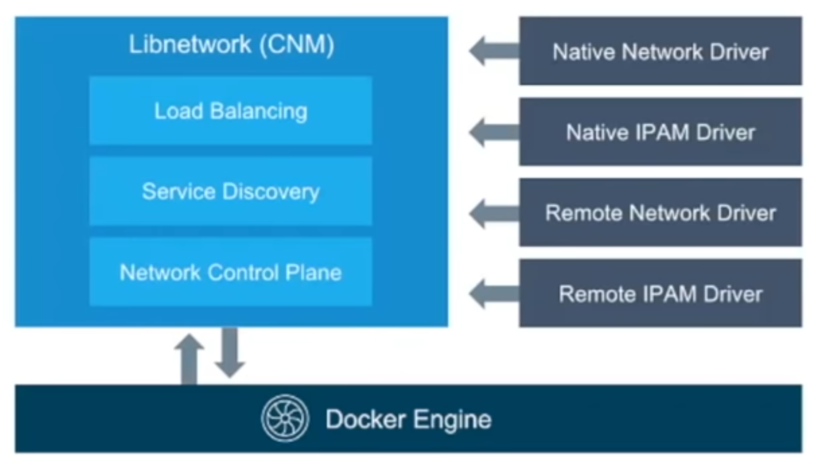
**Docker Networking**: - **Docker networking** is basically used to establish communication between the docker containers and the outside world via host machine or you can say it is a communication passage through which all the isolated containers communicate with each other in various situations to perform the required actions.

Three pillars of docker networking

CNM(container network model) Libnework() Drivers



**CNM**- it’s documentation of networking.

**Libnetwork**- it’s implementation of CNM.

**Driver(s)-** Protocol implementation, such as NAT driver.

**Note-** 1) Whenever docker client want to communicate they use Libnetwork concept. 2) Volume mapping is possible because of Libnetwork.

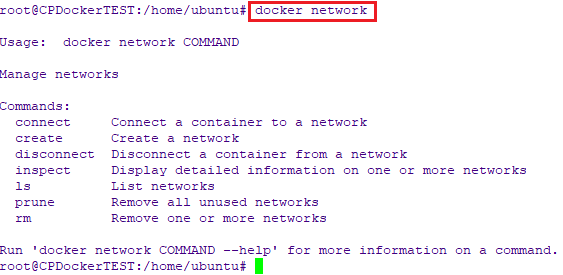
**How to check connectivity between host?** – Ping command.

**Explaining Docker Networking Concepts**

All commands listed below are tested with **root** privileges on **Ubuntu**.

To manage network operations, like creating a new network, connecting a container to a network, disconnect a container from the network, listing available networks and removing networks etc., we use the following command:

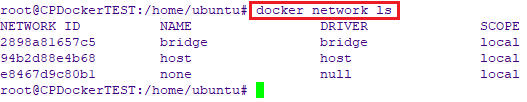
# docker network

[](https://www.ostechnix.com/wp-content/uploads/2019/10/docker-network-command.png)

**Types of docker network drivers**

To list all your networks, run:

# docker network ls

[](https://www.ostechnix.com/wp-content/uploads/2019/10/List-docker-networks.png)

Let’s have some short introduction on all of them.

1. **Bridge network :** When you start Docker, a default bridge network is created automatically. A newly-started containers will connect automatically to it. You can also create user-defined custom bridge networks. User-defined bridge networks are superior to the default bridge network.
2. **Host network :** It remove network isolation between the container and the Docker host, and use the host’s networking directly. If you run a container which binds to port 80 and you use host networking, the container’s application is available on port 80 on the host’s IP address. Means you will not be able to run multiple web containers on the same host, on the same port as the port is now common to all containers in the host network.
3. **None network :** In this kind of network, containers are not attached to any network and do not have any access to the external network or other containers. So, this network is used when you want to completely disable the networking stack on a container.
4. **Overlay network :** Creates an internal private network that spans across all the nodes participating in the swarm cluster. So, overlay networks facilitate communication between a docker swarm service and a standalone container, or between two standalone containers on different Docker Daemons.
5. **Macvlan network :** Some applications, especially legacy applications or applications which monitor network traffic, expect to be directly connected to the physical network. In this type of situation, you can use the Macvlan network driver to assign a MAC address to each container’s virtual network interface, making it appear to be a physical network interface directly connected to the physical network.

**Casestudy: Enabling communication between container in single host**

root@ip-172-31-23-238:~# docker run -d --name c1 --network mynet jenkins sleep 1d 07c7122bb75528efd9468a5e5268bc250b362be68dec953547f120b8e697716f

root@ip-172-31-23-238:~# docker run -d --name c2 --network mynet jenkins sleep 1d cd6168d5fd4cc6e2c3c987ac252ca323e11c559ca37e4552b6e6c02593109c3b

root@ip-172-31-23-238:~# docker exec -it p1 /bin/bash

jenkins@07c7122bb755:/$ ping p2 Unable to connect

root@ip-172-31-23-238:~# docker network create mynet 1ddc99f170ac1b2acc6e12be97475533a0e41b4c0dd12d04213b12bebffee4c8

root@ip-172-31-23-238:~# docker run -d --name p1 --network mynet jenkins sleep 1d 07c7122bb75528efd9468a5e5268bc250b362be68dec953547f120b8e697716f

root@ip-172-31-23-238:~# docker run -d --name p2 --network mynet jenkins sleep 1d cd6168d5fd4cc6e2c3c987ac252ca323e11c559ca37e4552b6e6c02593109c3b

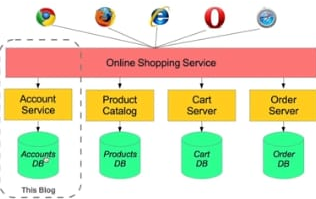
root@ip-172-31-23-238:~# docker exec -it p1 /bin/bash

jenkins@07c7122bb755:/$ ping p2 PING p2 (172.18.0.3) 56(84) bytes of data.

64 bytes from p2.mynet (172.18.0.3): icmp\_seq=1 ttl=64 time=0.066 ms 64 bytes from p2.mynet (172.18.0.3): icmp\_seq=2 ttl=64 time=0.062 ms 64 bytes from p2.mynet (172.18.0.3): icmp\_seq=3 ttl=64 time=0.060 ms

**Why we need to enable communication by name:** This concept is called service discover by name, if we do such things then it will be easy for us to work with driver.

Docker compose: Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.



Install Compose on Linux systems

On Linux, you can download the Docker Compose binary from the [Compose repository release page on GitHub](https://github.com/docker/compose/releases). Follow the instructions from the link, which involve running the curl command in your terminal to download the binaries. These step-by-step instructions are also included below.

For alpine, the following dependency packages are needed: py-pip, python3-dev, libffi-dev, openssl-dev, gcc, libc-dev, rust, cargo and make.

Step1: Run this command to download the current stable release of Docker Compose:

**sudo curl -L "https://github.com/docker/compose/releases/download/1.28.5/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose**

Note:To install a different version of Compose, substitute 1.28.5 with the version of Compose you want to use.

If you have problems installing with curl, see [Alternative Install Options](https://docs.docker.com/compose/install/#alternative-install-options) tab above.Apply executable permissions to the binary:

**sudo chmod +x /usr/local/bin/docker-compose**

Note: If the command docker-compose fails after installation, check your path. You can also create a symbolic link to /usr/bin or any other directory in your path.

**sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose**

$ docker-compose --version

docker-compose version 1.28.5, build 1110ad01

be understandable even if you’re not familiar with it.

## Prerequisites

Make sure you have already installed both [Docker Engine](https://docs.docker.com/get-docker/) and [Docker Compose](https://docs.docker.com/compose/install/). You don’t need to install Python or Redis, as both are provided by Docker images.

## Step 1: Setup

Define the application dependencies.

Task1 -Create a directory for the project:

mkdir composetest

cd composetest

Create a file called app.py in your project directory and paste this in:

import time

import redis

from flask import Flask

app = Flask(\_\_name\_\_)

cache = redis.Redis(host='redis', port=6379)

def get\_hit\_count():

retries = 5

while True:

try:

return cache.incr('hits')

except redis.exceptions.ConnectionError as exc:

if retries == 0:

raise exc

retries -= 1

time.sleep(0.5)

@app.route('/')

def hello():

count = get\_hit\_count()

return 'Hello World! I have been seen {} times.\n'.format(count)

TASK-2: Create another file called requirements.txt in your project directory and paste this in: flask redis

## TASK-3: Create a Dockerfile

FROM python:3.7-alpine

WORKDIR /code

ENV FLASK\_APP=app.py

ENV FLASK\_RUN\_HOST=0.0.0.0

RUN apk add --no-cache gcc musl-dev linux-headers

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

EXPOSE 5000

COPY . .

CMD ["flask", "run"]

This tells Docker to:

* Build an image starting with the Python 3.7 image.
* Set the working directory to /code.
* Set environment variables used by the flask command.
* Install gcc and other dependencies
* Copy requirements.txt and install the Python dependencies.
* Add metadata to the image to describe that the container is listening on port 5000
* Copy the current directory . in the project to the workdir . in the image.
* Set the default command for the container to flask run.

For more information on how to write Dockerfiles, see the [Docker user guide](https://docs.docker.com/develop/) and the [Dockerfile reference](https://docs.docker.com/engine/reference/builder/).

## TASK 4: Define services in a Compose file

Create a file called docker-compose.yml in your project directory and paste the following:

version: "3.9"

services:

web:

build: .

ports:

- "5000:5000"

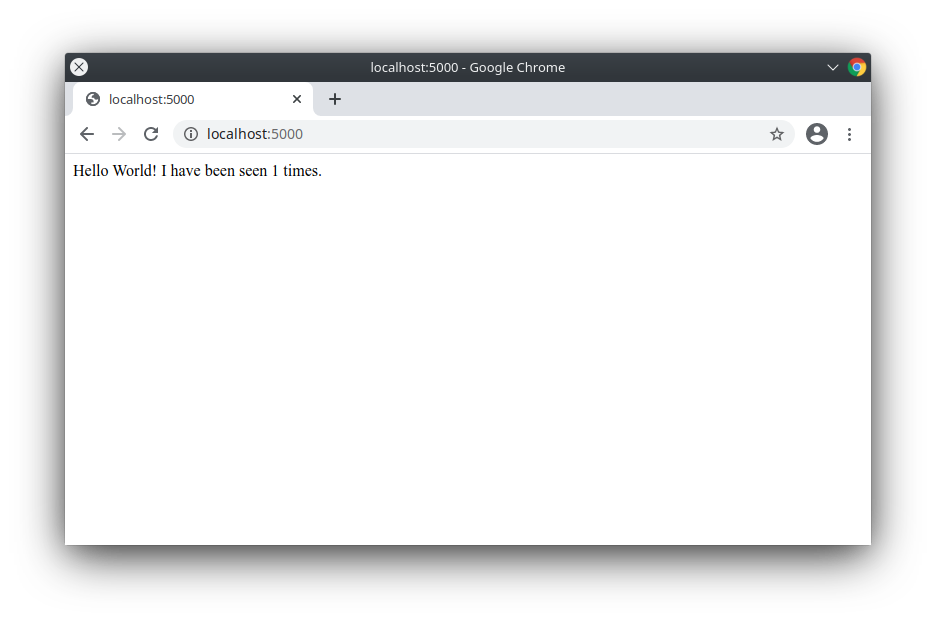
redis:

image: "redis:alpine"

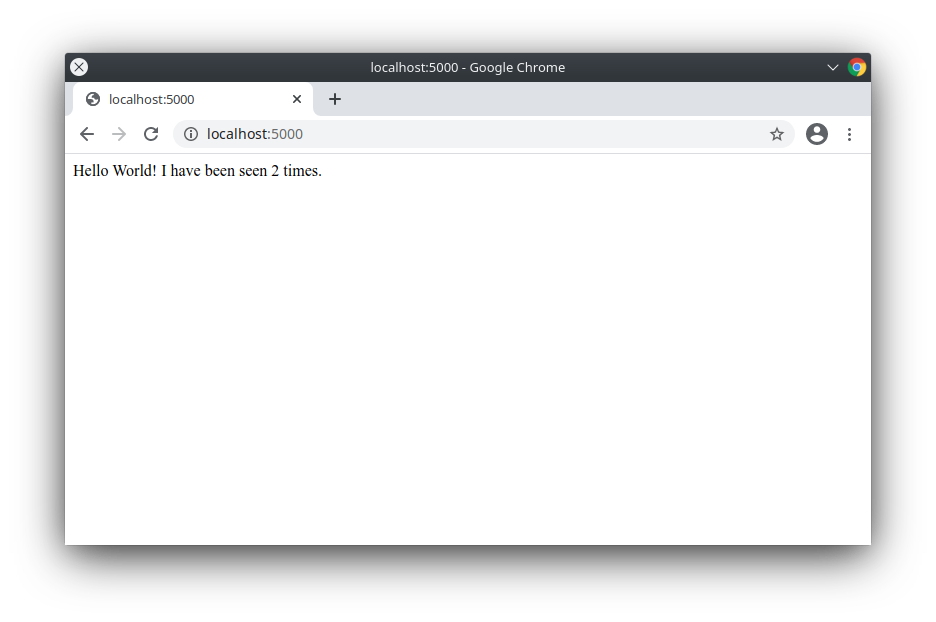
## TASK5: Build and run your app with Compose

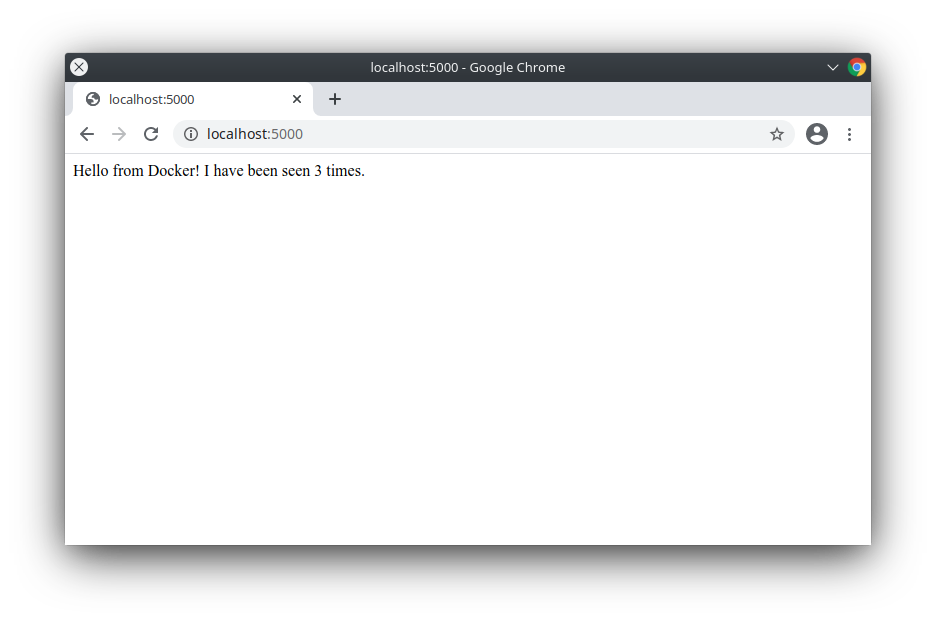
docker-compose up

Verify by using http:host-name:5000



Refresh the page.

The number should increment.



## TASK6: Experiment with some other commands

docker-compose up -d

docker-compose ps

docker-compose stop

Repository: <https://github.com/JanbaskDevops/dockerCompose.git>

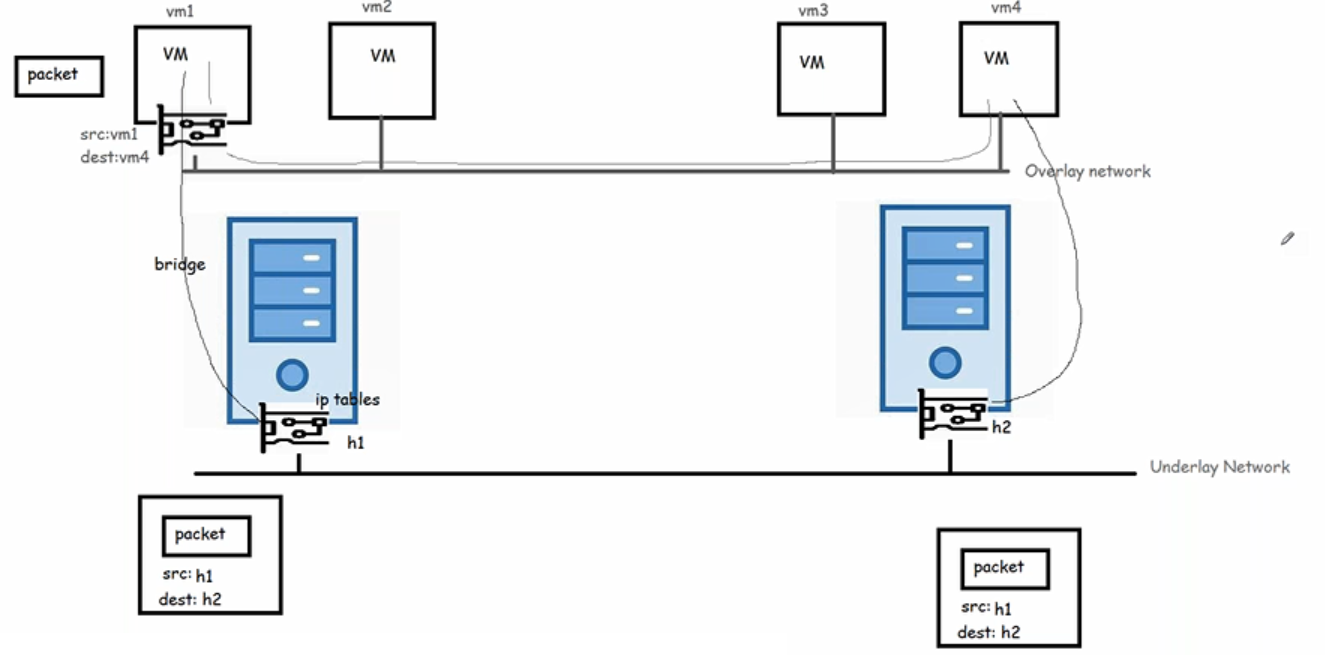
Docker compose: <https://github.com/docker/compose/releases>

**Docker Multi host networking**

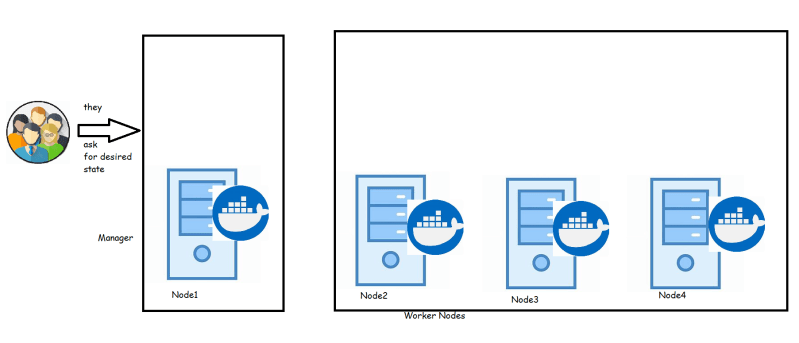
In Docker Overlay network driver provides multi host networking.

Underlay network docker: Two host connected in a same network, and they have ping connectivity with them.

Overlay Network: It’s Virtual network for container.



* To create a overlay network in docker we use docker swarm.
* Docker Swarm gives management & Orchestration features
* In Docker Swarm we specify desired state (i want some application to be run in two containers and swarm will try to maintain the state)



## Docker Swarm

* The cluster maanagement & Orchestration features are embedded inside Docker Engine.
* Docker swarm consists of multiple docker hosts which run in swarm mode.
* Two Roles **managers** and **workers** exist in Docker swarm
* **Manager** is responsible for membership & delegation
* **Worker** is responsible for running swarm services
* Each Docker Host can be a manager, a worker or both.
* In Docker Swarm **Desired State** is maintained. For instance if you are running one container in swarm on a particular node (Worker) and that node goes down, then Swarm schedules this nodes task on other node to maintain the state.
* **Task** is a running container which is part of swarm service managed by Swarm Manager

### Nodes

* It is instance of the docker engine participating in Swarm.
* There are two kinds of nodes
  + Manager nodes:
    - You communicate to manager node to deploy applications in the form of Service Definitions.
    - Manager nodes dispatch unit of work called as tasks to the Worker ndoes
  + Worker nodes:
    - They receive & execute the tasks dispatched from manager nodes.
    - An agent runs on the worker node & reports on the tasks assigned to it

### Services and tasks

* Service is the definition of the task to be executed.
* Typically it would be the application to be deployed.
* Two kinds of Service models are available
  + **Replicated Services model**: In this case swarm manager distributes a specific number of replica task among the nodes based upon the scale you set in the desired state
  + **Global Services Model**: In this case swarm runs one task for the service on every available node in the cluster.

**Task**

* + carries a Docker container and the commands to run inside the container.
  + It is the atomic scheduling unit of swarm.
  + Once a task is assigned to node, it cannot move to another node.
  + It can only run on the assigned node or fail.

### Swarm Setup

* In this series, I would be using 3 ubuntu 18 machines.
* One would be manager & other two would be workers.
* Install docker on all the machines.
* Login into ssh session of the machie which would be manager.
* Ensure all the machines can be communicated (or pingable from manager)
* Make a note of private ip address of the manager (In this example the managers ip address would be 172.31.42.125) and then exec

docker swarm init --advertise-addr <Manager-ip>

# In my case this is

docker swarm init --advertise-addr 172.31.42.125

##Outpu###

Swarm initialized: current node (uyclb1gbhhqhlo80aq7zhx2z4) is now a manager.

To add a worker to this swarm, run the following command:

docker swarm join --token SWMTKN-1-1w51ouq6zrmts85l71z53ruqcc1pivzprpigdodspu58o7dp3z-172clbo51xq8w7uwizm8cc19t 172.31.42.125:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

* Execute docker info on the manager and observe the output should consists of Swarm: active and other info about Docker Swarm.
* Execute command docker node ls and you should see the status of the manager node
* Now login into other nodes and execute docker swarm join command which is output of the docker swarm init command as mentioned above.

docker swarm join --token SWMTKN-1-1w51ouq6zrmts85l71z53ruqcc1pivzprpigdodspu58o7dp3z-172clbo51xq8w7uwizm8cc19t 172.31.42.125:2377

##Output##

This node joined a swarm as a worker.

* Now ssh into the manager and execute docker node ls and you should be able to see three nodes information.
* Lets create a tomcat service by using the following command

docker service create --replicas 2 --name tomcat tomcat:8

* This command leads to creation of tasks and output would be like

tvdml6nt5dryszozaydr8sv8o

overall progress: 2 out of 2 tasks

1/2: running [==================================================>]

2/2: running [==================================================>]

verify: Service converged

* Execute docker service ls and the possible output would look like

ID NAME MODE REPLICAS IMAGE PORTS

tvdml6nt5dry tomcat replicated 2/2 tomcat:8

Let’s inspect the service using docker service inspect --pretty tomcat and the output of the command would be

ID: tvdml6nt5dryszozaydr8sv8o

Name: tomcat

Service Mode: Replicated

Replicas: 2

Placement:

UpdateConfig:

Parallelism: 1

On failure: pause

Monitoring Period: 5s

Max failure ratio: 0

Update order: stop-first

RollbackConfig:

Parallelism: 1

On failure: pause

Monitoring Period: 5s

Max failure ratio: 0

Rollback order: stop-first

ContainerSpec:

Image: tomcat:8@sha256:bb4ceffaf5aa2eba6c3ee0db46d863c8b23b263cb547dec0942e757598fd0c24

Init: false

Resources:

Endpoint Mode: vip

* Execute docker service ps tomcat command to findout on which node the tasks are executed.
* Lets scale the number of containers running tomcat by using the following command docker service scale tomcat=4 and the output would be

tomcat scaled to 4

overall progress: 4 out of 4 tasks

1/4: running [==================================================>]

2/4: running [==================================================>]

3/4: running [==================================================>]

4/4: running [==================================================>]

verify: Service converged

* Service can be deleted using the follwing command docker service rm tomcat

### Rolling updates to docker swarm

* Execute the following command to deploy jenkins

docker service create --name jenkins --replicas 2 --update-delay 10s --publish published=8081,target=8080 jenkins:1.609.1

* Now navigate to ipaddress of any node and http:8081
* Now lets try to update to the newer version of jenkins

docker service update --image jenkins:latest jenkins

### Relevance to Docker Networking

* In this series so far we were able to run docker containers on different nodes from swarm manager.
* Now execute this command docker network ls on the manager and you should see the output which would look like

NETWORK ID NAME DRIVER SCOPE

4fa602b7a4ec bridge bridge local

c3ba8230b575 docker\_gwbridge bridge local

436956ab2dd0 host host local

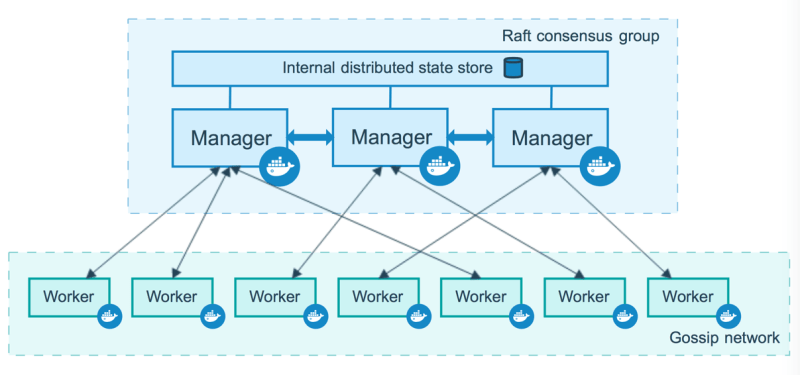
m6zbhh5cn5ag ingress overlay swarm

dfa793d07248 none null local

* In this overlay network driver is used and scope for that driver is swarm.
* So we can conclude that Docker swarm uses overlay and bridge (docker\_gwbridge) to enable multiple Docker Host Communications.

### How Docker Swarm Works?

* Docker swarm uses **RAFT** Consensus Algorithm to maintain a consistent internal state of the entire swarm and all the services running on it.



#### Manager Nodes

* Manager nodes handle cluster management tasks
  + cluster state management
  + service scheduling
  + serving Swarm mode

To take advantage of swarm mode’s fault-tolerance features, Docker recommends you implement an odd number of nodes according to your organization’s high-availability requirements. When you have multiple managers, you can recover from the failure of a manager node without downtime.

\* A three-manager swarm tolerates a maximum loss of one manager.

\* A five-manager swarm tolerates a maximum simultaneous loss of two manager nodes.

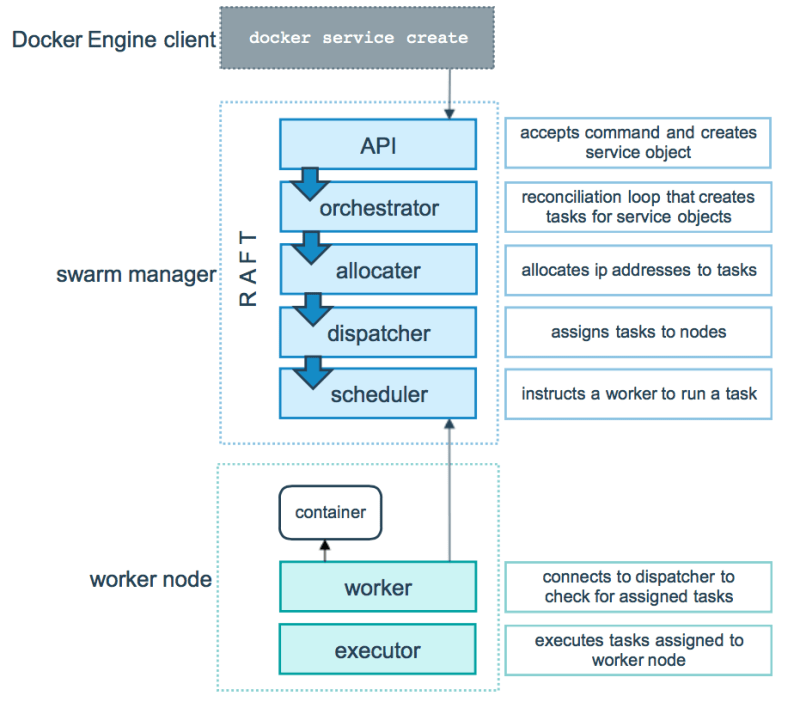
\* An N manager cluster tolerates the loss of at most (N-1)/2 managers.

\* Docker recommends a maximum of seven manager nodes for a swarm.

#### Worker Nodes

* Instance of Docker Engine whose purpose is to execute containers
* They dont participate in Raft distributed state or any of the managers tasks.
* Worker Node can be made Manager node by using docker node promote. note: This has to be executed by Manager Node

### Docker Service Lifecycle



### External Access For Docker Services

#### Ingress Mode Service Publishing

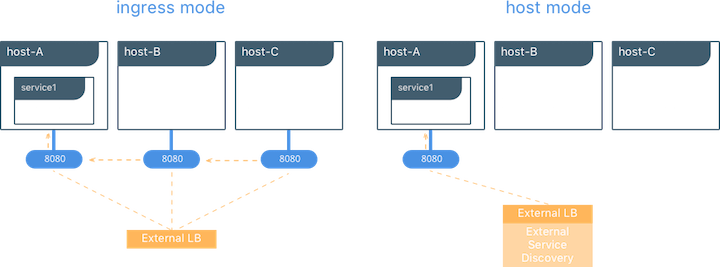
docker service create --replicas 2 --publish mode=ingress,target=80,published=8080 nginx

* This mode publishes the exposed port on every Swarm node.
* Load balancing happens in this mode

#### Host Mode Service Publishing

docker service create --replicas 2 --publish mode=host,target=80,published=8080 nginx

* In this mode the published port is exposed on the host where this service is running
* Load balancing doesn’t happen



**Installing Docker on Windows Server**

* Let’s create a Windows Server 2016 instance in any cloud
* Docs from Microsoft(https://docs.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/deploy-containers-on-server)
* Launch PowerShell as admin and execute the following commands

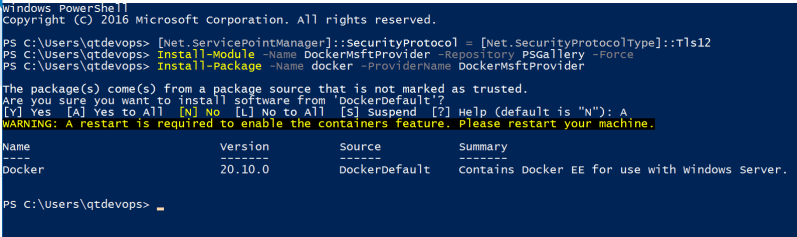
Set-ExecutionPolicy unrestricted

[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12

Install-Module -Name DockerMsftProvider -Repository PSGallery -Force

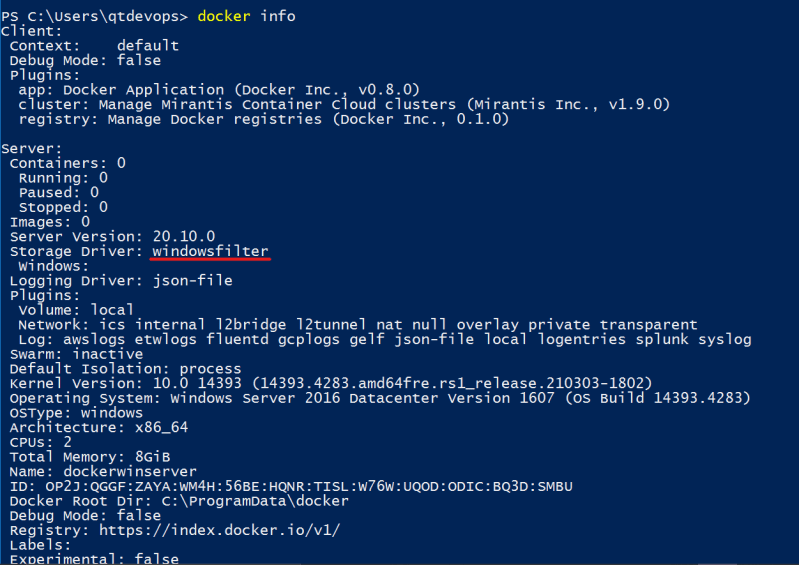
Install-Package -Name docker -ProviderName DockerMsftProvider

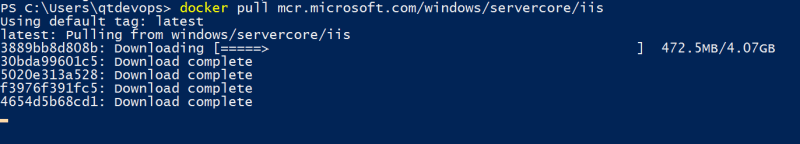
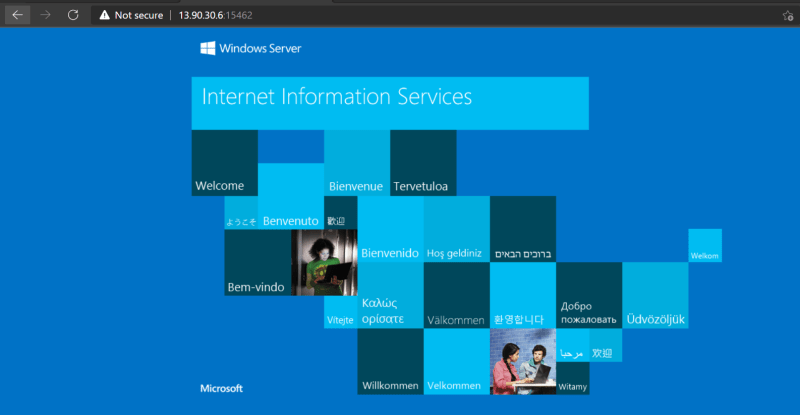
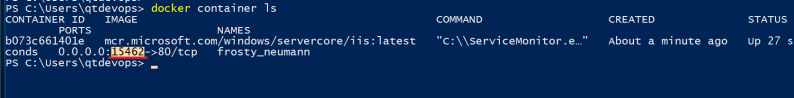
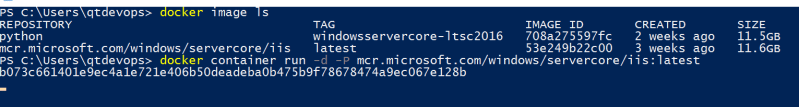
Restart-Computer -Force



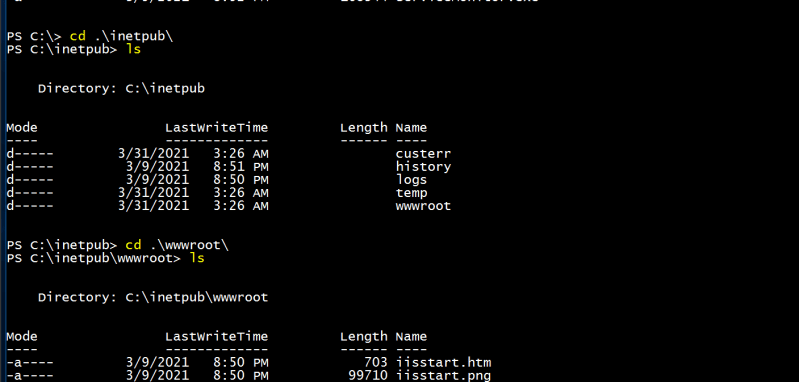
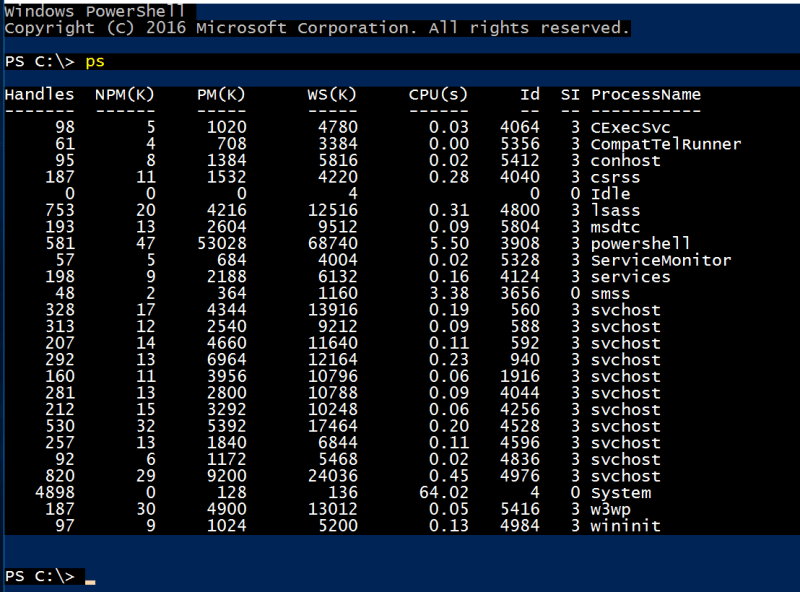
* Post machine restart, Launch PowerShell as admin

docker info



* Now lets try to pull iis server image 
* Now lets run the container 
* Run the following command

docker container exec -it <container id/name> powershell



* Launch powershell as admin & then execute docker info