docker

By Anand

Dockerfile

- Dockerfile -> Automated approach for crafting images
- Text file script file that contains special instructions in a sequence to build the right & relevant image from a base image
- Dockerfile can include
  - Base image selection
  - Installing required application
  - Adding configuration & data files
  - Automatically running services & exposing them to external world

Dockerfile

- Docker build subcommand is tightly integrated with the build process and is responsible for transferring build context (along with Dockerfile) to the daemon using CLI interface
- Docker daemon builds the image using the context files (and configuration/data files) passed on to it

Dockerfile Syntax

```
# Comment
INSTRUCTION arguments
```

- Instruction is not case sensitive, however it is advised to use UPPERCASE to distinguish them from arguments

Dockerfile Syntax

```
FROM <image>[:<tag>]
```

- Most important instructions and should be the first valid instruction in the Dockerfile
- <image>: This is the name of the image which will be used as the base image
- <tag>: This is the optional tag qualifier for that image. If any tag qualifier has not been specified, then the tag latest is assumed

```
FROM ubuntu:14.04
```

```
FROM centos
```

Dockerfile Syntax

```
MAINTAINER <author's detail>
```

The MAINTAINER instruction is an informational instruction of a Dockerfile. This instruction capability enables the authors to set the details in an image. Docker does not place any restrictions on placing the MAINTAINER instruction in Dockerfile. However, it is strongly recommended that you should place it after the FROM instruction.

```
MAINTAINER Anand R <anandr72@gmail.com>
```

Dockerfile Syntax

```
COPY <src> ... <dst>
```

- The COPY instruction enables you to copy the files from the Docker host to the filesystem of the new image
- Use absolute path for destination and if no path found, root path (/) is used

```
COPY html /var/www/html
COPY httpd.conf magic /etc/httpd/conf/
```

Dockerfile Syntax

```
ADD <src>... <dest>
ADD ["<src>",... "<dest>"]
```

-> (this form is required for paths containing whitespace)

- The ADD instruction copies new files, directories or remote file URLs from <src> to <dest>
- It can also handle tar/zip files & accepts wild cards

```
ADD web-page-config.tar /
ADD hom* /mydir/        # adds all files starting with "hom"
ADD hom?.txt /mydir/    # ? is replaced with any single character, e.g., "home.txt"
```

Dockerfile Syntax

```
ENV <key> <value>
ENV <key>=<value> ...   -> multiple values
```

- The ENV instruction sets the environment variable <key> to the value <value> in the container
- Multiple key value pairs can also be specified

```
ENV APACHE_LOG_DIR /var/log/apache
ENV JAVA_HOME=/java MVN_HOME=/mvn
```

Dockerfile Syntax

```
USER <UID>|<UName>
```

- The USER instruction sets the start up user ID or user Name in the new image.

```
USER anand
```

Dockerfile Syntax

```
WORKDIR <dirpath>
```

- The WORKDIR instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the Dockerfile
- If the WORKDIR doesn't exist, it will be created even if it's not used in any subsequent Dockerfile instruction

```
WORKDIR /var/log
```

Dockerfile Syntax

```
VOLUME <mountpoint>
```

- The VOLUME instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers

```
VOLUME /data
```

Dockerfile Syntax

```
EXPOSE <port>[/<proto>] [<port>[/<proto>]...]
```

- Informs Docker that the container listens on the specified network ports at runtime

```
EXPOSE 7373/udp 8080
```

Dockerfile Syntax

```
RUN <command>
RUN ["<exec>", "<arg-1>", ..., "<arg-n>"]
```

- Will execute any commands in a new layer on top of the current image and commit the results.
- The resulting committed image will be used for the next step in the Dockerfile

```
RUN echo "echo Welcome to Docker!" >> /root/.bashrc
RUN ["c:\\windows\\system32\\tasklist.exe"]
```

Dockerfile Syntax

CMD ["executable","param1","param2"] (exec form, this is the preferred form)

CMD ["param1","param2"] (as default parameters to ENTRYPOINT)

CMD command param1 param2 (shell form)

- Provides default entry to the container and only one CMD statement is allowed in a Dockerfile. If more, last one gets picked
- RUN executes at build time while CMD executes after the container is built

```
CMD ["echo", "Dockerfile CMD demo"]
```

Dockerfile Syntax

```
ENTRYPOINT ["executable", "param1", "param2"]
ENTRYPOINT command param1 param2 (shell form)
```

- Entry point to the application and can not be overridden by the arguments passed with RUN command

```
ENTRYPOINT ["echo", "Dockerfile ENTRYPOINT demo"]
```

.dockerignore

- Before the docker CLI sends the context to the docker daemon, it looks for a file named .dockerignore in the root directory of the context.
- If this file exists, the CLI modifies the context to exclude files and directories that match patterns in it.
- This helps to avoid unnecessarily sending large or sensitive files and directories to the daemon and potentially adding them to images using ADD or COPY

```
*/temp*
*/log/*
```

Specify hostname to identify containers

- When you have lots of containers, you can not identify which container you are in

```
$docker run –it Ubuntu /bin/bash
root@ca2db7d9cce5#
```

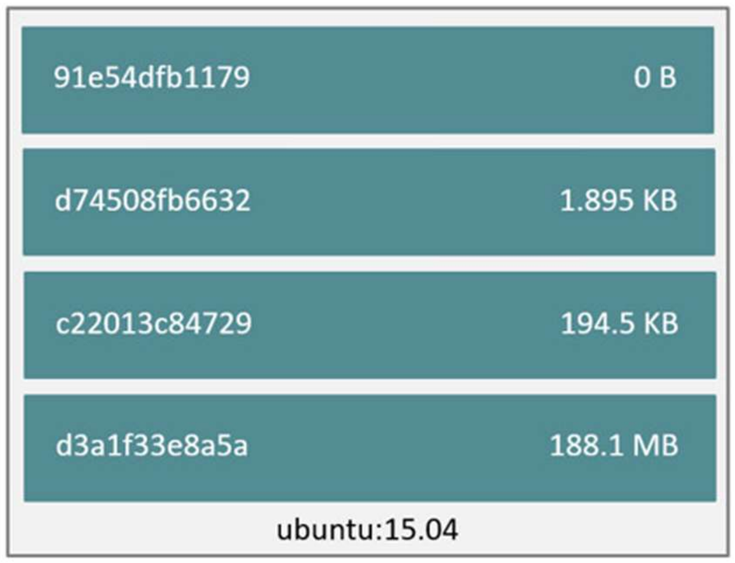
- Specify –hostname option to give a descriptive hostname

```
$docker run -it -h mysql.local ubuntu bash
root@mysql:/# cat /etc/hosts
172.17.0.3      mysql.local mysql -> mysql is an alias to mysql.local
```

Specify hostname to identify containers

- This hostname will be tied to the container and next time when you start the container, the same alias gets picked up
- Exercise : Stop the container, re-start the same container with its ID and exec into it and check if hostname comes up fine.

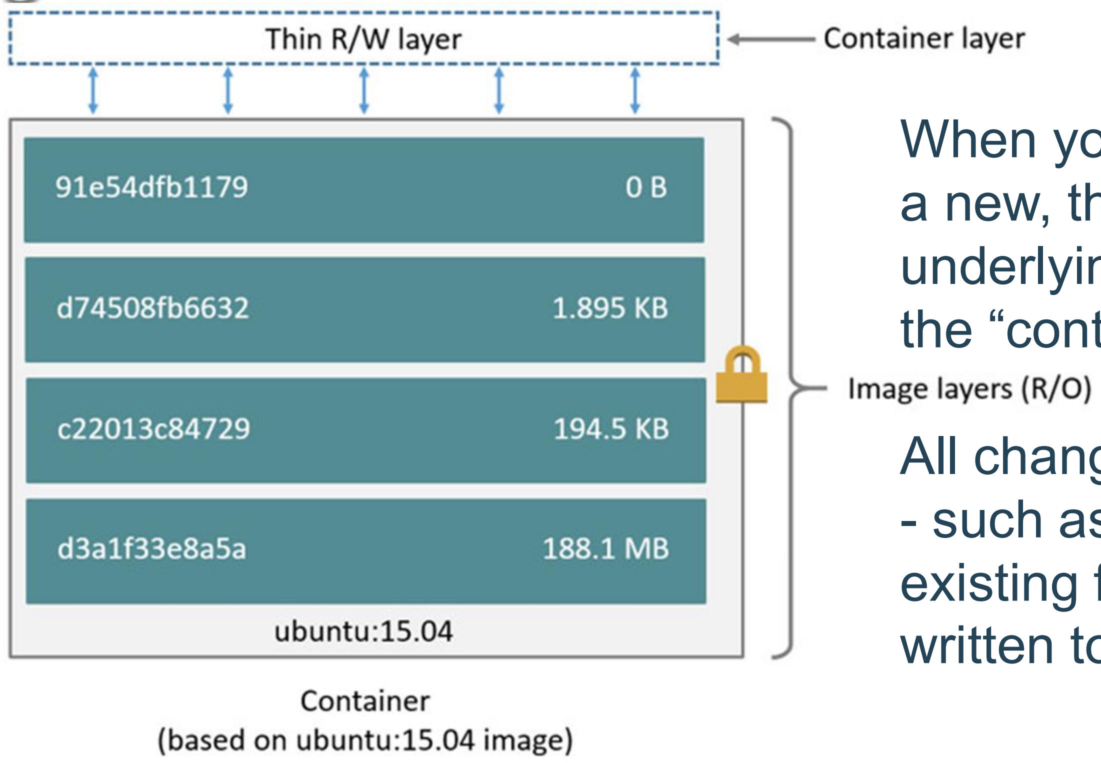
Docker – Understanding Storage



Image

Each Docker image references a list of read-only layers that represent filesystem differences. Layers are stacked on top of each other to form a base for a container’s root filesystem. The diagram shows the Ubuntu 15.04 image comprising 4 stacked image layers.

Docker – Understanding Storage



When you create a new container, you add a new, thin, writable layer on top of the underlying stack. This layer is often called the “container layer”.

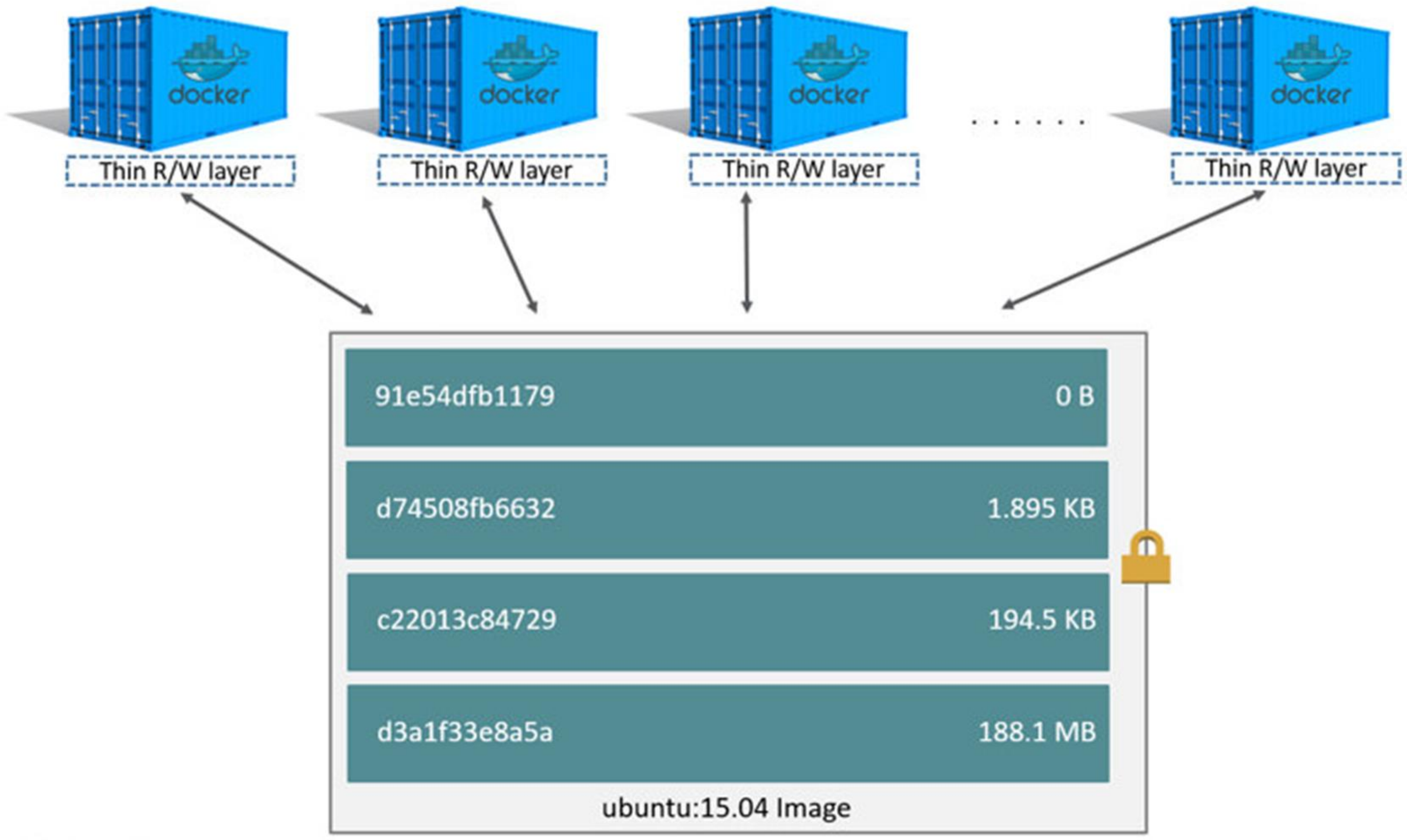
All changes made to the running container - such as writing new files, modifying existing files, and deleting files - are written to this thin writable container layer.

Docker – Container & Layers

The major difference between a container and an image is the top writable layer. All writes to the container that add new or modify existing data are stored in this writable layer. When the container is deleted the writable layer is also deleted.

The underlying image remains unchanged. Because each container has its own thin writable container layer, and all changes are stored in this container layer, this means that multiple containers can share access to the same underlying image and yet have their own data state.

Docker – Understanding Storage



Inspect changed ubuntu

- Write a Dockerfile to create a changed-ubuntu image which just has one extra file in it

```
FROM Ubuntu
RUN echo "Hello World" > /tmp/newfile
```

- Build an image out of it and call it 'changed-ubuntu'

```
$docker build -t changed-ubuntu .
```

- Check the difference between original Ubuntu & changed one

```
$docker history ubuntu
$docker history changed-ubuntu
```

Data Storing & Sharing

- Application data, logs generated and persisted etc need to be stored and shared in real time scenarios
- If stored in Containers, they will be removed once the container is deleted
- docker volumes are specially-designated directory within one or more containers which can be used to persist data within & across containers
- Docker Volumes can be created and attached in the same command that creates a container, or they can be created independent of any containers and then attached later

Docker volume

Creating an independent Volume by name DataVolume1

```
$docker volume create DataVolume1
```

Start a container attaching this volume with –v flag

```
$docker run -ti --rm -v DataVolume1:/datavolume1 ubuntu
```

Write some data to the volume

```
#echo "Example1" > /datavolume1/Example1.txt
```

Since we used –rm flag, container will be deleted upon exit but volume?

```
exit
$docker volume inspect DataVolume1
```

Docker volume

let's start a new container and attach DataVolume1

```
$docker run --rm -ti -v DataVolume1:/datavolume1 ubuntu
```

```
#cat /datavolume1/Example1.txt
```

Exit the container

Docker volume

Creating a volume at run time & attaching it to another container

```
$docker run -ti --name=Container2 -v DataVolume2:/datavolume2 ubuntu
```

Write some data to the volume

```
#echo "Example2" > /datavolume2/Example2.txt
```

```
#cat /datavolume2/Example2.txt
```

Exit the container (we did not run with –-rm option on purpose)

Docker volume

Let us restart the container

```
$docker start -ai Container2
```

Verify if the mount is indeed present and if it has the data

```
$cat /datavolume2/Example2.txt
```

Let us exit and see if we can clean up

```
$docker volume rm DataVolume2
```

Docker does not let us remove any volume as long as it is referenced

Remove the container and them remove DataVolume2

Docker Sharing volumes -–volumes-from

Exercise :

- Create Container4 (without rm) and DataVolume4
- Write a file - /DataVolume4/Example4.txt with one line in it
- Exit the container
- Create Container5 and Mount Volumes from Container4

```
$docker run -ti --name=Container5 --volumes-from Container4 Ubuntu
```

- Append one more line of text to /DataVolume4/Example4.txt
- Exit

Docker Sharing volumes -–volumes-from

Start container4 and verify if changes made by container5 is present or not

Issues?? Concerns??

Docker Sharing volumes --volumes-from Read Only

Docker does not provide any mechanism for file locking and applications will have to handle it themselves

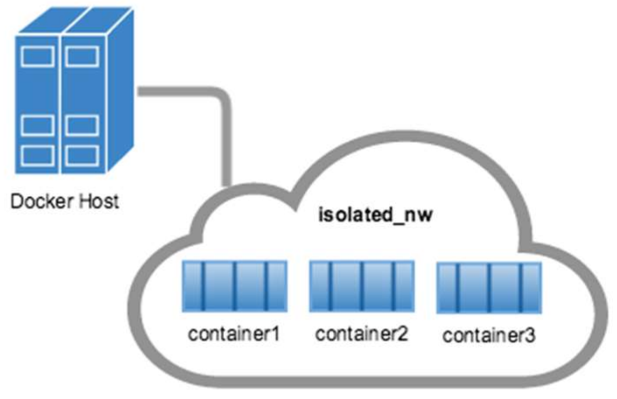
Read Only mount option is available if needed

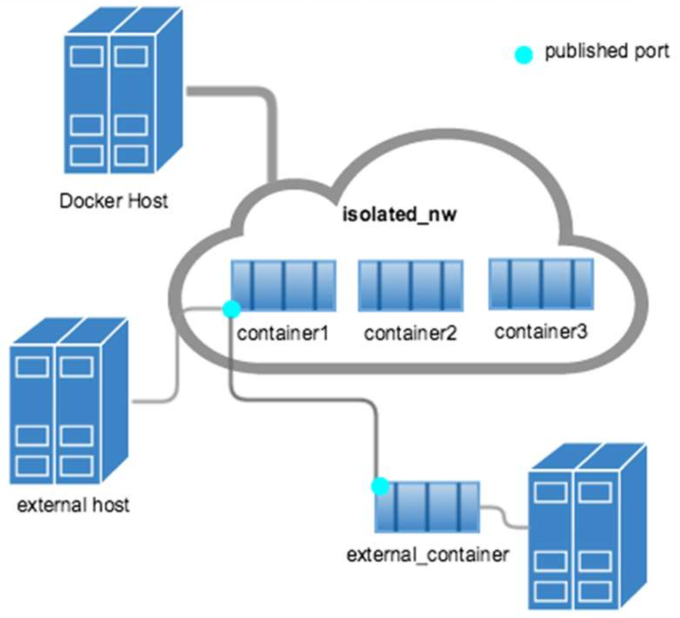
```
$docker run -ti --name=Container6 --volumes-from Container4:ro ubuntu
```

Try to remove volume

```
#rm /datavolume4/Example4.txt
```

Docker Networking





Docker Networking 101

- Docker containers and services need not be aware that they are deployed on Docker or whether their peers are also on Docker and so on and so forth
- As long as each Docker Node or Container is able to discover other nodes and communicate to it, distributed architecture & high availability of Docker networking is achieved

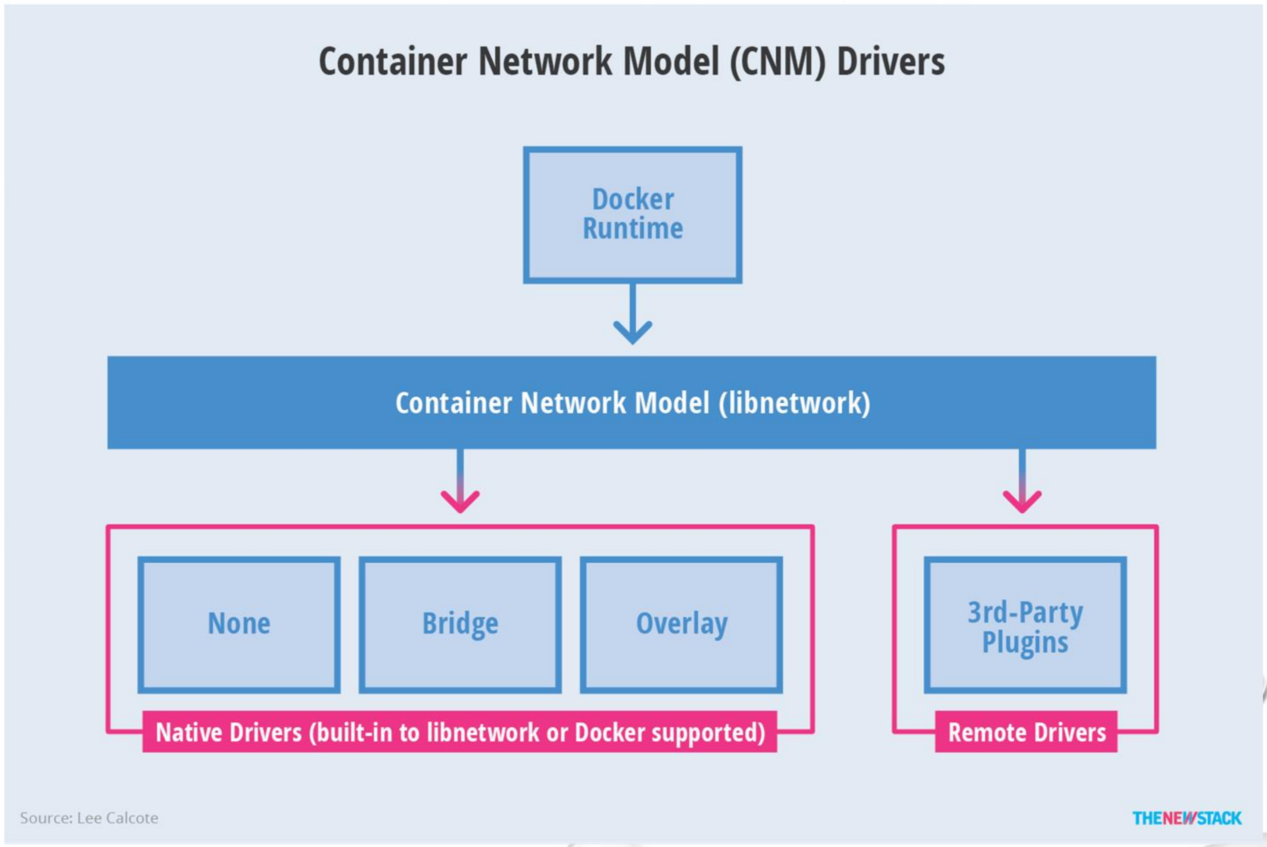
Goals of Container(Docker) Networking

- Flexibility
- Scalability
- User Friendly
- Cross Platform
- Decentralized
- Secure

Container Network Model (CNM)

CNM formalizes

the steps required to provide networking for containers while providing abstraction to support multiple network drivers


Container Network Model (CNM) Drivers
Docker Runtime
Container Network Model (libnetwork)
None
Bridge
Overlay
3rd-Party Plugins
Native Drivers (built-in to libnetwork or Docker supported)
Remote Drivers
Source: Lee Calcote
THENEWSTACK

Docker Network Drivers

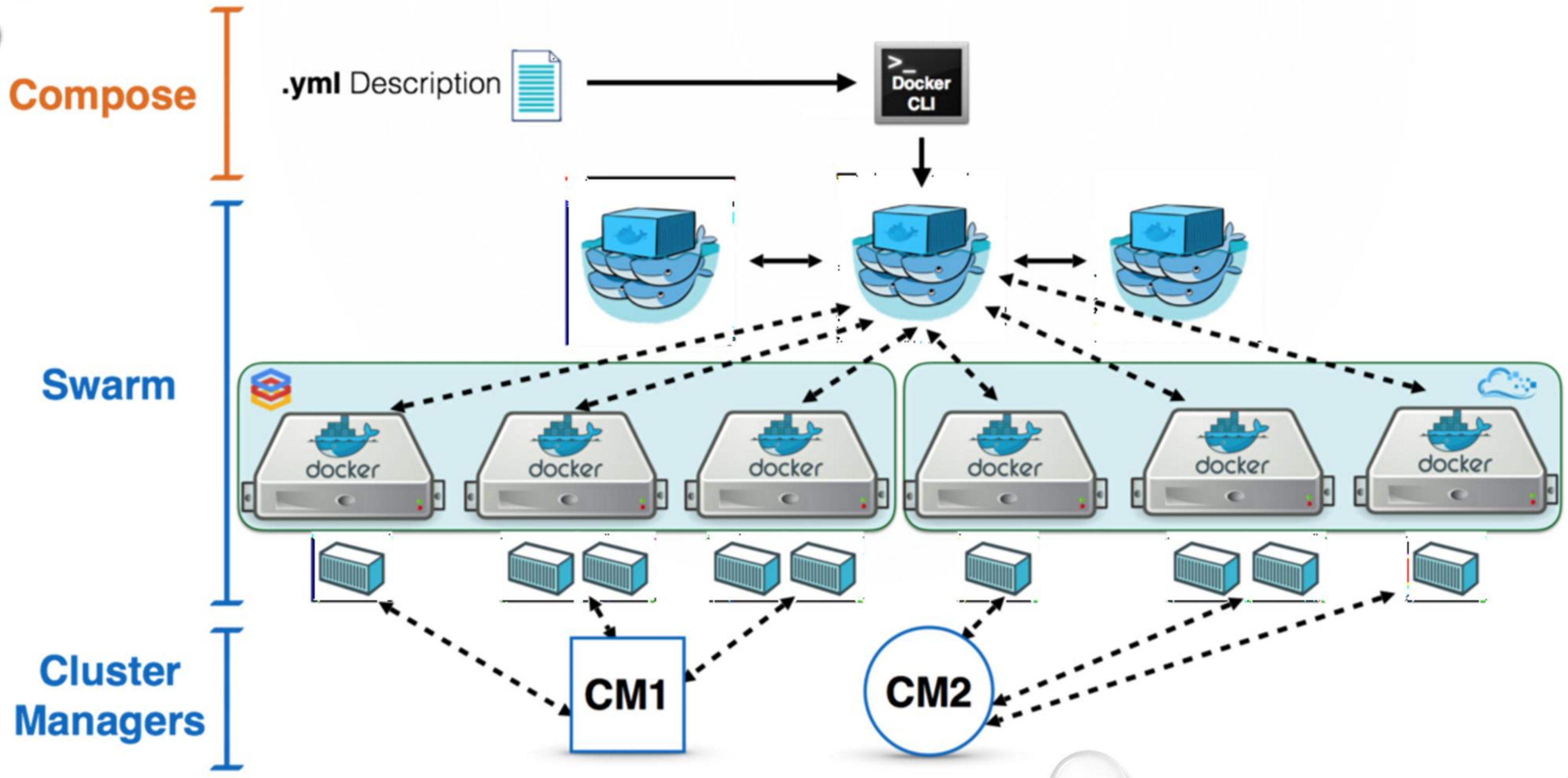
Pluggable subsystem that can provide core network functionality

- Bridge : default driver, used when applications run in standalone containers that need to communicate
- Host : removes network isolation between containers and Docker host and uses hosts networking services directly. Swarm services uses this

Docker Network Drivers

- None : Disable all networking and usually used in conjunction with custom network driver
- Overlay : connect multiple docker daemons together and enable swarm service to communicate with other daemons
- Macvlan : Assign MAC address to a container and make it appear like a physical device on your network

Docker Network Drivers : Swarm



Run Apache Web Server on port 8080

- Start a Ubuntu daemon container mapping port 80 to port 8080 of the host machine, name the container as ‘webserver’

```
$docker run -td --name webserver -p 8080:80 Ubuntu
```

- Check if ports are mapped correctly

```
$docker port webserver
```

- Get into the container bin/bash

```
$docker exec -it webserver /bin/bash
```

- Update Ubuntu source files

```
# apt-get update
```

Run Apache Web Server on port 8080

- Install Apache2 server & vim (text editor)

```
#apt-get install apache2 vim
```

- Navigate to apache root directory to find Document Root

```
#cd /etc/apache
#cat sites-enabled/000-default.conf
```

- Find Document Root which serves web contents

```
#cd /var/www/html
```

- Check if index.html is present

```
#ls
```

Run Apache Web Server on port 8080

- Now start apache2 services

```
#service apache2 restart
```

- Open up a browser and navigate to http://localhost:8080
- Standard Ubuntu apache web page should show up
- Remove index.html & check if you get any contents
- Put in hello.html and check if this content shows up

Questions

?