DOCKER SWARM

→ It is a container orchestration tool.

→ To run all the applications in live environment without down time is called container orchestration.

Scenario:

→ You have created code for application and deployed into tomcat container where your team of developers and testers can access.

→ In your organisation the dev and testing environment has less members for accessing that application, it won't create a problem

→ If a container crashes we can replace because all the team members will wait where as coming to live environment when application is exposed to the customers, if any container crashes or behaving as abnormal (hanging, not poping)

it creates big problem, because the the customers are running with downtime it creates service problem.

→ To overcome all the problems in the live environment without downtime, the container orchestration has to done and maintained.

Scenario:

→ when you install tomcat container, lets say it can withstand 100 users, if the users are increasing it can't withsand.

→ For that we are creating another container on which the same tomcat software is running. now it can balance the load of more 100 users, like wise we can increase the containers on which the same application tomcat is running

Cluster :- The hardware resources of two computers are merged as a single computer.

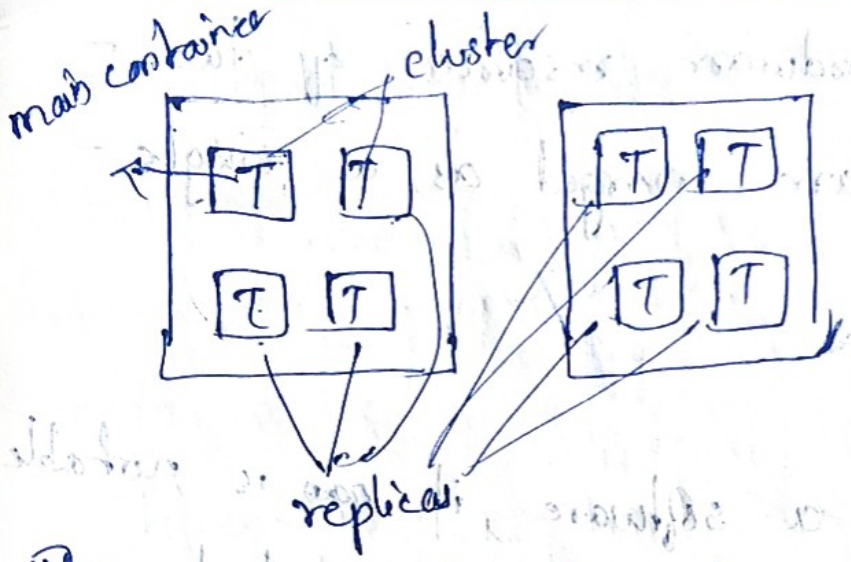
eg :-

we have a software, it is notable install on one computer due to lack of hardware resources, then take one more computer and install the same software on both, these both computers form a cluster.

Ily

→ I have created one tomcat container, on which tomcat software is installed, being tomcat software is big, one container is unable to withstand, so I am creating same another container on which the tomcat software is installed, in this way can form clusters.

Then the load is balanced due to the cluster.



These are 8 tomcat containers where only one tomcat application is running. therefore these 8 tomcat containers are called as replicas

Replicas :- Multiple containers where only one software is running.

→ when we are increasing the replicas we are able to perform load balancing

Load balancing :-

They have distributed their application on multiple replicas, is called load balancing

eg :- when millions of users are accessing

one application, they application can't withstand the traffic, so with the help of load balancing balancing we over come that problem.

Scaling :-

In the background depending on the requirement they are increasing the capability or decreasing the capability is known as scaling.

eg :- For amazon big sale offer starts, at that time when it is started there is big up search, millions of users are accessing it, in the background they are increasing the capability to withsand.

Lets say they has 3 servers with 4 replica on each servers initially. when they think that load is increasing they increase the servers and replicas, when the up search is completed they delete the increased servers and replicas, this is called upscaling and down scaling

Rolling update operations! :-

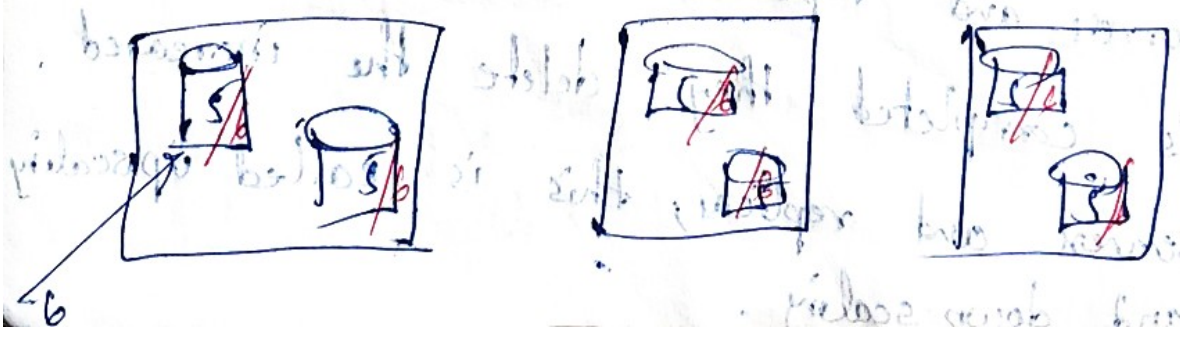
For migration of software, we use rolling update operations.

Scenario!

Lets say HDFC bank, ~~wants to~~ created they database on mysql. they has 6 replicas and 3 servers. They want to update the version from '5.0' to '6.0',

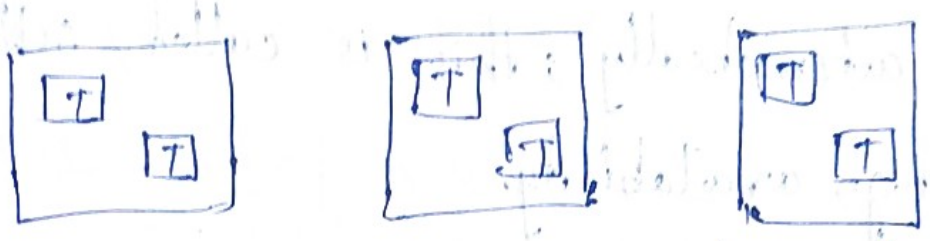
→ Mean while they don't want to have down time for the customers. So they identify the when the less no. of transcations are happening. based on that they trigger rolling update

→ First they will update one database containers mean while customers are doing transcations with other container, In this way they update all the servers and replicas



High availability :

Let's say we have designed 6 tomcat containers on 3 servers,



the 6 container replicas are called as desired state

→ See if the orchestration tool maintains the desired state whatever it happens.

→ Suppose one container in the server 3 is crushed, it will be automatically created because to maintain the desired state.

→ If the server 3 is crushed the no. of containers on the server 3 are automatically created on server 2.

→ If server 2 and server 3 are crushed, the containers on these are replicated on server 1.

→ This is called as high availability

→ The moment there is a mismatch between actually count and desired count, the container orchestration tool will create another one automatically; this is called selfhealing in high availability.

Container orchestration tools

1) Docker Swarm
2) Kubernetes
3) Openshift
4) Mesos
5) ECS

→ Handling docker containers in production is all about container orchestration, this can be done with above tools.

Installation of docker swarm on Manager, worker1 and worker2

1) Create 3 instances on AWS and install docker on them
2) Name them as Manager, worker1 and worker

```
vim /etc/hostname
```

(or)

```
sudo hostnamectl set-hostname name.of the host
```

3) Restart

```
init 6
```

4) To initialize the manager with docker swarm

Connect to the manager

```
docker swarm init
```

This will create a docker swarm and it will generate a token ID.

5). Copy and paste the token ID in worker1 and worker2.

Communication between docker machine and docker hosts.

→ Tcp port 2376 for secure docker client communication. This port is required for docker machine to work. Docker machine is used to orchestrate docker hosts.

→ Tcp port 2377. This port is used for communication between the nodes of a docker swarm or cluster. It only needs to be opened on manger nodes.

→ Tcp and Udp port 7946 for communication among nodes (container network discovery).

→ UDP port 4789 for overlay network traffic (container ingress networking)

→ To see no. of workers in a manager

```
docker node ls
```

Load Balancing :-

→ Each docker containers has a capability to sustain a specific user load. To increase this capability we can increase the no. of replicas (containers) on which a service can run.

Scenario :-

① Create nginx with 5 replicas and check where these replicas are running.

→ Create nginx with 5 replicas

```
docker service create --name webserver
-p 8888:80 --replicas 5 nginx
```

→ To check the services running in swarm

```
docker service ls
```

→ To check where these replicas are running

```
docker service ps webserver
```

→ To access the nginx from browser.

```
public-ip-of-manager/worker1/worker2 : 8888
```

→ To delete the service with all replicas

```
docker service rm webserver
```

Scenario1 ←

* Create mysql with 3 replicas and also pass the necessary environment variables

```
docker service create --name mydb --replicas
3 -e MYSQL_ROOT_PASSWORD=sudarshan
mysql:5
```

→ To check if 3 replicas of mysql are running.

```
docker service ps mydb
```

Scaling :-

This is the process of increasing the no. of replicas or decreasing the replicas count based on requirement without the end user experiencing any downtime.

Scenario

* Create tomcat with 4 replicas and scale it upto 8 and scale down to 2.

→ Create tomcat with 4 replicas

```
docker service create --name appserver
-p 9090:8080 --replicas 4 tomcat
```

→ Check if 4 replicas are running

```
docker service ps appserver
```

→ Increase the replicas count to 8

```
docker service scale appserver=8
```

→ Check if 8 replicas are running

```
docker service ps appserver
```

→ Decrease the replicas count to 2

```
docker service scale appserver=2
```

→ Check if 2 replicas are running

```
docker service ps appserver
```

Rolling updates:-

Services running in docker swarm should be updated from one version to other version without the end user downtime.

Scenario:-

* Create redis:3 with 5 replicas and later update it to redis:4 also rollback to redis:3

→ Create redis:3 with 5 replicas

```
docker service create --name myredis --replicas
5 redis:3
```

→ Check if all 5 replicas are running

```
docker service ps myredis
```

→ Perform a rolling update from redis:3 to redis:4

```
docker service update --image redis:4 myredis
```

→ Check redis:3 replicas are shut down and in this place redis:4 replicas are running

```
docker service ps myredis
```

→ Roll back from redis:4 to redis:3

```
docker service update --rollback myredis
```

→ Check if redis:4 replicas are shut down and in its place redis:3 is running

```
docker service ps myredis
```

→ To remove a worker from the swarm

```
docker node update --availability drain worker_name
```

→ To make the same worker rejoin the swarm

```
docker node update --availability active worker_name
```

→ To make worker2 leave the swarm

Connect to worker2

```
docker swarm leave
```

→ To generate the tokenid for a machine to join swarm as worker

```
docker swarm join-token worker
```

→ To generate the tokenid for a machine to join swarm as manager

```
docker swarm join-token manager
```

→ To promote 'worker1' as a manager

```
docker node promote worker1
```

→ To demote 'worker1' as back to a worker state

```
docker node demote worker1
```

→ To remove the worker1 permanently

```
docker node rm worker1
```

High availability :-

→ If any container/server is crashed, the containers on that server are recreated automatically in the next server.

→ If the container is crashed, the same container is recreated automatically, this is called self healing

→ These self healing is automatically done by docker swarm

Scenario!

Failure scenarios of Workers

(*) Create httpd with 6 replicas and delete one replica running on the manager; check if all 6 replicas are still running.

(*) Drain worker 1 from the docker swarm and check if all 6 replicas are running on manager and worker 2, make worker 1 rejoin the swarm

(*) Make worker 2 leave the swarm and check if all the 6 replicas are running on manager and worker 1

→ Create httpd with 6 replicas

```
docker service create --name webserver
-p 9090:80 --replicas 6 httpd
```

→ Check the replicas running on manager

```
docker service ps webserver | grep manager
```

→ Check the container id

```
docker container ls
```

→ Delete a replica

```
docker rm -f container-id-from step-above
```

→ Check if all 6 replicas are running

```
docker service ps webserver
```

→ Now drain worker1 from the swarm

```
docker node update --availability drain worker1
```

→ Now check if all 6 replicas are running

```
docker service ps webserver
```

→ Make worker1 rejoin the swarm

```
docker node update --availability active
worker1
```

→ Now make worker2 leave the swarm
Connect to worker2 using git bash

```
docker swarm leave
```

→ Now check if all the 6 replicas are still running.

```
docker service ps webserver
```

Failure scenarios of managers

→ If a worker instance crashes all the replicas running on that worker will be moved to the manager or the other workers.

→ If the manager itself crashes the swarm becomes headless ie we can't perform containers orchestration activities in this swarm cluster.

→ To avoid this we should maintain multiple managers., Manager nodes have the status as leader or Reachable

→ If one manager node goes down other manager becomes the leader, Quorum is responsible for doing this activity and it uses a RAFT algorithm for handling the failovers of managers.

→ Quorum also is responsible for maintaining the min no. of managers.

→ Min count of manager required for docker swarm should be always more than half of the total count of managers.

Total manager count = Min Manager Required + Fault tolerance

1	1	0
2	2	0
3	2	1
4	3	1

5	3	2
6	4	2
7	4	3
8	5	3
9	5	4
10	6	4

This is taken as the $\frac{\text{no. of managers}}{2} = x$

Min managers required = next number of x

Fault tolerance = total mangers − min manager required

eg:- We have total managers are 9

min manager required $= \frac{9}{2} = 4.5$

the next no. of 4.5 is 5

Fault tolerance : $9 - 5 = 4$

Docker stack :

docker compose + docker swarm = docker stack

docker compose + kubernetes = Kompose

Docker compose when implemented at the level of docker swarm it is called docker stack.

→ Using docker stack we can create a micro service architecture at the level of production servers

→ To create a stack from a compose file

```
docker stack deploy -c compose_filename
  stack_name
```

→ To see the list of stacks created

```
docker stack ls
```

→ To see on which nodes the stack service are running

```
docker stack ps stack_name
```

→ To delete a stack

```
docker stack rm stack_name
```

(*) Create a docker stackfile to start 3 replicas of wordpress and one replica of mysql

vim stack1.yml

```
---
version: '3.8'
services:
    db:
      image: mysql:5
      environment:
        MYSQL_ROOT_PASSWORD: sudarshan
    wordpress:
      image: wordpress
      ports:
        - 8181:80
      deploy:
        replicas: 3
...
```

→ To start the stack file

docker stack deploy -c stack1.yml mywordpress

→ To see the services running

```
docker service ls
```

→ To check where the services are running

```
docker stack ps mywordpress
```

→ To delete the stack

```
docker stack rm mywordpress
```

Scenario

* Create a stackfile to setup CI-CD architecture where jenkins container is linked with tomcats for qa and prod environments.

The jenkins container should only run on manager, tomcat of qa should run on worker1 and tomcat of prod should run on worker2

vim stack2.yml

```
---
version: '3.8'
services:
    myjenkins:
        image: jenkins/jenkins
        ports:
          - 8282:8080
        deploy:
          replicas: 2
          placement:
            constraints:
              - node.hostname == Manager
    qaserver:
        image: tomcat
        ports:
          - 8383:8080
        deploy:
          replicas: 3
          placement:
            constraints:
              - node.hostname == Worker1

    prodserver:
        image: tomcat
```

```
ports:
  - 8484:8080
deploy:
  replicas: 4
  placement:
    constraints:
      - node.hostname == Worker2
...
```

→ To start the services

```
docker stack deploy -c stack2.yml CD-CD
```

→ To see the stack

```
docker stack ls
```

Overlay Networking:

→ This is the default n/w used by swarm and this n/w perform n/w load balancing.

i,e even if a service is running on a specific worker we can access it from other slave.

→ The name of the overlay n/w is ingres

→ The containers which are created on this n/w are communicated with each other.

→ So these containers don't want to communicate with each other, so we create customised networks.

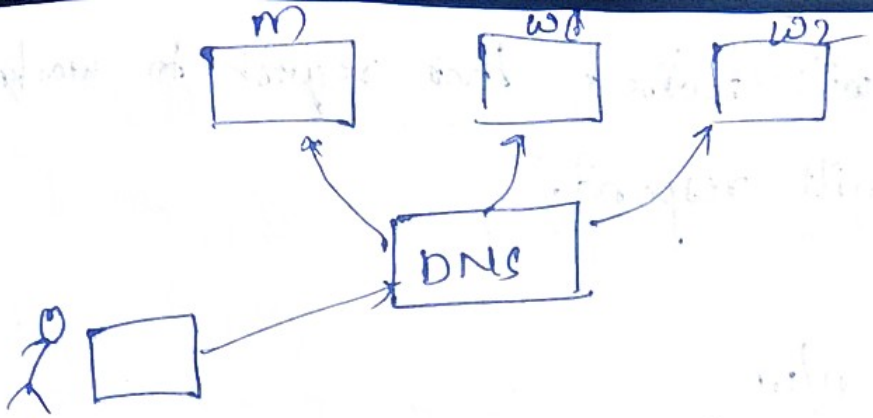
Scenario: Load balancing in n/w

(*) start nginx with 4 replicas and check if we can access it from browser from manager and all workers

```
docker service create --name mynginx
-p 8080:80 --replicas 4 nginx
```

To see all these containers

```
docker service ps myngnix
```



→ I have created 4 nginx services, and those are distributed on manager and workers

→ Now I scaled downed to 1; then only one nginx container is running, let's say on worker 1. On other hand there are no nginx containers on manager and worker 2.

→ Now even if you give ip address of manager and worker 2, the nginx will respond.

Conclusion:-

→ It is not that my application should run on all the servers to respond.

→ If the request goes to worker 2, but my application is on worker 1, then

worker2 will redirect that request to worker1 and it will respond.

Overlay n/w

Scenario:

* Create 2 overlay n/w's intelliqit1 and intelliqit2.

Create http2 with 5 replicas on intelliqit1 n/w.

Create tomcat with 5 replicas on default overlay "ingress" n/w. and later perform rolling n/w update to intelliqit2 n/w.

① Create 2 overlay n/w's

```
docker network create --driver overlay
intelliqit1
```

```
docker network create --driver overlay
intelliqit2
```

② check if 2 overlay n/w's are created

```
docker network ls
```

③ Create httpd with 5 replicas on intelliqit !

```
docker service create --name webservice
-p 8888:80 --replicas 5 -- network intelliqit
httpd
```

④ To check if httpd is running on intelliqit n/w.

```
docker service inspect webservice
```

This command will generate the o/p in JSON format

To see the above o/p in normal text format

```
docker service inspect webservice --pretty
```

⑤ Now create 5 replicas on the default ingress n/w

```
docker service create --name appserver
-p 9999:8080 --replicas 5 tomcat
```

(6) Perform a rolling n/w update from ingres to intelliqit2 n/w.

```
docker service update --network-add
intelliqit2 appserver
```

(7) Now check if tomcat is running on intelliqit2

```
docker service inspect appserver
```

→ To remove from intelliqit2 network

```
docker service update --network-rm
intelliqit2 appserver
```

Setting upperlimit of resources for the container

→ Docker automatically allocates the resources for the container to use.

→ If this is the case, other container might not be getting the minimum resources.

→ To avoid this, we can set up upperlimit of resources like cpu, memory to a container.

Syntax

```
deploy:
  resources:
    limits:
      cpus: "0.1"      → in percentage
      memory: "200M"   → megabytes
```

Scenario:

① Create a stack file to setup the selenium hub and nodes architecture but also specify a upper limit on the hardware

vim stack3.yml

```
---
version: '3.8'

services:
  hub:
    image: selenium/hub
    ports:
      - 4444:4444
    deploy:
      replicas: 2
      resources:
        limits:
          cpus: "0.1"
          memory: "200M"

  firefox:
    image: selenium/node-firefox-debug
    ports:
      - 5901:5900
    deploy:
      replicas: 2
      resources:
        Limits:
          cpus: "0.1"
          memory: "300m"

  chrome:
    image: selenium/node-chrome-debug
```

```
ports :
  - 5902:5900
deploy :
  replicas : 2
  resources :
    Limits :
      cpus : "0.1"
      memory : "300M"
```

Number of servers must have same no. of exactly one replica on each

→ let's say if I have 5 servers, I must have 5 replicas, exactly one replica on each server. this must be allocated by the docker automatically.

Eg:-

I have one manager server and one worker server.

→ when I create a nginx service it must be replicated one on manager and

one on worker1.

```
docker service create --name mynginx -p
8080:80 --mode=global nginx
```

`--mode=global` → it creates no. of replicas = no. of servers.

→ If the container is deleted we must have the data should be backedup, for this we have volumes in docker.

→ But in docker swarm we don't have volumes.

→ Instead we have secrets

Docker secrets:-

→ This is a feature of docker swarm using which we can pass secret data to the services running in swarm cluster.

→ These secrets are created on the host

machine and they will be available from all the replicas in the swarm cluster.

→ To create a docker secret

```
echo "Hello sudarshan" | docker secret create
mysecret -
```

→ Now create redis database with 5 replicas and mount mysecret

```
docker service create --name myredis
--replicas 5 --secret mysecret redis
```

→ Now capture the one of the replica container id.

```
docker container ls
```

→ Check if the secret data is available

```
docker exec -it container_id bash
```

```
cd run/secrets/mysecrets
```

Scenario:

⑧ Create docker file for maven project from intelliJ

→ First clone it from the hub

```
git clone paste the url
```

→ Now go inside the maven folder and ~~execute~~ build that into war file.

```
cd maven
```

```
mvn package
```

→ Now go inside target folder which is present maven/webapp/target.

there you find webapp.war file, copy that file /root

```
CP -R * ../../../.
```

⑨ ~~Now~~ create a ~~docker~~ file.

→ Copy that only webapp.war file into /root

→ In tomcat container you find a file webapp, take that path and paste in dockerfile.

eg:- path is `/usr/local/tomcat/webapps`

→ You also find webapps folder and webapps.dist

change their names as

webapps → webappsbkp/any name eg: sudarshan/

webapps.dist → webapps webapp/

vim dockerfile

```
FROM tomcat:9
MAINTAINER sudarshan
RUN mv webapps webappsbkp
RUN mv webapps.dist webapps
COPY *.war /usr/local/tomcat/webapps/testapp.war
```

→ Now build the image

→ Create container → Now access from the browser → `ip address/port no/testapp`

```
Vim dockerfile

FROM tomcat:9
MAINTAINER sudarshan.
COPY *.war /usr/local/tomcat/webapps/testapp.war
```

→ build image → create container then
The access from the browser as

ip_address/port.no/testapp