybr bridge!"
```

## Internal Scripts

### add\_eth.sh

```
#!/bin/bash
#Take input arguments as satelliteID, ContainerID & name, interface name and IP address
SAT_IDX=$1
CT_IDX=$2
CT_NAME=$3
ETH_INF=$4
IP_ADDR=$5

# Calculate the most and least bytes of MAC address for eth1 interface
MAC_LSB_ADDR=$( printf '%x\n' $((256 - $SAT_IDX)) )
MAC_SLB_ADDR=$( printf '%x\n' $((256 - $CT_IDX)) )

#Create symbolic link for the satellite
pid=$(sudo docker inspect --format '{{ .State.Pid }}' "$CT_NAME")
name=$(sudo docker inspect --format '{{ .Name }}' "$CT_NAME")
ln -sf /proc/$pid/ns/net /var/run/netns/$name
ip link set $ETH_INF netns $CT_NAME
ip netns exec $CT_NAME ip link set $ETH_INF up
ip netns exec $CT_NAME ip addr add $IP_ADDR dev $ETH_INF

# Set MAC address for eth1 interface that was calculated previously
if [ "$ETH_INF" = "eth1" ]; then
    ip netns exec $CT_NAME ifconfig eth1 hw ether 00:00:00:00:$MAC_SLB_ADDR:$MAC_LSB_ADDR
fi
```

### env\_setup.sh

```
#!/bin/bash
SAT_IDX=$1
PHY_NUM=$2

# Calculate the least byte of MAC address for gnat
MAC_LSB_ADDR=$( printf '%x\n' $SAT_IDX )

# Start ovs bridge
```

```

export PATH=$PATH:/usr/local/share/openvswitch/scripts
ovs-ctl start
ifconfig br0 up

# Check whether docker daemon started
while ! ps -A | grep -q dockerd ; do
    service docker start
    sleep 1
done

# Start gnat container, get the process ID, set ipaddr variable
docker container start gnat &
process_id=$!
wait $!
ipaddr_gnat="192.168.0.2/24"

# Check whether docker daemon started, if not restart
while ! ps -A | grep -q dockerd ; do
    service docker start &
    process_id=$!
    wait $!
    docker container stop gnat
    docker container start gnat &
    process_id=$!
    wait $!
done

# Get gnat pid, create a symbolic link for it
pid=$(sudo docker inspect --format '{{ .State.Pid }}' "gnat")
name=$(sudo docker inspect --format '{{ .Name }}' "gnat")
mkdir -p /var/run/netns
ln -sf /proc/$pid/ns/net /var/run/netns/$name

# Set link between br0 and gnat
#veth ends- eth_bg for bridge gnat connection, eth2 for gnat interface
ip link add eth_bg type veth peer name eth2
#Set the eth2 in gnat namespace
ip link set eth2 netns gnat
ip link set eth_bg nomaster
#Set eth2 up in gnat nested container, add a virtual interface(macvlan) on eth2 'gnat'
ip netns exec gnat ifconfig eth2 up
ip netns exec gnat ip link add gnat link eth2 type macvlan mode bridge
ip netns exec gnat ifconfig gnat up
#Set MAC address for gnat macvlan interface
ip netns exec gnat ifconfig gnat hw ether 00:00:00:00:00:$MAC_LSB_ADDR
#Set IP address as set above
ip netns exec gnat ip addr add $ipaddr_gnat dev gnat
#Add other end of veth link to br0 bridge
ovs-vsctl add-port br0 eth_bg
ifconfig eth_bg up

# Start msgen container, set ipaddr variables for the mgen interfaces
docker container start msgen &

```

```

process_id=$!
wait $!
ipaddr_mgenc="192.168.0.50/24"
ipaddr_mgend="192.168.0.51/24"

# Check whether docker daemon started
while ! ps -A | grep -q dockerd ; do
    service docker start &
    process_id=$!
    wait $!
    docker container stop msgen
    docker container start msgen &
    process_id=$!
    wait $!
done

# Get msgen pid, create a symbolic link for it
pid=$(sudo docker inspect --format '{{ .State.Pid }}' "msgen")
name=$(sudo docker inspect --format '{{ .Name }}' "msgen")
mkdir -p /var/run/netns
ln -sf /proc/$pid/ns/net /var/run/netns/$name

# Set link between br0 and msgen
#Veth ends- eth_bm for bridge msgen connection, eth2 for msgen interface
ip link add eth_bm type veth peer name eth2
#Set eth2 in msgen namespace
ip link set eth2 netns msgen
ip link set eth_bm nomaster
ip netns exec msgen ifconfig eth2 up
# Create 2 macvlan virtual interfaces over eth2, mgen.d and mgen.c
ip netns exec msgen ip link add mgen.c link eth2 type macvlan mode bridge
ip netns exec msgen ip link add mgen.d link eth2 type macvlan mode bridge
ip netns exec msgen ifconfig mgen.c up
ip netns exec msgen ip addr add $ipaddr_mgenc dev mgen.c
ip netns exec msgen ifconfig mgen.d up
ip netns exec msgen ip addr add $ipaddr_mgend dev mgen.d
#Add other end of veth link on br0 bridge
ovs-vsctl add-port br0 eth_bm
ifconfig eth_bm up

# Start rfphy container, set ipaddr
docker container start rfphy &
process_id=$!
wait $!
ipaddr_rpc="192.168.0.200/24"
ipaddr_rpd="192.168.0.201/24"

# Check whether docker daemon started
while ! ps -A | grep -q dockerd ; do
    service docker start &
    process_id=$!
    wait $!
    docker container stop rfphy

```

```

    docker container start rfphy &
    process_id=$!
    wait $!
done

# Get rfphy pid, create a symbolic link for it
pid=$(sudo docker inspect --format '{{ .State.Pid }}' "rfphy")
name=$(sudo docker inspect --format '{{ .Name }}' "rfphy")
mkdir -p /var/run/netns
ln -sf /proc/$pid/ns/net /var/run/netns/$name

# Set link between br0 and rfphy
#veth ends- eth_brp for bridge rfphy connection, eth2 for rfphy interface
ip link add eth_brp type veth peer name eth2
#Set eth2 in rfphy namespace
ip link set eth2 netns rfphy
ip link set eth_brp nomaster
ip netns exec rfphy ifconfig eth2 up
#Set rfphy.c and rfphy.d, 2 virtual macvlan interfaces over hard eth2 interface
ip netns exec rfphy ip link add rfphy.c link eth2 type macvlan mode bridge
ip netns exec rfphy ip link add rfphy.d link eth2 type macvlan mode bridge
ip netns exec rfphy ifconfig rfphy.c up
ip netns exec rfphy ip addr add $ipaddr_rpc dev rfphy.c
ip netns exec rfphy ifconfig rfphy.d up
ip netns exec rfphy ip addr add $ipaddr_rpd dev rfphy.d
#Set the bridge port with other end of veth
ovs-vsctl add-port br0 eth_brp
ifconfig eth_brp up

# Loop for PHY configuration
for i in $(seq 1 $PHY_NUM)
do
    # Start phy container, set ipaddr
    docker container start phy$i &
    process_id=$!
    wait $!
    #Set IP address such that last byte is set using the phy number.
    ipaddr_phy_1c="192.168.0.$(( 100 + ($i - 1)*2 ))/24"
    ipaddr_phy_1d="192.168.0.$(( 100 + ($i - 1)*2 + 1 ))/24"

    # Check whether docker daemon started
    while ! ps -A | grep -q dockerd ; do
        service docker start
        process_id=$!
        wait $!
        docker container stop phy$i
        docker container start phy$i &
        process_id=$!
        wait $!
    done

    # Get phy_i pid, create a symbolic link for it
    pid=$(sudo docker inspect --format '{{ .State.Pid }}' "phy$i")

```

```

name=$(sudo docker inspect --format '{{.Name}}' "phy$i")
mkdir -p /var/run/netns
ln -sf /proc/$pid/ns/net /var/run/netns/$name

# Set link between br0 and phy_i
#veth ends- eth_bp(PHY no.) for bridge PHY connection, eth2 for PHY interface
ip link add eth_bp$i type veth peer name eth2
ip link set eth2 netns phy$i
ip link set eth_bp$i nomaster
ip netns exec phy$i ifconfig eth2 up
#Add virtual macvlan interface direct.1c and direct.1d inside the phy over eth2
ip netns exec phy$i ip link add direct.1c link eth2 type macvlan mode bridge
ip netns exec phy$i ip link add direct.1d link eth2 type macvlan mode bridge
ip netns exec phy$i ifconfig direct.1c up
ip netns exec phy$i ip addr add $ipaddr_phy_1c dev direct.1c
ip netns exec phy$i ifconfig direct.1d up
ip netns exec phy$i ip addr add $ipaddr_phy_1d dev direct.1d
#Add other veth end on the satellite bridge br0
ovs-vsctl add-port br0 eth_bp$i
ifconfig eth_bp$i up

```

done

## env\_clear.sh

```

#!/bin/bash
PHY_NUM=$1

# Clear setting for gnat
#Down the links
ip netns exec gnat ifconfig gnat down
ip netns exec gnat ifconfig eth2 down
ifconfig eth_bg down
#Delete the port and veth link
ovs-vsctl del-port br0 eth_bg
ip link del eth_bg type veth peer name eth2

# Clear setting for msgen
ip netns exec msgen ifconfig mgen.c down
ip netns exec msgen ifconfig mgen.d down
ip netns exec msgen ifconfig eth2 down
ifconfig eth_bm down
#Delete the port and veth link
ovs-vsctl del-port br0 eth_bm
ip link del eth_bm type veth peer name eth2

# Clear setting for rfphy
ip netns exec rfphy ifconfig rfphy.c down
ip netns exec rfphy ifconfig rfphy.d down
ip netns exec rfphy ifconfig eth2 down
ifconfig eth_brp down
#Delete the port and veth link

```

```

ovs-vsctl del-port br0 eth_brp
ip link del eth_brp type veth peer name eth2

# Clear setting for phys
for i in $(seq 1 $PHY_NUM)
do
    ip netns exec phy$i ifconfig direct.1c down
    ip netns exec phy$i ifconfig direct.1d down
    ip netns exec phy$i ifconfig eth2 down
    ifconfig eth_bp$i down
    #Delete the port and veth link
    ovs-vsctl del-port br0 eth_bp$i
    ip link del eth_bp$i type veth peer name eth2
done

# Turn down the bridge
ifconfig br0 down

# Turn off the nested containers
docker container stop gnat
docker container stop msgen
docker container stop rfphy
for i in $(seq 1 $PHY_NUM)
do
    docker container stop phy$i
done

```