H3ABioNet
Pan African Bioinformatics Network for H3Africa

# HOWTO: ADVANCED GUIDE FOR DOCKER

A hands-on step-by-step advanced guide to Docker essentials.

**Background**

This guide is produced by the Computing & Infrastructure working group under the Pipelines and Computing work package. It is intended to serve as an advanced supplementary guide to the basic guide for seasoned system admins across the H3ABioNet and its collaborating partners. The guide has been designed to complement the basic guide for experienced sys admins effective as an almost self-sufficient walkthrough of major aspects; networking, storage, jupyter notebooks and security considerations.

| **Version Amendment History** | | |
|---|---|---|
| Version | Date | Reason for change/Remarks |
| 1.0 | April 2021 | Creation |
| | | |

| **Document Control Box** | |
|---|---|
| Procedure title: | Docker – HowTo Advanced Guide |
| Date approved: | May 2021 ?? |
| Approving body: | PC Work Package & Management Committee |
| Version: | 1.0 |
| Previous review dates: | |
| Next review date: | |
| Related information: | |
| Procedure owner: | Computing Infrastructure Project Team |

| **Project Members** | | | |
|---|---|---|---|
| Last Name | First Name | Institution | Country |
| Ghanmi | Nidhal | | Tunisia |
| Lukyamuzi | Edward | Uganda Virus Research Institute | Uganda |
| Maslamoney | Suresh | Computational Biology Division, University of Cape Town | South Africa |
| Kimbowa | Timothy | Uganda Virus Research Institute | Uganda |

## Table of contents

# 1.0 Docker Networking

## 1.1 Using OVS bridge for Docker networking

OVS bridges or Open vSwitch bridges are used as an alternative to the native bridges in linux. It supports most features which are in a physical switch while also supporting multiple vLANs on a single bridge. It is widely used in Docker networking because it proves to be useful for multiple host networking and provides more secure communication compared to native bridges. Let us now create, add and configure a new OVS bridge to get docker containers on different networks to connect to each other

**Install OVS**

```
$ sudo apt-get install openvswitch-switch
```

**Install ovs-docker utility**

```
$ cd /usr/bin
$ wget
https://raw.githubusercontent.com/openvswitch/ovs/master/utilities/ovs-docker
$ chmod a+rwx ovs-docker
```

**Create an OVS bridge**

Here we will be adding a new OVS bridge and configuring it, so that we can get the containers connected on the different network.

```
$ ovs-vsctl add-br ovs-br1
$ ifconfig ovs-br1 10.0.0.1 netmask 255.0.0.0 up
```

**Add a port from OVS bridge to the Docker Container**

1. Create two ubuntu Docker Containers

```
$ docker run -i -t --name container1 ubuntu /bin/bash
$ docker run -i -t --name container2 ubuntu /bin/bash
```

2. Connect the container to OVS bridge

```
$ ovs-docker add-port ovs-br1 eth1 container1 --ipaddress=10.0.0.2/8
$ ovs-docker add-port ovs-br1 eth1 container2 --ipaddress=10.0.0.3/8
```

3. Test the connection between two containers connected via OVS bridge using Ping command

**Extra configuration**

If the containers are required to be connected to internet then a port is required to be added to the ethernet bridge of host which can be configured as follows. Please add an extra bridge eth1 so that we don't affect the present state of the host.

```
$ ovs-vsctl add-port ovs-br1 eth1
```

## 1.2 Weave Networking for Docker

Weave creates a virtual network that enables users to connect docker containers on different host and enable their auto-discovery

**Install Weave**

```
$ sudo wget -O /usr/local/bin/weave \
https://github.com/weaveworks/weave/releases/download/latest_release/weave
$ sudo chmod a+x /usr/local/bin/weave
```

**Launch weave containers:** internally pull weave router container and run it

```
$ weave launch
```

**Start two application containers on weave network**

```
$ C=$(weave run 10.10.1.1/24 -i -t ubuntu)
$ C12=$(weave run 10.10.1.2/24 -i -t ubuntu)
```

*C* and *C12* hold the containerId of the containers created

*weave run* command will internally run *docker run -d* command in order to set the ip address of weave network and start the ubuntu containers. Test the connection between two containers connected via weave network by using the *ping* command

```
$ docker attach $C
$ ping 10.10.1.2 -c 4
```

# 2.0 Docker Volumes

## 2.1 Volumes from Docker Image

**Docker Image with Volume specified in Dockerfile**

Let's look at: How to add a file as a volume using Dockerfile format, Create an Image from the Dockerfile and Use the Image to create a container and check if the file exists.

Create a Dockerfile in a directory. Make sure *log1* exists in the same directory. We are going to mount *log1* as */h3abionet1/log* and mount */h3abionet1* as a volume.

```
FROM ubuntu:20.04
ADD log1 /h3abionet1/log
VOLUME /h3abionet1
CMD /bin/sh
```

Build the image test/volume-by-dockerfile

```
docker build -t test/volume-by-dockerfile .
```

Create a container from the image *test/volume-by-dockerfile*

```
docker run -it test/volume-by-dockerfile
```

Check if the volume */h3abionet1* is mounted using ls command

```
# ls
bin  boot  dev    etc  home  lib  .. sbin  srv  sys  tmp  h3abionet1  usr  var
# ls /h3abionet1
log
```

**Docker Container with volume from Command Line and Image**

Let's use command line to mount a host directory into a container created from an image

Create a local directory *h3abionet2* and two files *log1* and *log2* in that directory.

```
$ sudo mkdir -p /h3abionet2
[sudo] password for ubuntu:
$ sudo touch /h3abionet2/log1
$ sudo touch /h3abionet2/log2
```

Create a container with a volume *h3abionet2* from the image: *test/volume-by-dockerfile* by specifying the directory to be mounted on the command line with a flag *-v*.

```
$ docker run -it -v /h3abionet2:/h3abionet2 test/volume-by-dockerfile
```

Check that the directory *h3abionet2* got mounted in the docker container. Run *ls* in the container shell.

```
# ls
bin  boot  dev    etc  home  lib sys  tmp  h3abionet1  h3abionet2     usr  var
# ls h3abionet2
log1  log2
```

As you can see above both *h3abionet1* and *h3abionet2* got mounted as volumes.

**Container with ReadOnly Volume**

Specify *the :ro* as shown below to make the volume readonly.

```
$ docker run -it -v /h3abionet2:/h3abionet2:ro test/volume-by-dockerfile
```

Try creating a new file in that volume from the bash shell of the container.

```
# touch /h3abionet2/log3
touch: cannot touch '/h3abionet2/log3': Read-only file system
```

## 2.2 Volumes from Another Container

Volumes from a container can be bound to another container *using --volumes-from <container-name>* flag. Make sure there is host directory with contents */h3abionet1/log*

```
$ ls /h3abionet1
log
```

**Create a Container with a Volume:** Create a container with name *h3abionet01* from image *ubuntu*

```
        $ docker run -it --name h3abionet01 -v /h3abionet1:/h3abionet1 ubuntu
        root@#############:/# ls h3abionet1
log
```

**Create Second Container with shared volumes**: Create a second container *h3abionet02* with volumes from *h3abionet01*

```
        $ docker run -it --name h3abionet02 --volumes-from h3abionet01 ubuntu
```

Check that the *h3abionet1* volume is bound as expected

```
root@b28ca7033e9d:/# ls
bin  boot  dev  etc  home  ..  h3abionet1  usr  var
root@b28ca7033e9d:/# ls h3abionet1
log
```

# 3.0 Docker security

**Docker Security CIS Benchmark**

The section is based on the Centre for Internet Security (CIS) benchmark, CIS DOCKER BENCHMARK V1.2.0. Here, we will be covering all the important guidelines to run docker containers in secured environment.

## 3.1 Host Configuration

This section covers security recommendations that you should follow to prepare the host machine that you plan to use for executing containerized workloads. Securing the Docker host and following your infrastructure security best practices would build a solid and secure foundation for executing containerized workloads.

### 3.1.1 General Configuration

This section contains general host recommendations for systems running Docker

- Harden container host
- Keep docker version up to date

By staying up to date on Docker updates, vulnerabilities in the Docker software can be mitigated. An attacker may exploit known vulnerabilities when attempting to attain access or elevate privileges. Not installing regular Docker updates may leave you with running vulnerable Docker software. It might lead to elevation privileges, unauthorized access or other security breaches.

```
        $ docker version
```

### 3.1.2 Linux Hosts Specific Configuration

This section contains recommendations that securing Linux Hosts running Docker Containers

- Create separate partition for containers
- Only allow trusted users to control Docker daemon

The Docker daemon currently requires ‘root’ privileges. A user added to the ‘docker’ group gives him full ‘root’ access rights. Hence, only verified users should be added to docker group.

```
$ useradd test

$ usermod -G docker test

$ docker ps
```

- Audit Docker daemon

Apart from auditing your regular Linux file system and system calls, audit Docker daemon as well. Docker daemon runs with ‘root’ privileges. It is thus necessary to audit its activities and usage

```
$ apt-get install auditd
```

Add the audit rules for docker service and audit the docker service

```
$ nano /etc/audit/audit.rules
-w /usr/bin/docker -k docker

$ service auditd restart

$ ausearch -k docker
```

- Audit Docker files and directories

## 3.2 Docker Daemon Configuration

This section lists the recommendations that alter and secure the behaviour of the Docker daemon. The settings that are under this section affect ALL container instances. Note: Docker daemon options can also be controlled using files such as */etc/sysconfig/docker*, */etc/default/docker*, the *systemd* unit file or */etc/docker/daemon.json.* Also, note that Docker in daemon mode can be identified as */usr/bin/dockerd*, or having *-d* or daemon as the argument to docker service.

- Restricted network traffic between containers on the default bridge

By default, unrestricted network traffic is enabled between all containers on the same host. Thus, each container has the potential of reading all packets across the container network on the same host. This might lead to unintended and unwanted disclosure of information to other containers. Hence, restrict the inter container communication by setting the *icc* flag to false

```
$ service docker stop

$ docker -d --icc=false &
```

- Set logging level to 'info' .
- Allow Docker to make changes to iptables
- Do not use insecure registries
- Do not use aufs storage driver
- Configure TLS authentication for Docker daemon
- Appropriately configure e the default ulimit
- Enable user namespace support

- Confirm default cgroup usage
- Do not change base device size until needed
- Enable authorization for Docker client commands
- Configure centralized and remote logging
- Enable live restore
- Disable Userland Proxy
- Appropriately apply daemon-wide custom seccomp profile
- Do not implement experimental features in production
- Restrict containers from acquiring new privileges

## 3.3 Docker Daemon Configuration Files

This section covers Docker related files and directory permissions and ownership. Keeping the files and directories, that may contain sensitive parameters, secure is important for correct and secure functioning of Docker daemon.

- Verify that the docker.service file ownership is set to root:root
- Verify that docker.service file permissions are appropriately set
- Verify that docker.socket file ownership is set to root:root

If you are using Docker on a machine that uses systemd to manage services, then verify that the 'docker.service' file ownership and group-ownership is correctly set to 'root'. So that when the account is switched to another user (say, test) he is not able to access the docker daemon as he is not authorized to do so by root account.

```
$ stat -c %U:%G /usr/lib/docker | grep -v root:root
$ stat -c %U:%G /usr/lib/docker | grep root:root
```

If the permission is not set to root:root then it can be changed by using the following command

```
$ chown root:root /usr/lib/systemd/system/docker.service
$ su test
$ docker ps
Cannot connect to the Docker daemon. Is the docker daemon running on this host?
test@ubuntu:/etc/init.d$
```

- Verify that docker.socket file permissions are set to 644 or more restrictive 660

If you are using Docker on a machine that uses systemd to manage services, then verify that the 'docker.service' file permissions are correctly set to '644' or more restrictive.

As it can be seen below if we allocate 666 as the permission then the "test" user will be also be available to access the Docker daemon

```
$ ls -l /var/run/docker.sock

$ chmod 666 /var/run/docker.sock

$ su test
test@ubuntu:/etc/init.d$ docker ps
```

As soon as we change the permission to 660 we will be able to see that the "test" user is not able to access the docker daemon

```
$ chmod 660 /var/run/docker.sock
```

```
$ su test
test@ubuntu:/etc/init.d$ docker ps
Cannot connect to the Docker daemon. Is the docker daemon running on this host?
```

- Set */etc/docker* directory ownership to root:root
- Set */etc/docker* directory permissions to 755 or more restrictively
- Verify that registry certificate file ownership is set to root:root
- Ensure that registry certificate file permissions are set to 444 or more restrictively
- Ensure that TLS CA certificate file ownership is set to root:root
- Ensure that TLS CA certificate file permissions are set to 444 or more restrictively
- Verify that Docker server certificate file ownership is set to root:root
- Ensure that the Docker server certificate file permissions are set to 444 or more restrictively
- Ensure that the Docker server certificate key file ownership is set to root:root
- Ensure that the Docker server certificate key file permissions are set to 400
- Verify that the Docker socket file ownership is set to root:docker (
- Ensure that the Docker socket file permissions are set to 660 or more restrictively
- Ensure that the *daemon.json* file ownership is set to root:root
- Ensure that *daemon.json* file permissions are set to 644 or more restrictive
- Ensure that the */etc/default/docker* file ownership is set to root:root
- Ensure that the */etc/sysconfig/docker* file ownership is set to root:root
- Ensure that the */etc/sysconfig/docker* file permissions are set to 644 or more restrictively
- Ensure that the */etc/default/docker* file permissions are set to 644 or more restrictively

## 3.4 Container Runtime

There are many security implications associated with the ways that containers are started. Some runtime parameters can be supplied that have security consequences that could compromise the host and the containers running on it. It is therefore very important to verify the way in which containers are started, and which parameters are associated with them. Container runtime configuration should be reviewed in line with organizational security policy. Various recommendations to assess the container runtime are as below:

- Ensure that, if applicable, an AppArmor Profile is enabled
- Ensure that, if applicable, SELinux security options are set
- Ensure that Linux kernel capabilities are restricted within containers
- Ensure that privileged containers are not used

Docker supports the addition and removal of capabilities, allowing use of a non-default profile. This may make Docker more secure through capability removal, or less secure through the addition of capabilities. It is thus recommended to remove all capabilities except those explicitly required for your container process.

As seen below when we run the container without the privileged mode, we are unable to change the Kernel parameters but when we run the container in privileged mode using the -privileged flag it is able to change the Kernel Parameters easily, which can cause security vulnerability.

```
$ docker run -it centos /bin/bash
[root@########/]# sysctl -w net.ipv4.ip_forward=0
sysctl: setting key "net.ipv4.ip_forward": Read-only file system
```

```
$ docker run --privileged -it centos /bin/bash
[root@######## /]#  sysctl -a | wc -l

[root@######## /]# sysctl -w net.ipv4.ip_forward=0
net.ipv4.ip_forward = 0
```

So, while auditing it should be made sure that all the containers should not have the privileged mode set to true.

```
$ docker ps -q | xargs docker inspect --format '{{ .Id }}: Privileged={{ .HostConfig.Privileged }}'
```

- Ensure sensitive host system directories are not mounted on containers

If sensitive directories are mounted in read-write mode, it would be possible to make changes to files within those sensitive directories. The changes might bring down security implications or unwarranted changes that could put the Docker host in compromised state.

If /run/systemd, sensitive directory is mounted in the container then we can shut down the host from the container itself.

```
$ docker run -ti -v /run/systemd:/run/systemd centos /bin/bash
[root@1aca7fe47882 /]# systemctl status docker

[root@1aca7fe47882 /]# shutdown
```

It can be audited by using the command below which returns the list of current mapped directories and whether they are mounted in read-write mode for each container instance;

```
$ docker ps -q | xargs docker inspect --format '{{ .Id }}: Volumes={{ .Volumes }} VolumesRW={{ .VolumesRW }}
```

- Ensure sshd is not run within containers
- Ensure privileged ports are not mapped within containers
- Ensure that only needed ports are open on the container
- Ensure that the host's network namespace is not shared
- Ensure that the memory usage for containers is limited
- Ensure that CPU priority is set appropriately on containers
- Ensure that the container's root filesystem is mounted as read only
- Ensure that incoming container traffic is bound to a specific host interface

If you have multiple network interfaces on your host machine, the container can accept connections on the exposed ports on any network interface. This might not be desired and may not be secured. Many a times a particular interface is exposed externally and services such as intrusion detection, intrusion prevention, firewall, load balancing, etc. are run on those interfaces to screen incoming public traffic. Hence, you should not accept incoming connections on any interface. You should only allow incoming connections from a particular external interface.

As shown below the machine has two network interfaces and by default if we run a nginx container it will get binded to localhost (0.0.0.0) that means this container will be accessible from both the IP address which can result in intrusion attack if any of them are not monitored.

```
$ ifconfig

$ docker run -d -p 4915:80 nginx

$ docker port <CONTAINER ID>
```

```
80/tcp -> 0.0.0.0:4915
```

In order to restrict this we should bind container to one of the host interface IP address using the "-p" flag

```
$ docker run -d -p <Host IP>:4915:80 nginx

$ docker port <CONTAINER ID>
80/tcp -> <Host IP>:4915
```

- Ensure that the 'on-failure' container restart policy is set to '5'
- Ensure that the host's process namespace is not shared

PID namespace provides separation of processes. The PID Namespace removes the view of the system processes, and allows process ids to be reused including PID 1. If the host's PID namespace is shared with the container, it would basically allow processes within the container to see all the processes on the host system. This breaks the benefit of process level isolation between the host and the containers. Someone having access to the container can eventually know all the processes running on the host system and can even kill the host system processes from within the container. This can be catastrophic. Hence, do not share the host's process namespace with the containers.

In this section we can see that if the container gets the pid of the host then it can access all the system level process of the host and can kill them as well causing potential threat. So, thus while auditing it should be checked that PID Mode should not be set to host for all the containers.

```
$ docker run -it --pid=host ubuntu /bin/bash
$ ps -ef

$ docker ps -q | xargs docker inspect --format '{{ .Id }}: PidMode={{ .HostConfig.PidMode }}'
<CONTAINER ID>: PidMode=host
```

- Ensure that the host's IPC namespace is not shared
- Ensure that host devices are not directly exposed to containers
- Ensure that the default ulimit is overwritten at runtime if needed
- Ensure mount propagation mode is not set to shared
- Ensure that the host's UTS namespace is not shared
- Ensure the default seccomp profile is not Disabled
- Ensure that docker exec commands are not used with the privileged option
- Ensure that docker exec commands are not used with the user=root option
- Ensure that cgroup usage is confirmed
- Ensure that the container is restricted from acquiring additional privileges
- Ensure that container health is checked at runtime
- Ensure that Docker commands always make use of the latest version of their image
- Ensure that the PIDs cgroup limit is used
- Ensure that Docker's default bridge "docker0" is not used
- Ensure that the host's user namespaces are not shared
- Ensure that the Docker socket is not mounted inside any containers

## 3.5 Container Images and Build file

Container base images and build files govern the fundamentals of how a container instance from a particular image would behave. Ensuring that you are using proper base images and appropriate build files can be very important for building your containerized infrastructure. Below are some of the

recommendations that you should follow for container base images and build files to ensure that your containerized infrastructure is secure.

- Ensure that a user for the container has been created

It is thus highly recommended to ensure that there is a non-root user created for the container and the container is run using that user.

By default, Centos docker image has user field as blank that means by default container will get root user during runtime which should be avoided.

```
$ docker inspect centos
```

While building the docker image we can provide the "test" user the less-privileged user in the Dockerfile as shown below

```
$ cd
$ mkdir test-container
$ cd test-container/
$ cat Dockerfile
FROM centos:latest
RUN useradd test
USER test

root@ubuntu:~/test-container# docker build -t h3abionet .

$ docker images | grep h3abionet
```

When we start the docker container we can see that it gets "test" user and docker inspect command also shows the default user as "test"

```
$ docker run -it h3abionet /bin/bash
[test@########## /]$ whoami
test

$ docker inspect h3abionet
```

- Ensure that containers use only trusted base images
- Ensure that unnecessary packages are not installed in the container
- Ensure images are scanned and rebuilt to include security patches
- Ensure Content trust for Docker is Enabled
- Ensure that HEALTHCHECK instructions have been added to container images
- Ensure update instructions are not use alone in the Dockerfile
- Ensure setuid and setgid permissions are removed
- Ensure that COPY is used instead of ADD in Dockerfiles
- Ensure secrets are not stored in Dockerfiles
- Ensure only verified packages are are installed

## 3.6 Docker Security Operations

This section covers some of the operational security aspects for Docker deployments. These are best practices that should be followed. Most of the recommendations here are just reminders that organizations should extend their current security best practices and policies to include containers.

- Avoid Container Sprawl

The flexibility of containers makes it easy to run multiple instances of applications and indirectly leads to Docker images that exist at varying security patch levels. It also means that you are consuming host resources that otherwise could have been used for running 'useful' containers. Having more than just the manageable number of containers on a host makes the situation vulnerable to mishandling, misconfiguration and fragmentation. Thus, avoid container sprawl and keep the number of containers on a host to a manageable total.

```
$ docker info
```

Few containers can be seen in the docker info command but there are no running containers, the rest containers can be listed using *docker ps* which are not in running state but occupying space on the host and can cause container sprawl.

```
$ docker ps -a
```

It is always advisable to run the docker container with "rm" option so that when you exit the container it gets removed from the host as well

```
$ docker run --rm=true -it h3abionet

$ docker ps -a
```

In order to remove all the non-running containers from the host following command can be used

```
$ docker rm `docker ps --no-trunc -aq`
```

- Avoid image sprawl

## 3.7 Docker Swarm Configuration

This section lists the recommendations that alter and secure the behaviour of the Docker Swarm. If you are not using Docker Swarm, then the recommendations in this section do not apply

- Ensure swarm mode is not Enabled, if not needed
- Ensure that the minimum number of manager nodes have been created in a swarm
- Ensure that swarm services are bound to a specific host interface
- Ensure that all Docker swarm overlay networks are encrypted
- Ensure that Docker's secret management commands are used for managing secrets in a swarm cluster
- Ensure that swarm manager is run in auto-lock mode
- Ensure that the swarm manager auto-lock key is rotated periodically
- Ensure that node certificates are rotated as appropriate
- Ensure that CA certificates are rotated as appropriate
- Ensure that management plane traffic is separated from data plane traffic

# 4.0 Linking Docker Containers

## 4.1 Docker Link Flag

In order to connect together multiple docker containers or services running inside docker container, '--link' flag can be used in order to securely connect and provide a channel to transfer information from one container to another. Let's use a simple application of using a Wordpress container linked to MySQL container.

**Pull the latest MySql container**

```
$ docker pull mysql:latest
```

**Run MySql Container in detach mode**

```
$ docker run --name mysql-container -e MYSQL_ROOT_PASSWORD=wordpress -d mysql
89c8554d736862ad5dbb8de5a16e338069ba46f3d7bbda9b8bf491813c842532
```

Let's run this database container with name "*mysql-container*" and set root password for MySQL container.

**Pull Wordpress docker container**

In the new terminal, pull the official wordpress container

```
$ docker pull wordpress:latest
```

**Run the wordpress container linking it to MySQL Container**

```
$ docker run -e WORDPRESS_DB_PASSWORD=password --name wordpress-container --link mysql-container:mysql -p 8080:80 -d wordpress
```

As, we have linked both the containers; now wordpress container is accessible from browser using the address http://localhost:8080 and setup of wordpress can be done easily.

## 4.2 Docker Compose

Let's run the same previous application of wordpress and MySQL linking in this tutorial but with Docker compose wh ich is a tool use to define and run complex linked applications with docker. With docker compose we can define the entire multi-container application in single file and then the application can be spinned up using one command.

**Create a new project folder**

```
$ mkdir dockercompose
$ cd dockercompose
```

**Create Docker compose file**

Create *docker-compose.yml* with preferred editor having the following contents

```
web:
  image: wordpress
  links:
   - mysql
  environment:
   - WORDPRESS_DB_PASSWORD=sample
  ports:
```

```
  - "127.0.0.3:8080:80"
mysql:
image: mysql:latest
environment:
 - MYSQL_ROOT_PASSWORD=sample
 - MYSQL_DATABASE=wordpress
```

**Get the linked containers up**

```
$ docker-compose up
Creating dockercompose_mysql...
Creating dockercompose_web...
Attaching to dockercompose_mysql, dockercompose_web
mysql | Initializing database
.............
```

Visit the IP address http://127.0.0.3:8080 to see the setup page of the newly created linked wordpress container.

# 5.0 Swarmkit

First let's look at the overall architecture of Swarmkit, a distributed resource manager. This can be bundled to run Docker tasks or other types of Tasks.

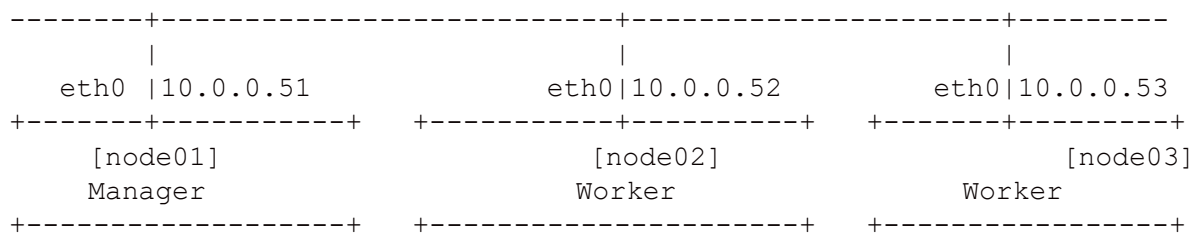
Main Components of Swarmkit

Swarmkit is composed of two types of Nodes:

1. Managers: Responsible for assigning tasks to workers
2. Workers: Place where actual tasks run

## 5.1 Configure Swarm Cluster

**Configure Docker Swarm to create Docker Cluster with multiple Docker nodes.**
There are 2 roles on Swarm Cluster: Manager and Worker nodes. Here, we configure Swarm Cluster with 3 Docker nodes illustrated as follows.




1) Install and run Docker service on all nodes

2) Configure Swarm Cluster on Manager Node

```
$ docker swarm init
```

3) Join in Swarm Cluster on all Worker Nodes

```
$ docker swarm join \
--token <##########################################> 10.0.0.51:2377
```

4) Verify that worker nodes successfully joined cluster

```
$ docker node ls
```

**5)** Next, configure services that the Swarm Cluster provides

Create the same container image on all Nodes for the service first.
For example, let's create a Container image which provides http service on all Nodes

```
$ root@node01:~# vi Dockerfile
FROM ubuntu:20.04

RUN apt-get update
RUN apt-get -y install apache2
RUN echo "node01" > /var/www/html/index.html

EXPOSE 80
CMD ["/usr/sbin/apachectl", "-D", "FOREGROUND"]

root@node01:~# docker build -t apache2_server:latest ./docker node ls
```

6) Configure service on Manager Node

Access the Manager node's Hostname or IP address to verify it works normally. Note that requests to worker nodes are load-balanced with round-robin

Create a service with 2 replicas

```
$ docker service create --name swarm_cluster --replicas=2 -p 80:80 apache2_server:latest
```

Show service list

```
$ docker service ls
```

Inspect the service

```
$ docker service inspect swarm_cluster --pretty
```

Show service state

```
$ docker service ps swarm_cluster
```

Verify it works normally

```
$ curl http://node01
```

```
$ curl http://node01
$ curl http://node01
```

7) If you'd like to change the number of replicas, configure as follows**.**

```
$ docker service scale swarm_cluster=3
```

# 6.0 Jupyter notebook on Docker

Machine Learning and Data Analytics are becoming quite popular for main stream data processing

## 6.1 Numpy

In this section we learn how to run numpy programs on Jupyter which is served from inside a docker container.

**Setup Docker**

Let's assume you have the latest version of docker running on your computer.

**Download Run Docker Jupyter Image**

Run the jupyter/scipy-notebook in the detached mode. Please note the container port 8888 is mapped to host port of 8888.

```
docker run -d -p 8888:8888 jupyter/scipy-notebook
```

You can inspect the container running and get the container id

```
docker ps -a
```

**Get the Security token**

Since the jupyter notebooks from this image have a security token associated, execute the following command to get the token

```
docker exec <CONTAINER ID> jupyter notebook list
```

Output of the command above will give the URL with security token

**Access Jupyter Notebook**

Direct the Host browser at the URL generated

## 6.2 Tensorflow

In this section we learn how to run Tensorflow programs on Jupyter which is served from inside a docker container.

**Setup Docker**

We assume you have the latest version of docker running on your computer.

**Download Run Docker Jupyter Image**

Run the jupyter/scipy-notebook in the detached mode. Please note the container port 8888 is mapped to host port of 8888.

```
docker run -d -p 8888:8888 jupyter/tensorflow-notebook
```

Output of the above command will show the CONTAINER_ID of the container

Let's inspect the container running and get the container id

```
docker ps -a
```

**Get the Security token**

Since the jupyter notebooks from this image have a security token associated, execute the following command to get the token

```
docker exec <CONTAINER ID> jupyter notebook list
```

Output of the command above will give the URL with security token

## Access Jupyter Notebook

Direct the Host browser at the URL generated above