Container Orchestration

Prepared by:
Ms. Avita Katal
Assistant Professor (SG)
School of Computer Science
UPES, Dehradun

Objectives

- Understanding the purpose and benefits of container orchestration.
- Learning about the features and different container orchestration tools available.

Container Orchestration

- ❑ Container orchestration is a process that **automates the container lifecycle of container-based (or “containerized”) applications**. This includes **deployment, management, scaling, networking, and availability.**

- ❑ Container orchestration is a necessity in large, dynamic environments, since it:
  - **Streamlines complexity**.
  - **Enables hands-off deployment and scaling.**
  - **Increases speed, agility, and efficiency.**
  - **Seamlessly integrates into CI/CD workflows and DevOps practices.**
  - **And allows development teams to use resources more efficiently.**

- ❑ Container orchestration can be implemented **on-premises, and on public, private, or multi-cloud environments.**

- ❑ It is often a critical part of an organization’s **security, orchestration, automation, and response** requirements, also known as “**SOAR**” requirements.

Container Orchestration Features

These features include:

- Defining which **container images make up the application**, and **where they are located** (in what registry).
- Improving **provisioning and deployment of containers** for a more automated, unified, and smooth process.
- **Securing network connections between containers**.
- Ensuring **availability and performance by relocating the containers to another host if an outage or shortage of system resources occurs**.

Container Orchestration Features

- **Scaling containers** to meet demand, and **load balance requests.**

- Handling **resource allocation and scheduling of containers** to the underlying infrastructure.

- Performing **rolling updates and roll backs and conducting health checks to ensure applications are running**, or performing the necessary actions when checks fail.

How Container Orchestration Works

Configuration File
YAML* and *JSON FILES configure each container so it can
- Find resources
- Establish a network
- Store logs

Deployment Scheduling
- ***automatically schedules the deployment*** of a new container to a cluster.
- ***finds the right host*** based on predefined settings or restrictions.

Manages container Lifecycle
Configuration file specs inform container decisions
- System parameters
- File parameters

Scaling and Productivity
Automation is used to:
- Maintain productivity
- Support scaling

Container Orchestration Tools

UPES
UNIVERSITY OF TOMORROW

Marathon

- Marathon is a **framework** for **Apache Mesos.**
- An **open-source cluster manager** that was developed by the **University of California at Berkeley.**
- Scales container infrastructure by automating the bulk of management and monitoring tasks.

Nomad

•**HashiCorp's** Nomad is a **free and open-source** cluster management and scheduling tool

• Supports **Docker and other standalone, virtualized, or containerized applications** on all major operating systems across all infrastructure, whether on-premises or in the cloud.

Docker Swarm

• **Open-source orchestration tool.**

• Automates the deployment of containerized applications.

• Designed **specifically to work with Docker Engine** and other Docker tools making it a popular choice for teams.

Kubernetes

•**Developed by Google and maintained by the Cloud Native Computing Foundation (CNCF)**, this **open-source platform Kubernetes** is the de facto standard for container orchestration.

• **Robust feature set, widely supported**

Container Orchestration Benefits

Container orchestration helps to meet business goals and increase profitability by using automation
The benefits of container orchestration for developers and administrators include:

- **Increased productivity:** Removing the burden of individually installing and managing each container, which, in turn reduces errors and frees development teams to focus on application improvement.
- **Faster deployments:** Iteratively releasing new features and capabilities faster and deploying containers and containerized applications rapidly.
- **Reduced costs:** Being cost-effective since containers have lower overhead and use fewer resources than virtual machines or traditional servers.
- **Stronger security:** Sharing resources and isolating application processes, improving the container's overall security.
- **Easier scalability**: Scaling applications using a single command.
- **Faster error recovery**: Maintaining high availability by detecting and resolving issues like infrastructure failures automatically.

Docker Swarm

Prepared by:
Ms. Avita Katal
Assistant Professor (SG)
School of Computer Science
UPES, Dehradun

Objectives

1. Understand Docker Swarm, its features and use cases.

2. Learning different commands to use Docker Swarm

What is Swarm?

- The **cluster management and orchestration features** embedded in the Docker Engine are built using **swarmkit.**
- Swarmkit is a separate project which **implements Docker's orchestration layer** and is used directly within Docker.
- A swarm consists of **multiple Docker hosts** which run in swarm mode and act as managers (to manage membership and delegation) and workers (which run swarm services).
- A given Docker host can be a **manager, a worker, or perform both roles.**
- When you create a service, you define its **optimal state** (**number of replicas, network and storage resources available to it, ports the service exposes to the outside world, and more**).
- **Docker works to maintain that desired state**. For instance, if a worker node becomes unavailable, Docker schedules that node's tasks on other nodes. A task is a running container which is part of a swarm service and is managed by a swarm manager, as opposed to a standalone container.

What is Swarm?

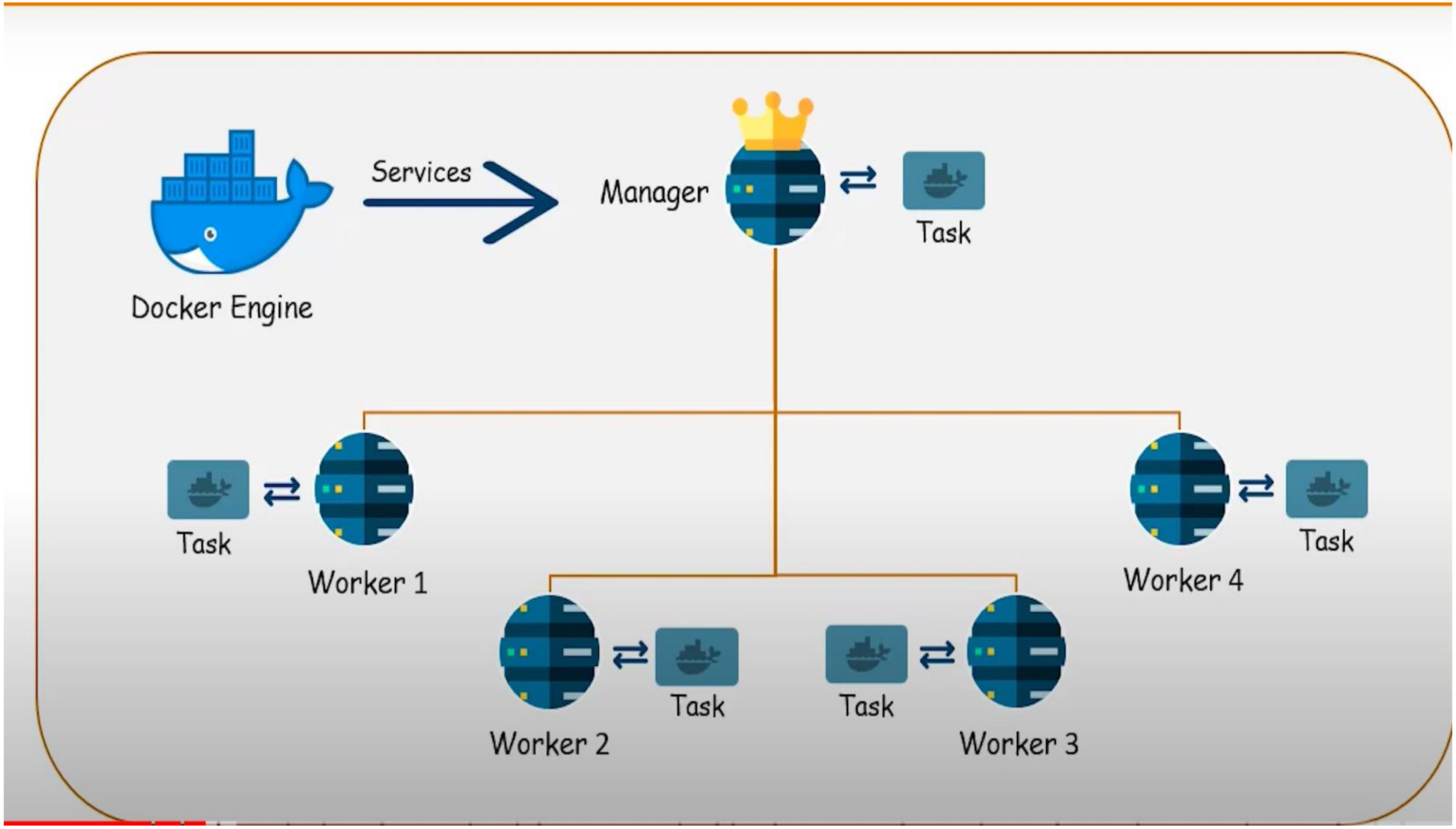
- One of the key advantages of swarm services over standalone containers is that you can modify a **service's configuration, including the networks and volumes it is connected to, without the need to manually restart the service.**

- When Docker is running in swarm mode, you can **still run standalone containers on any of the Docker hosts participating in the swarm, as well as swarm services**.

- A key difference between standalone containers and swarm services is that only **swarm managers can manage a swarm, while standalone containers can be started on any daemon**. Docker daemons can participate in a swarm as managers, workers, or both.

- In the same way that you can use Docker Compose to define and run containers, you can define and run Swarm service stacks.

Docker Swarm Architecture



Swarm Mode Concepts

Nodes:

- **A node is an instance of the Docker engine (Docker node) participating in the swarm.** There can be one or more nodes running on a single physical computer or cloud server, but **production swarm deployments typically include Docker nodes distributed across multiple physical and cloud machines**.

i) Manager Node

- To deploy your application to a swarm, you submit a **service definition** to a **manager node**. The manager node dispatches units of work called **tasks to worker nodes.**

- Manager nodes also perform the **orchestration and cluster management functions** required to maintain the **desired state** of the swarm. **Manager nodes elect a single leader to conduct orchestration tasks.**

Swarm Mode Concepts

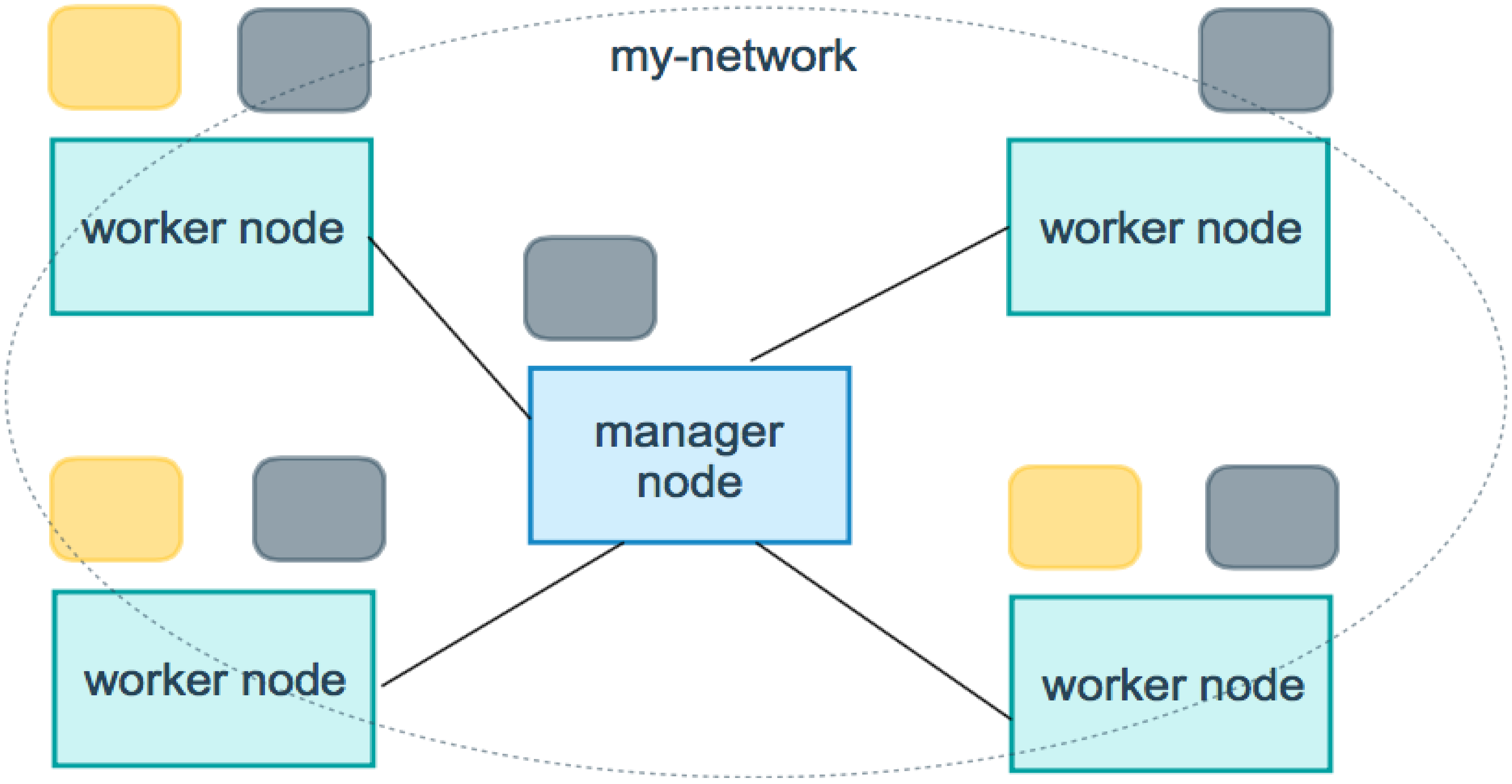
ii) Worker Nodes

- Worker nodes **receive and execute tasks dispatched** from manager nodes.
- By default **manager nodes also run services as worker nodes, but you can configure them to run manager tasks exclusively and be manager-only nodes**.
- An **agent runs on each worker node and reports on the tasks assigned to it**.
- The worker node **notifies the manager node of the current state** of its assigned tasks so that the manager can maintain the desired state of each worker.

Swarm Mode Concepts

Services:

- A service **is the definition of the tasks to execute on the manager or worker nodes**.
- It is the **central structure** of the swarm system and the **primary root of user interaction** with the swarm.
- When you **create a service, you specify which container image to use and which commands to execute inside running containers.**
    - In the **replicated services model**, the swarm manager distributes a specific number of replica tasks among the nodes based upon the scale you set in the desired state.
    - For **global services**, the swarm runs one task for the service on every available node in the cluster.



🟨 replicated service with 3 replicas

⬛ global service with replicas on every node

Swarm Mode Concepts

Load balancing

- The swarm manager uses **ingress load balancing** to expose the services you want to make available externally to the swarm.

- The swarm manager can **automatically assign the service a PublishedPort** or you can **configure a PublishedPort for the service.**

- You can specify any unused port. If you do not specify a port, the swarm manager assigns the service a port in the **30000-32767 range**.

- External components, such as **cloud load balancers, can access the service on the PublishedPort** of any node in the cluster whether or not the node is currently running the task for the service. All nodes in the swarm route ingress connections to a running task instance.

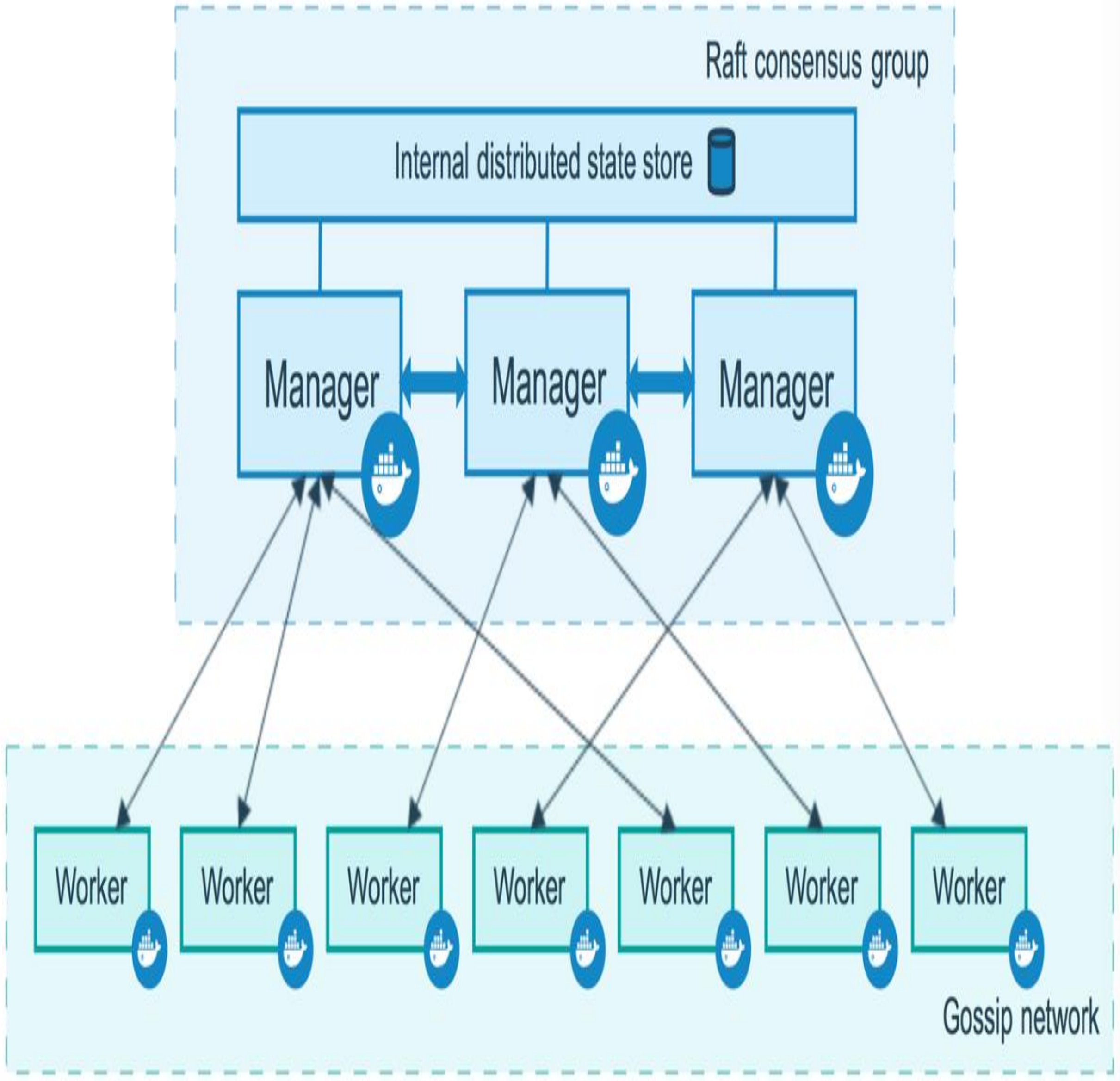
- Swarm mode has an **internal DNS component** that automatically assigns each service in the swarm a DNS entry.

- The swarm manager uses **internal load balancing** to distribute requests among services within the cluster based upon the DNS name of the service.

Swarm Mode Feature Highlights

1. Cluster management integrated with Docker Engine
2. Decentralized design
3. Declarative service model
4. Scaling
5. Desired state reconciliation
6. Multi-host networking
7. Service discovery
8. Load balancing
9. Secure by default
10. Rolling updates

Swarm Mode: Node Working



There are two types of nodes: **managers and workers.**

Manager nodes handle **cluster management tasks:**

- *maintaining cluster state*
- *scheduling services*
- *serving swarm mode HTTP API endpoints*

Using a **Raft implementation**, the managers maintain a consistent internal state of the entire swarm and all the services running on it.

For testing purposes it is OK to run a swarm with a **single manager.**

If the manager in a **single-manager swarm fails, your services continue to run, but you need to create a new cluster to recover.**

Swarm Mode: Node Working

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 odd number **N of manager nodes** in the cluster tolerates the loss of at most **(N-1)/2 managers**. Docker recommends a maximum of seven manager nodes for a swarm.

Swarm Mode: Node Working

- Worker nodes are also **instances of Docker Engine whose sole purpose is to execute containers.**

- Worker nodes don't participate in the **Raft distributed state**, make scheduling decisions, or serve the swarm mode HTTP API.

- You can create a swarm of **one manager node**, but you **cannot have a worker node without at least one manager node**. By default, **all managers are also workers**. In a single manager node cluster, you can run commands like **docker service create** and the scheduler places all tasks on the local Engine.

- To prevent the scheduler from placing tasks on a manager node in a multi-node swarm, **set the availability for the manager node to Drain**.

- The scheduler gracefully **stops tasks on nodes in Drain mode and schedules the tasks on an Active node.** The scheduler does not assign new tasks to nodes with Drain availability.
    - **docker node update** command line reference to see how to change node availability.

Swarm Mode: Node Working

- You can **promote a worker node to be a manager** by running **docker node promote**. For example, you may want to promote a worker node when you take a manager node offline for maintenance.

- **Promote one or more nodes to manager in the swarm.**

```
$ docker node promote <node name>
```

- **Demote one or more nodes from manager in the swarm**

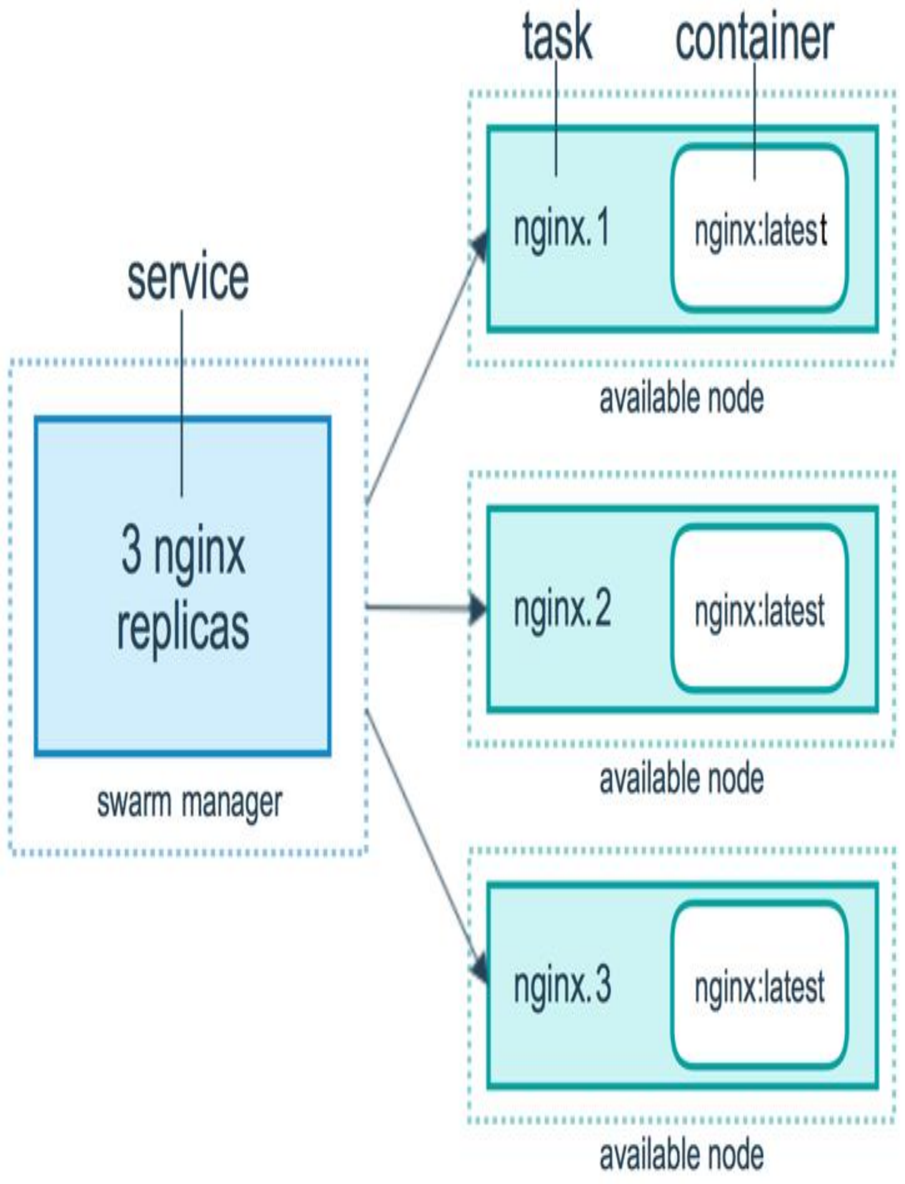
```
$ docker node demote <node name>
```

Swarm Mode: Service Working

To **deploy an application image** when Docker Engine is in swarm mode, **you create a service**. ***Frequently, a service is the image for a microservice within the context of some larger application.*** Examples of services might include an **HTTP server, a database, or any other type of executable program** that you wish to run in a distributed environment.

When you create a service, **you specify which container image to use and which commands to execute inside running containers.** You also define options for the service including:

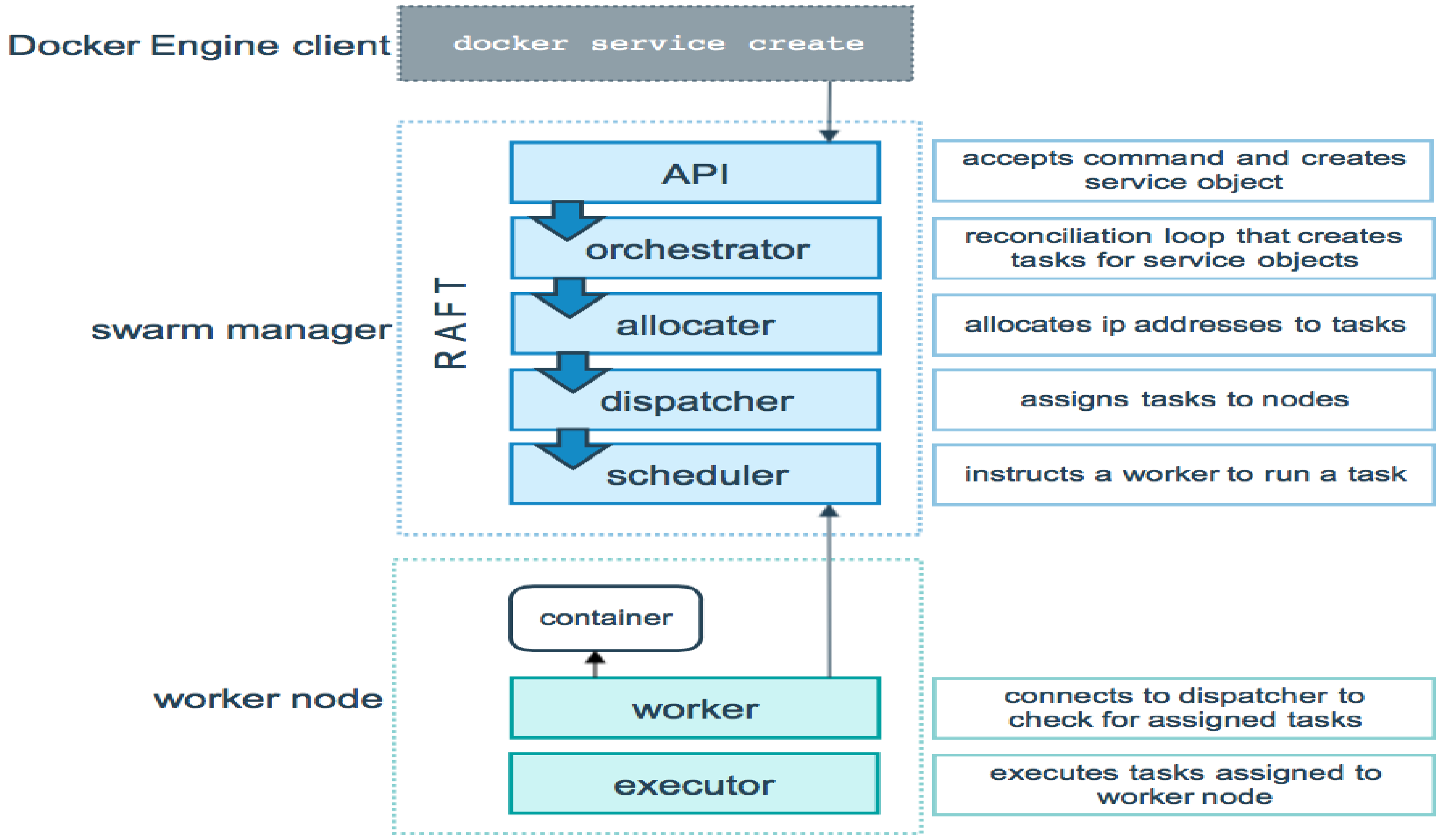
- *the port where the swarm makes the service available outside the swarm*
- an *overlay network for the service to connect to other services in the swarm*
- *CPU and memory limits and reservations*
- *a rolling update policy*
- *the number of replicas of the image to run in the swarm*



Swarm Mode: Service Working

- When you **deploy the service to the swarm, the swarm manager accepts your service definition as the desired state for the service**.
- Then it schedules the **service on nodes in the swarm as one or more replica tasks**. The tasks run independently of each other on nodes in the swarm.
- **A task is the atomic unit of scheduling within a swarm.**
- For example, imagine you want to load balance between three instances of an HTTP listener. The diagram below shows an HTTP listener service with three replicas. Each of the three instances of the listener is a task in the swarm
- A container is an isolated process. In the swarm mode model, each task invokes exactly one container. **A task is analogous to a "slot" where the scheduler places a container**. Once the container is live, the scheduler recognizes that the task is in a running state. If the container fails health checks or terminates, the task terminates.

Swarm Mode: Service Working-Tasks and Scheduling


Docker Engine client
docker service create
swarm manager
RAFT
API
orchestrator
allocater
dispatcher
scheduler
accepts command and creates service object
reconciliation loop that creates tasks for service objects
allocates ip addresses to tasks
assigns tasks to nodes
instructs a worker to run a task
worker node
container
worker
executor
connects to dispatcher to check for assigned tasks
executes tasks assigned to worker node

Swarm task states

Docker lets you create services, which can start tasks. **A service is a description of a desired state, and a task does the work.** Work is scheduled on swarm nodes in this sequence:

- Create a service by using docker service create.
- The request goes to a Docker manager node.
- The Docker manager node schedules the service to run on particular nodes.
- Each service can start multiple tasks.
- Each task has a life cycle, with states like NEW, PENDING, and COMPLETE.

Tasks are execution units that run once to completion. When a task stops, it isn't executed again, but a new task may take its place.

Tasks advance through a number of states until they complete or fail. Tasks are initialized in the NEW state. The task progresses forward through a number of states, and its state doesn't go backward. For example, a task never goes from COMPLETE to RUNNING.

Swarm task states

Tasks go through the states in the following order:

Task state	**Description**
NEW	The task was initialized.
PENDING	Resources for the task were allocated.
ASSIGNED	Docker assigned the task to nodes.
ACCEPTED	The task was accepted by a worker node. If a worker node rejects the task, the state changes to REJECTED.
READY	The worker node is ready to start the task
PREPARING	Docker is preparing the task.
STARTING	Docker is starting the task.
RUNNING	The task is executing.
COMPLETE	The task exited without an error code.
FAILED	The task exited with an error code.
SHUTDOWN	Docker requested the task to shut down.
REJECTED	The worker node rejected the task.
ORPHANED	The node was down for too long.
REMOVE	The task is not terminal but the associated service was removed or scaled down.

View task state

Run docker service ps <service-name> to get the state of a task. The CURRENT STATE field shows the task's state and how long it's been there.

docker service ps webserver

```
ID                  NAME            IMAGE   NODE       DESIRED STATE  CURRENT STATE            ERROR                            PORTS
owsz0yp6z375   webserver.1      nginx   UbuntuVM   Running        Running 44 seconds ago
j91iahr8s74p    \_ webserver.1   nginx   UbuntuVM   Shutdown       Failed 50 seconds ago    "No such container: webserver.â
7dyaszg13mw2    \_ webserver.1   nginx   UbuntuVM   Shutdown       Failed 5 hours ago       "No such container: webserver.â
```

Getting started with swarm mode

The work guides you through the following activities:

- *initializing a cluster of Docker Engines in swarm mode*
- *adding nodes to the swarm*
- *deploying application services to the swarm*
- *managing the swarm once you have everything running*

To run this work, you need the following:

- *three Linux hosts which can communicate over a network, with Docker installed*
- *the IP address of the manager machine*
- *open ports between the hosts*

This work requires three Linux hosts which have Docker installed and can communicate over a network. These can be physical machines, virtual machines, Amazon EC2 instances, or hosted in some other way.

Install Docker Engine on Linux machines

If you are using Linux based physical computers or cloud-provided computers as hosts, simply follow the Linux install instructions for your platform. Spin up the three machines, and you are ready. You can test both single-node and multi-node swarm scenarios on Linux machines.

Make sure that Swarm is enabled on your Docker Desktop by typing ***docker system info,*** and looking for a message Swarm: active (you might have to scroll up a little).

If Swarm isn't running, simply type ***docker swarm init*** in a shell prompt to set it up.

Commands-Create a Docker Swarm

1. Run the following command to create a new swarm:

docker swarm init --advertise-addr <MANAGER-IP>

In the tutorial, the following command creates a swarm on the manager1 machine

```
docker swarm init --advertise-addr 192.168.99.100
Swarm initialized: current node (dxn1zf6l61qsb1josjja83ngz) is now a manager.

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

    docker swarm join \
    --token SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-
8vxv8rssmk743ojnwacrr2e7c \
    192.168.99.100:2377
To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

Commands-Create a Docker Swarm

2. Run ***docker info*** to view the current state of the swarm:

```
docker info
Containers: 2
Running: 0
Paused: 0
Stopped: 2
  ...snip...
Swarm: active
  NodeID: dxn1zf6l61qsb1josjja83ngz
  Is Manager: true
  Managers: 1
  Nodes: 1
  ...snip...
```

Commands-Create a Docker Swarm

3. Run the ***docker node ls*** command to view information about nodes:
docker node ls

```
ID                           HOSTNAME   STATUS  AVAILABILITY  MANAGER STATUS
dxn1zf6l61qsb1josjja83ngz *  manager1   Ready   Active        Leader
```

The * next to the node ID indicates that you're currently connected on this node.

Docker Engine swarm mode automatically names the node with the machine host name.

Commands- Adding Worker Node1 to the Swarm

4. Once you've created a swarm with a manager node, you're ready to add worker nodes.

4.1. Open a terminal and ssh into the machine where you want to run a worker node. This tutorial uses the name worker1.

4.2. Run the command produced by the docker swarm init output from the Create a swarm tutorial step to create a worker node joined to the existing swarm:

```
docker swarm join \
  --token  SWMTKN-1-
49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-
8vxv8rssmk743ojnwacrr2e7c \
  192.168.99.100:2377
```

Commands- Adding Worker Node1 to the Swarm

This node joined a swarm as a worker.

- If you don't have the command available, you can run the following command on a manager node to retrieve the join command for a worker:

docker swarm join-token worker

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

```
    docker swarm join \
    --token SWMTKN-1-
49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-
8vxv8rssmk743ojnwacrr2e7c \
    192.168.99.100:2377
```

Commands-Adding Worker Node 2 to the Swarm

4.3. Open a terminal and ssh into the machine where you want to run a second worker node. This tutorial uses the name worker2.

4.4. Run the command produced by the docker swarm init output from the Create a swarm tutorial step to create a second worker node joined to the existing swarm:

```
docker swarm join \
  --token SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-8vxv8rssmk743ojnwacrr2e7c \
  192.168.99.100:2377
```

This node joined a swarm as a worker.

Commands-Adding Worker Node 2 to the Swarm

4.5 Open a terminal and ssh into the machine where the manager node runs and run the docker node ls command to see the worker nodes:

docker node ls

```
            ID                      HOSTNAME     STATUS          AVAILABILITY  MANAGER STATUS
03g1y59jwfg7cf99w4lt0f662    worker2         Ready           Active
9j68exjopxe7wfl6yuxml7a7j    worker1         Ready           Active
dxn1zf6l61qsb1josjja83ngz *  manager1        Ready           Active              Leader
```

The MANAGER column identifies the manager nodes in the swarm. The empty status in this column for worker1 and worker2 identifies them as worker nodes.

Swarm management commands like **docker node ls only work on manager nodes.**

Commands-Deploying a service to the Swarm

5. After you create a swarm, you can deploy a service to the swarm.

5.1. Open a terminal and ssh into the machine where you run your manager node. For example, the tutorial uses a machine named manager1.

Run the following command:

```
docker service create --replicas 1 --name helloworld alpine ping docker.com

9uk4639qpg7npwf3fn2aasksr
```

- The *docker service create* command creates the service.
- The --*name* flag names the service helloworld.
- The --*replicas* flag specifies the desired state of 1 running instance.
- The arguments *alpine ping docker.com* define the service as an Alpine Linux container that executes the command ping docker.com.

Commands-Deploying a service to the Swarm

5.2. Run docker service ls to see the list of running services:

docker service ls

```
ID           NAME        SCALE  IMAGE   COMMAND
9uk4639qpg7n  helloworld  1/1    alpine  ping docker.com
```

Commands-Inspecting a service

6. When you have deployed a service to your swarm, you can use the Docker CLI to see details about the service running in the swarm.

6.1. If you haven't already, open a terminal and ssh into the machine where you run your manager node. For example, the tutorial uses a machine named manager1.

6.2. Run **docker service inspect --pretty <SERVICE-ID>** to display the details about a service in an easily readable format.

To see the details on the helloworld service:

```
[manager1]$ docker service inspect --pretty helloworld
ID:                9uk4639qpg7npwf3fn2aasksr
Name:              helloworld
Service Mode:      REPLICATED
 Replicas:         1
Placement:
UpdateConfig:
 Parallelism:      1
ContainerSpec:
 Image:            alpine
 Args:      ping docker.com
Resources:
Endpoint Mode:  vip
```

Tip: To return the service details in json format, run the same command without the --pretty flag.

Commands-Inspecting a service

6.3. Run **docker service ps <SERVICE-ID>** to see **which nodes are running** the service:

[manager1]$ **docker service ps helloworld**

```
NAME                                 IMAGE      NODE      DESIRED STATE    CURRENT STATE     ERROR        PORTS
helloworld.1.8p1vev3fq5zm0mi8g0as41w35   alpine     worker2   Running              Running           3 minutes
```

In this case, the one instance of the helloworld service is running on the worker2 node. You may see the service running on your manager node. By default, manager nodes in a swarm can execute tasks just like worker nodes.

Swarm also shows you the DESIRED STATE and CURRENT STATE of the service task so you can see if tasks are running according to the service definition.

Commands-Inspecting a service

6.4. Run **docker ps** on the **node where the task is running to see details about the container for the task.**

Tip: If helloworld is running on a node other than your manager node, you must ssh to that node.

```
[worker2]$ docker ps
```

```
CONTAINER ID     IMAGE           COMMAND           CREATED              STATUS          PORTS          NAMES
e609dde94e47     alpine:latest   "ping docker.com"   3 minutes ago      Up 3 minutes                   helloworld.1.8p1vev3fq5zm0mi8g0as41w35
```

Commands-Scaling a service

Once you have deployed a service to a swarm, you are ready to use the Docker CLI to scale the number of containers in the service. Containers running in a service are called "tasks."

7. If you haven't already, open a terminal and ssh into the machine where you run your manager node. For example, the tutorial uses a machine named manager1.

7. 1. Run the following command to change the desired state of the service running in the swarm:

```
docker service scale <SERVICE-ID>=<NUMBER-OF-TASKS>
```

For example:

```
docker service scale helloworld=5
```

```
helloworld scaled to 5
```

Commands-Scaling a service

7.2. Run docker service ps <SERVICE-ID> to see the updated task list:
docker service ps helloworld

```
NAME                                  IMAGE    NODE      DESIRED STATE  CURRENT STATE
helloworld.1.8p1vev3fq5zm0mi8g0as41w35  alpine   worker2   Running        Running 7 minutes
helloworld.2.c7a7tcdq5s0uk3qr88mf8xco6  alpine   worker1   Running        Running 24 seconds
helloworld.3.6crl09vdcalvtfehfh69ogfb1  alpine   worker1   Running        Running 24 seconds
helloworld.4.auky6trawmdlcne8ad8phb0f1  alpine   manager1  Running        Running 24 seconds
helloworld.5.ba19kca06l18zujfwxyc5lkyn  alpine   worker2   Running        Running 24 seconds
```

You can see that swarm has created 4 new tasks to scale to a total of 5 running instances of Alpine Linux. The tasks are distributed between the three nodes of the swarm. One is running on manager1.

Commands-Deleting a service

- The remaining steps in the tutorial don't use the helloworld service, so now you can delete the service from the swarm.

8.1. If you haven't already, open a terminal and ssh into the machine where you run your manager node. For example, the tutorial uses a machine named manager1.

8.2. Run **docker service rm helloworld** to remove the helloworld service.

```
docker service rm helloworld

Helloworld
```

8.3. Run docker service inspect <SERVICE-ID> to verify that the swarm manager removed the service. The CLI returns a message that the service is not found:

```
docker service inspect helloworld

Status: Error: no such service: helloworld, Code: 1
```

Commands-Deleting a service

8.4. Even though **the service no longer exists, the task containers take a few seconds to clean up.** You can use **docker ps** on the nodes to verify when the tasks have been removed.

docker ps

```
CONTAINER ID       IMAGE              COMMAND                CREATED            STATUS            PORTS     NAMES
db1651f50347       alpine:latest      "ping docker.com"      44 minutes ago     Up 46 seconds
helloworld.5.9lkmos2beppihw95vdwxy1j3w
43bf6e532a92       alpine:latest      "ping docker.com"      44 minutes ago     Up 46 seconds
helloworld.3.a71i8rp6fua79ad43ycocl4t2
5a0fb65d8fa7       alpine:latest      "ping docker.com"      44 minutes ago     Up 45 seconds
helloworld.2.2jpgensh7d935qdc857pxulfr
afb0ba67076f       alpine:latest      "ping docker.com"      44 minutes ago     Up 46 seconds
helloworld.4.1c47o7tluz7drve4vkm2m5olx
688172d3bfaa       alpine:latest      "ping docker.com"      45 minutes ago     Up About a minute
helloworld.1.74nbhb3fhud8jfrhigd7s29we
```

Commands-Applying Rolling Updates

In a previous step of the tutorial, you scaled the number of instances of a service. In this part of the tutorial, you deploy a service based on the Redis 3.0.6 container tag. Then you upgrade the service to use the Redis 3.0.7 container image using rolling updates.

9.1. If you haven't already, open a terminal and ssh into the machine where you run your manager node. For example, the tutorial uses a machine named manager1.

9.2. Deploy your Redis tag to the swarm and configure the swarm with a 10 second update delay. Note that the following example shows an older Redis tag:

```
docker service create \
--replicas 3 \
--name redis \
--update-delay 10s \
redis:3.0.6
```

```
0u6a4s31ybk7yw2wyvtikmu50
```

Commands-Applying Rolling Updates

- You configure the rolling update policy at service deployment time.
- The **--update-delay flag** configures the time delay between updates to a service task or sets of tasks. You can describe the time T as a combination of the number of seconds Ts, minutes Tm, or hours Th. So **10m30s indicates a 10 minute 30 second delay**.
- By default the **scheduler updates 1 task at a time**. You can pass the **--update-parallelism flag** to configure the **maximum number of service tasks that the scheduler updates simultaneously.**

By default, when an update to an individual task returns a state of RUNNING, the scheduler schedules another task to update until all tasks are updated.
If at any time during an update a task returns FAILED, **the scheduler pauses the update**.

- You can control the behavior using the **--update-failure-action flag for docker service create or docker service update.**

Commands-Applying Rolling Updates

9.3. Inspect the redis service:

docker service inspect --pretty redis

```
ID:          0u6a4s31ybk7yw2wyvtikmu50
Name:          redis
Service Mode:   Replicated
 Replicas:      3
Placement:
 Strategy:    Spread
UpdateConfig:
 Parallelism:   1
 Delay:        10s
ContainerSpec:
 Image:        redis:3.0.6
Resources:
Endpoint Mode:  vip
```

Now you can update the container image for redis. The swarm manager applies the update to nodes according to the UpdateConfig policy:

```
docker service update --image redis:3.0.7 redis
redis
```

Commands-Applying Rolling Updates

Now you can update the container image for redis. The swarm manager applies the update to nodes according to the UpdateConfig policy:

```
$ docker service update --image redis:3.0.7 redis
```

```
redis
```

Commands-Applying Rolling Updates

The scheduler applies rolling updates as follows by default:

- Stop the first task.
- Schedule update for the stopped task.
- Start the container for the updated task.
- If the update to a task returns RUNNING, wait for the specified delay period then start the next task.
- If, at any time during the update, a task returns FAILED, pause the update.

Commands-Applying Rolling Updates

9.4. Run docker service inspect --pretty redis to see the new image in the desired state:

docker service inspect --pretty redis

```
ID:             0u6a4s31ybk7yw2wyvtikmu50
Name:           redis
Service Mode:   Replicated
 Replicas:      3
Placement:
 Strategy:      Spread
UpdateConfig:
 Parallelism:   1
 Delay:         10s
ContainerSpec:
 Image:         redis:3.0.7
Resources:
Endpoint Mode:  vip
```

Commands-Applying Rolling Updates

9.5. The output of service inspect shows if your update paused due to failure:

docker service inspect --pretty redis

```
ID:           0u6a4s31ybk7yw2wyvtikmu50
Name:         redis
...snip...
Update status:
 State:      paused
 Started:    11 seconds ago
 Message:    update paused due to failure or early termination of task 9p7ith557h8nd
...snip...
```

Commands-Applying Rolling Updates

9.6. To restart a paused update run docker service update <SERVICE-ID>. For example:f0ui9s0q951b

docker service update redis

To avoid repeating certain update failures, you may need to reconfigure the service by passing flags to docker service update.
Run docker service ps <SERVICE-ID> to watch the rolling update

docker service ps redis

```
NAME                              IMAGE        NODE       DESIRED STATE  CURRENT STATE           ERROR
redis.1.dos1zffgeofhagnve8w864fco      redis:3.0.7  worker1    Running        Running 37 seconds
 \_ redis.1.88rdo6pa52ki8oqx6dogf04fh  redis:3.0.6  worker2    Shutdown       Shutdown 56 seconds ago
redis.2.9l3i4j85517skba5o7tn5m8g0      redis:3.0.7  worker2    Running        Running About a minute
 \_ redis.2.66k185wilg8ele7ntu8f6nj6i  redis:3.0.6  worker1    Shutdown       Shutdown 2 minutes ago
redis.3.egiuiqpzrdbxks3wxgn8qib1g      redis:3.0.7  worker1    Running        Running 48 seconds
 \_ redis.3.ctzktfddb2tepkr45qcmqln04  redis:3.0.6  mmanager1  Shutdown       Shutdown 2 minutes ago
```

Before Swarm updates all of the tasks, you can see that some are running redis:3.0.6 while others are running redis:3.0.7. The output above shows the state once the rolling updates are done.

Commands-Drain a node on the swarm

- In earlier steps of the tutorial, all the nodes have been running with ACTIVE availability. The swarm manager can assign tasks to any ACTIVE node, so up to now all nodes have been available to receive tasks.
- Sometimes, such as planned maintenance times, you need to set a node to DRAIN availability. **DRAIN availability prevents a node from receiving new tasks from the swarm manager. It also means the manager stops tasks running on the node and launches replica tasks on a node with ACTIVE availability**.

10.1. If you haven't already, open a terminal and ssh into the machine where you run your manager node. For example, the tutorial uses a machine named manager1.

10.2. Verify that all your nodes are actively available.

```
$ docker node ls
docker node ls
```

```
ID                            HOSTNAME     STATUS        AVAILABILITY    MANAGER STATUS
1bcef6utixb0l0ca7gxuivsj0      worker2     Ready         Active
38ciaotwjuritcdtn9npbnkuz      worker1     Ready         Active
e216jshn25ckzbvmwlnh5jr3g *    manager1    Ready         Active              Leader
```

Commands-Drain a node on the swarm

10.3. Run docker node update --availability drain <NODE-ID> to drain a node that had a task assigned to it:

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

```
worker1
```

10.4. Inspect the node to check its availability:

```
$ docker node inspect --pretty worker1
```

```
ID:                 38ciaotwjuritcdtn9npbnkuz
Hostname:           worker1
Status:
 State:             Ready
 Availability:      Drain
...snip...
```

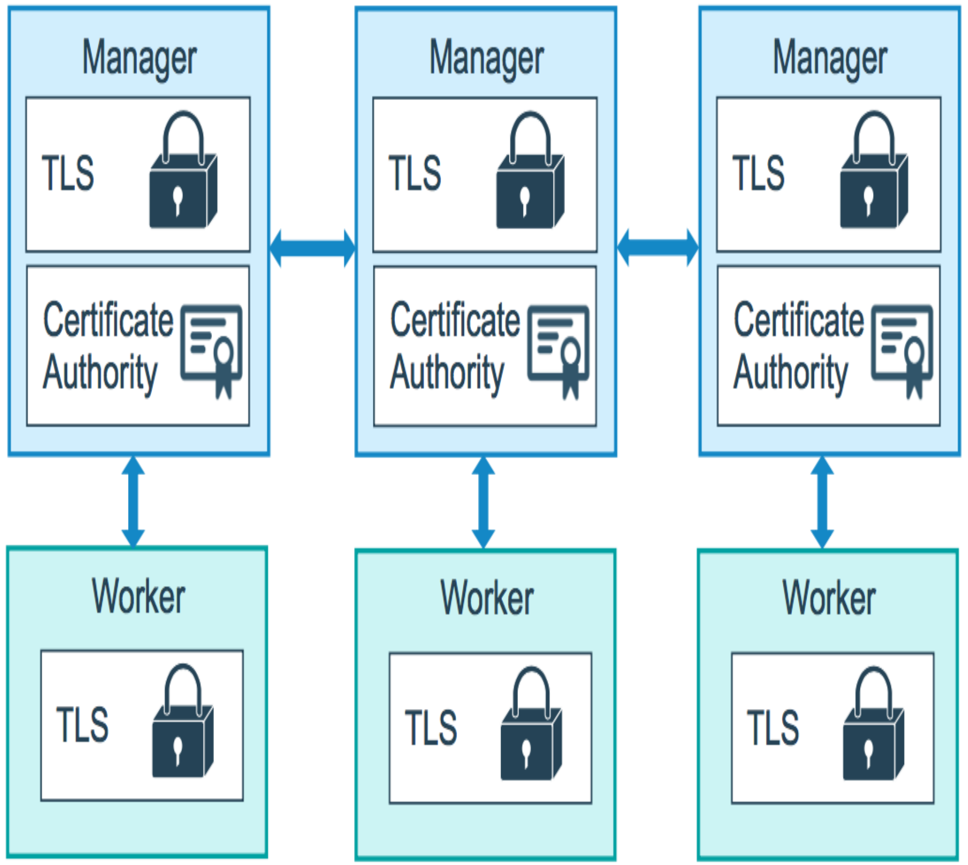
The drained node shows Drain for AVAILABILITY

Commands-Drain a node on the swarm

When you set the node back to Active availability, it can receive new tasks:

- during a service update to scale up
- during a rolling update
- when you set another node to Drain availability
- when a task fails on another active node

Commands-Managing swarm security with Public Key infrastructure (PKI)



- The swarm mode public key infrastructure (PKI) system built into Docker makes it simple to securely deploy a container orchestration system.
- The nodes in a swarm use mutual Transport Layer Security (TLS) to authenticate, authorize, and encrypt the communications with other nodes in the swarm.
- When you create a swarm by running docker swarm init, Docker designates itself as a manager node.
- By default, the manager node generates a new root Certificate Authority (CA) along with a key pair, which are used to secure communications with other nodes that join the swarm.
- If you prefer, you can specify your own externally-generated root CA, using the --external-ca flag of the docker swarm init command.

Commands-Managing swarm security with Public Key infrastructure (PKI)

- The manager node also generates two tokens to use when you join additional nodes to the swarm: one worker token and one manager token.
  - Each token includes the digest of the root CA's certificate and a randomly generated secret.
  - When a node joins the swarm, the joining node uses the digest to validate the root CA certificate from the remote manager. The remote manager uses the secret to ensure the joining node is an approved node.

- Each time a new node joins the swarm, the manager issues a certificate to the node.
  - The certificate contains a randomly generated node ID to identify the node under the certificate common name (CN) and the role under the organizational unit (OU).
  - The node ID serves as the cryptographically secure node identity for the lifetime of the node in the current swarm.

UPES
UNIVERSITY OF TOMORROW

Thank You