Systems of Systems (SoS)

M.Sc. In Critical Computing Systems Engineering

ISEP/IPP – 2021/22, 2nd semester

Containers & Docker

Pedro Santos

Outline

1. Introduction to Containers
2. Architecture & Components
3. How to setup a container
4. Multi-container Applications & Docker Networking
5. Container Orchestration

What is a Container?

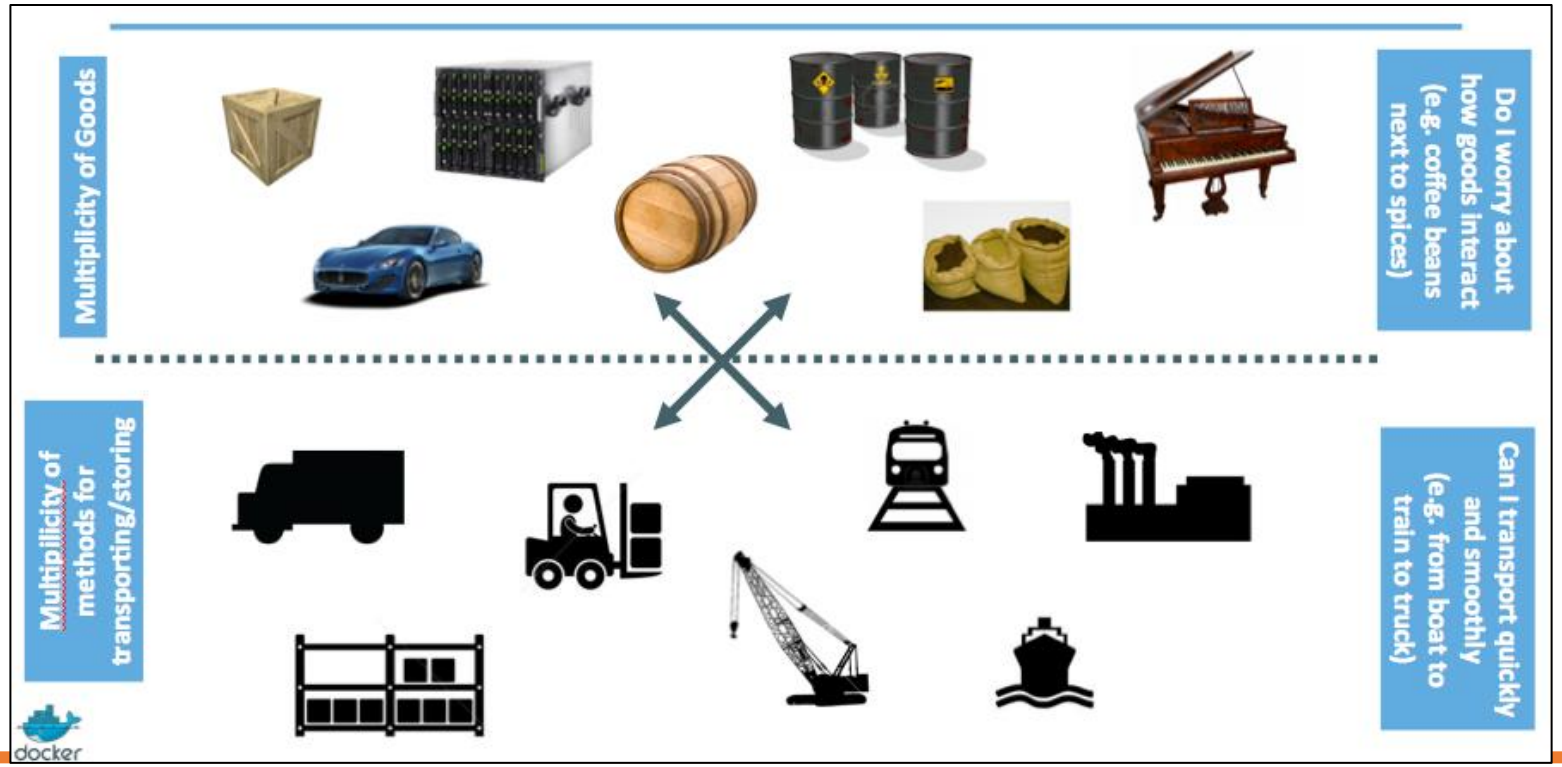
- Containerization is an approach to software development in which an application or service, its dependencies, and its configuration (abstracted as deployment manifest files) are packaged together as a container image.

- Simply put, a container is a sandboxed process on your machine that is isolated from all other processes on the host machine.
- That isolation leverages kernel namespaces and cgroups. These features that have been in Linux for a long time, but Docker has worked to make these capabilities approachable and easy to use.
- Containerized applications run on top of a container host that in turn runs on the OS (Linux or Windows), therefore have a significantly smaller footprint than virtual machine (VM) images.
- Each container can run a whole web application or a service.

- To summarize, a container:
    - is a runnable instance of an image. You can create, start, stop, move, or delete a container using the DockerAPI or CLI.
    - can be run on local machines, virtual machines or deployed to the cloud.
    - is portable (can be run on any OS)
    - is isolated from another containers and run their own software, binaries, and configurations.

Why “Container”?

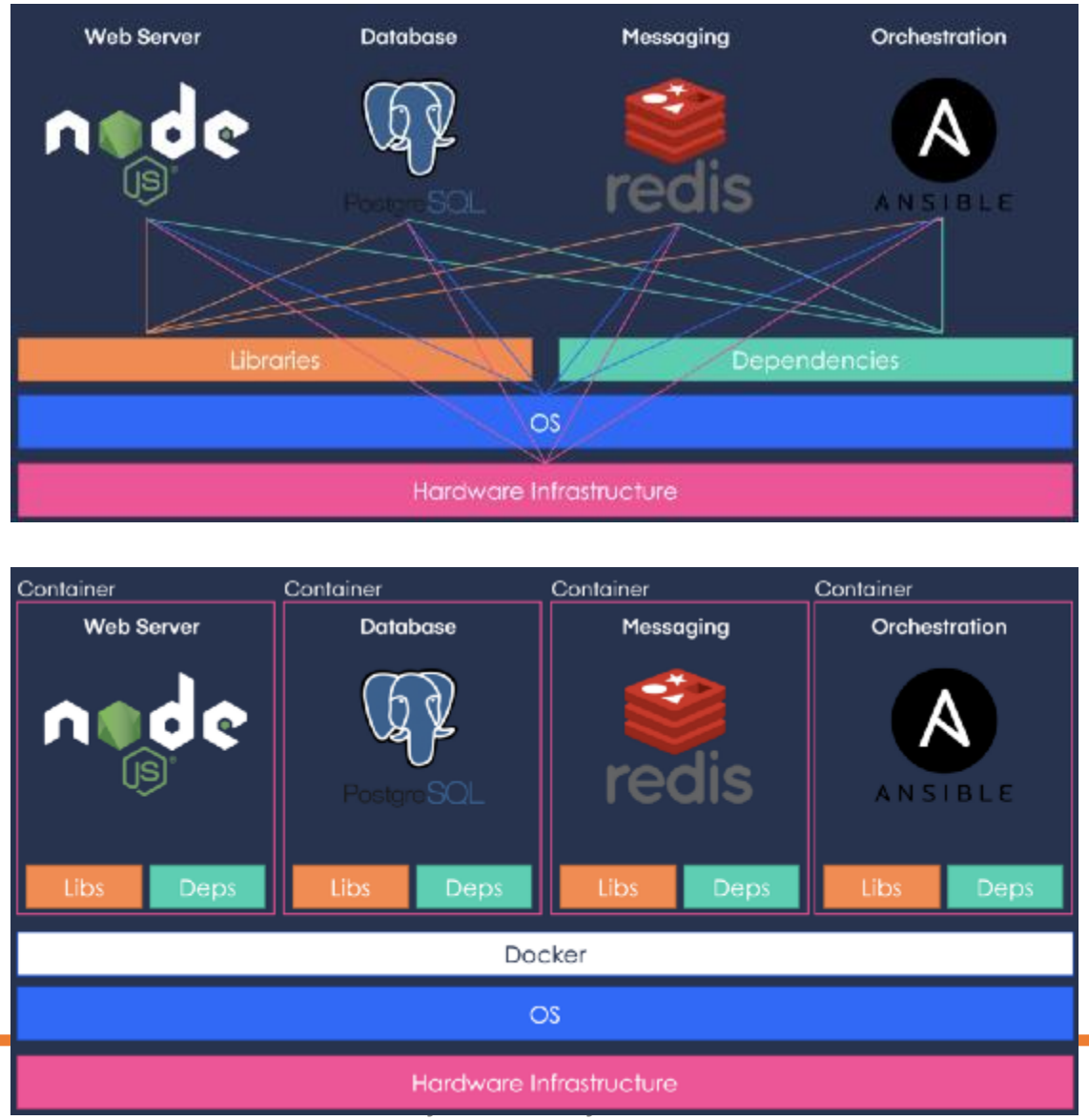
- Just as shipping containers allow goods to be transported by ship, train, or truck regardless of the cargo inside, software containers act as a standard unit of software deployment that can contain different code and dependencies.
- Containerizing software this way enables developers and IT professionals to deploy them across environments with little or no modification.



Why “Container”?

- Just as shipping containers allow goods to be transported by ship, train, or truck regardless of the cargo inside, software containers act as a standard unit of software deployment that can contain different code and dependencies.
- Containerizing software this way enables developers and IT professionals to deploy them across environments with little or no modification.

Why “Container”?



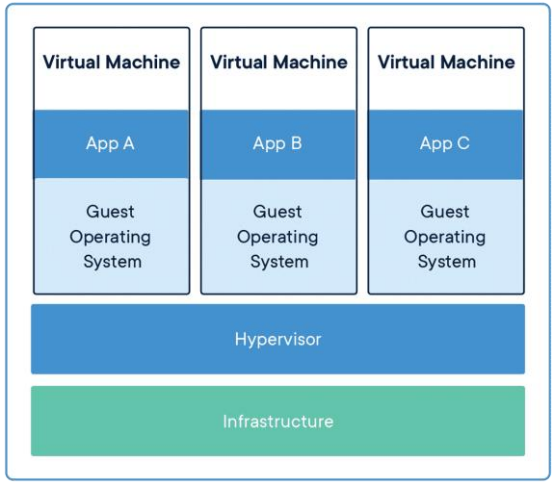
Benefits of Containers

- Ease of setup
  - From an application point of view, instantiating an image (creating a container) is similar to instantiating a process like a service or a web app.
- Less overhead
  - Containers require less system resources than traditional or hardware virtual machine environments because they don’t include operating system images.
- Portability
  - Applications running in containers can be deployed easily to multiple different operating systems and hardware platforms.
- Consistency
  - Applications in containers will run the same, regardless of where they are deployed.
- Greater efficiency
  - Containers allow applications to be more rapidly deployed, patched, or scaled.
- Better application development
  - Containers support agile and DevOps efforts to accelerate development, test, and production cycles.

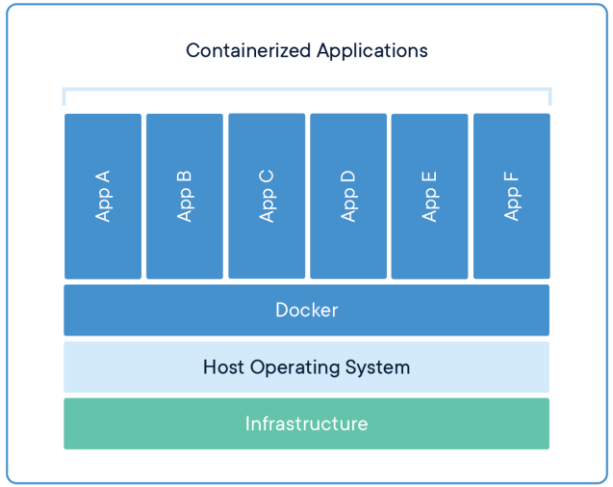
- In short, containers offer the benefits of isolation, portability, agility, scalability, and control across the whole application lifecycle workflow. The most important benefit is the environment's isolation provided between Dev and Ops.

Containers vs Virtual Machines

- Containers and virtual machines have similar resource isolation and allocation benefits, but function differently because containers virtualize the operating system instead of hardware. Containers are more portable and efficient.



- CONTAINERS
  - Containers are an abstraction at the app layer that packages code and dependencies together.
  - Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space.
  - Containers take up less space than VMs (container images are typically tens of MBs in size), can handle more applications and require fewer VMs and Operating systems.



- VIRTUAL MACHINES
  - Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers.
  - The hypervisor allows multiple VMs to run on a single machine.
  - Each VM includes a full copy of an operating system, the application, necessary binaries and libraries – taking up tens of GBs.
  - VMs can also be slow to boot.

Containers vs. Virtual Machines

- Here are three main scenarios in which containers provide compelling advantages compared to virtual machines:
  - Microservices – containers are highly suitable for a microservices architecture, in which applications are broken into small, self-sufficient components, which can be deployed and scaled individually. Containers are an attractive option for deploying and scaling each of those microservices.
  - Multi-cloud – containers provide far more flexibility and portability than VMs in multi-cloud environments. When software components are deployed in containers, it is possible to easily “lift and shift” those containers from on-premise bare metal servers, to on-premise virtualized environments, to public cloud environments.
  - Automation – containers are easily controlled by API, and thus are also ideal for automation and continuous integration / continuous deployment (CI/CD) pipelines.

Docker & Containers

- Why is Docker synonymous with containers?
  - Docker set the industry standard, to a point where the terms '*Docker*' and '*Containers*' are used interchangeably
  - However, there are other tools for containerization

- What other alternatives are there?
  - Artifactory Docker Registry
  - LXC/LXD (Linux)
  - Hyper-V and Windows Containers
  - rkt (works with Kubernetes)
  - Podman (open-source container engine)
  - runC (portability solution)
  - containerd (a container runtime)
  - Buildah
  - BuildKit
  - Kaniko

Architecture & Components

What is a Container? (v2)

- A container is really two different things.
- Like a normal Linux program, containers really have two states - rest and running.
- When at rest:
  - A container is a file (or set of files) that is saved on disk.
  - This is referred to as a Container Image or Container Repository.
- When you start a container:
  - The Container Engine unpacks the required files and meta-data, then hands them off to the the Linux kernel.
  - Starting a container is very similar to starting a normal Linux process and requires making an API call to the Linux kernel. This API call typically initiates extra isolation and mounts a copy of the files that were in the container image.
  - Once running, Containers are just a Linux process. The process for starting containers, as well as the image format on disk, are defined and governed by standards.

Container & Images

Images

- When running a container, it uses an isolated filesystem. This custom filesystem is provided by a container image.
- Since the image contains the container’s filesystem, it must contain everything needed to run an application - all dependencies, configuration, scripts, binaries, etc.
- The image also contains other configuration for the container, such as environment variables, a default command to run, and other metadata.
- In a sense, an image is a read-only template with instructions for creating a Docker container.

Containers

- A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.
- A container is a standard Linux process typically created through a clone() system call instead of fork() or exec().
- By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container’s network, storage, or other underlying subsystems are from other containers or from the host machine.
- A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

Engine, Host & Registry

Container Engine

- A container engine is a piece of software that accepts user requests, including command line options, pulls images, and from the end user's perspective runs the container.

Container Host

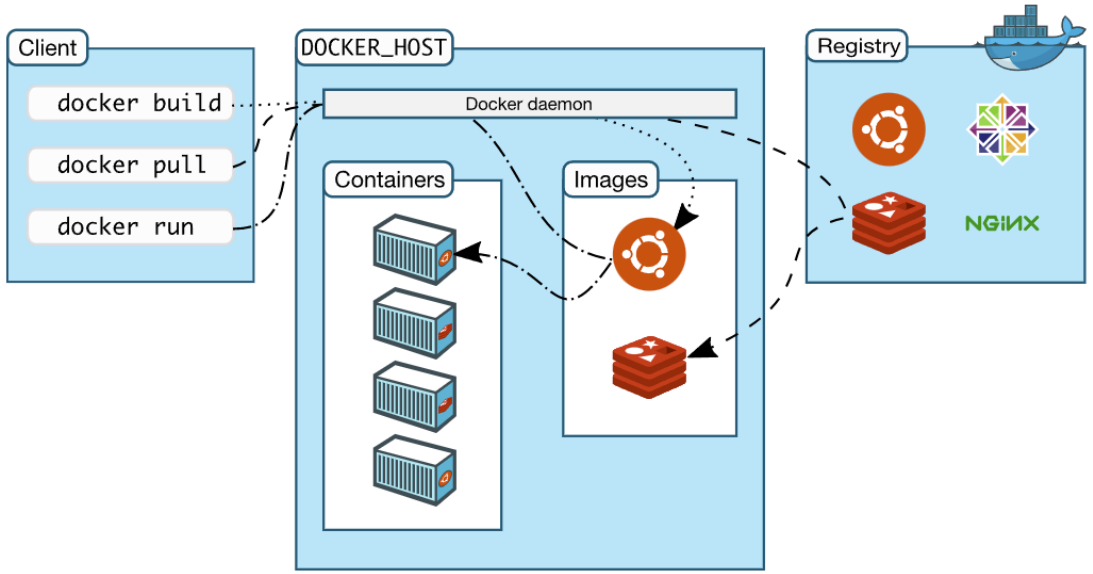
- The container host is the system that runs the containerized processes, often simply called containers.
- Once a container image (aka repository) is pulled from a Registry Server to the local container host, it is said to be in the local cache.

Registry Server

- A registry server is essentially a file server that is used to store docker repositories.
- Typically, the registry server is specified as a normal DNS name and optionally a port number to connect to.
- Much of the value in the docker ecosystem comes from the ability to push and pull repositories from registry servers.
- When a docker daemon does not have a locally cached copy of a repository, it will automatically pull it from a registry server.
- Most Linux distributions have the docker daemon configured to pull from docker.io.

Specific case: Docker

- Docker uses a client-server architecture.
- The Docker client talks to the Docker daemon (*Engine*), which does the heavy lifting of building, running, and distributing your Docker containers.
- The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon.
- The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.
- Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.



Specific case: Docker

Docker daemon:

- The Docker daemon (*dockerd*) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes.
- A daemon can also communicate with other daemons to manage Docker services.

Docker client:

- the Docker client (*docker*) is the primary way that many Docker users interact with Docker.
- When you use commands such as docker run, the client sends these commands to *dockerd*, which carries them out. The docker command uses the Docker API.
- The Docker client can communicate with more than one daemon.

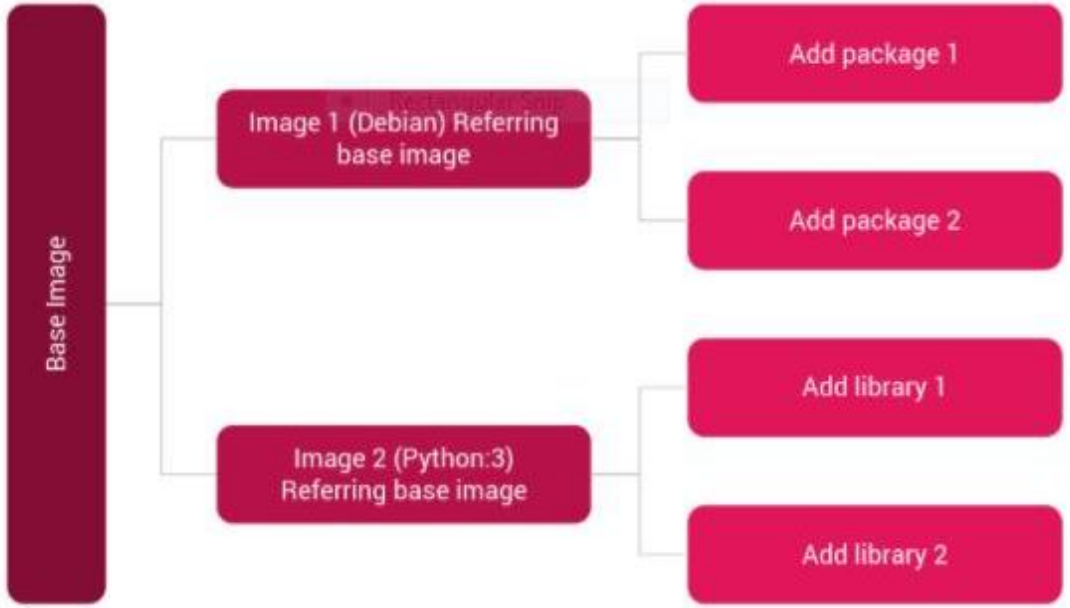
Docker registries:

- A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.
- When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

How to setup a container?

How to setup a container?

- Most of the times, you will use images created by others and published in a registry (although you can create your own).
- To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it.
- Each instruction in a Dockerfile creates a layer in the image.
- When you change the Dockerfile and rebuild the image, only the layers that have changed are rebuilt.
- This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.



Dockerfile

- A **dockerfile** contains a set of instructions that are executed step by step when you use the *docker build* command to build the docker image.
  - It contains instructions and commands that define the structure of your image;
  - instructions related to docker build context;
  - information related to the packages and libraries to be installed in the container;
  - etc.
- Benefits of a Dockerfile:
  - This will allow a developer to create an execution environment and will help him to automate the process and make it repeatable.
  - It provides you with flexibility, readability, accountability and helps in easy versioning of the project.
- The architecture of any Docker image can be considered as a layered structure:
  - It contains some base images which might be pulled from *dockerhub* and over that some modifications inside the base image or addition of new images.
  - This architecture allows users to reuse the pulled docker images, to more efficiently utilise disk storage, and to cache the docker build process.

```
FROM python:3

#set working directory
WORKDIR /usr/src/myapp

RUN apt-get -y update
RUN apt-get -y install vim

#copying all the files in the
  container
COPY . .

#specify the port number to be
  exposed
EXPOSE 8887

# run the command
CMD ["python3", "./file.py"]
```

Dockerfile Commands

FROM

- Almost all dockerfiles starts with the FROM command.
- A FROM command allows you to create a base image such as an operating system, a programming language, etc. All the instructions executed after this command take place on this base image.
- It contains an image name and an optional tag name. If you already have the base image pulled previously in your local machine, it doesn't pull a new one.
- There are several pre-published docker base images available in the docker registry. You can also push your own customized base image inside the docker registry.

RUN

- A RUN instruction is used to run specified commands.
- Each RUN command creates a new cache layer or an intermediate image layer and hence chaining all of them into a single line, becomes efficient.
  However, chaining multiple RUN instructions could lead to cache bursts as well.
- It is more efficient to combine all the RUN instructions into a single one. You can chain multiple RUN instructions in the following way:

```
RUN apt−get −y update \
&& apt−get −y install firefox \
&& apt−get −y install vim
```

```
FROM python:3

#set working directory
WORKDIR /usr/src/myapp

RUN apt-get -y update
RUN apt-get -y install vim

#copying all the files in the
  container
COPY . .

#specify the port number to be
  exposed
EXPOSE 8887

# run the command
CMD ["python3", "./file.py"]
```

Dockerfile Commands

CMD

- The main purpose of a CMD is to provide defaults for an executing container.
- If you want to run a docker container by specifying a default command that gets executed by all containers of that image, you can use a CMD command.
- In case you specify a command during the *docker run* command, it overrides the default one.
- In case you specify more than one CMD instructions, will allow only the last one to get executed.
- Example:
  1. CMD echo "Welcome to SYOSY class“
  2. sudo docker run −it <image_name>
  3. Output − “Welcome to SYOSY class”

```
FROM python:3

#set working directory
WORKDIR /usr/src/myapp

RUN apt-get -y update
RUN apt-get -y install vim

#copying all the files in the
   container
COPY . .

#specify the port number to be
   exposed
EXPOSE 8887

# run the command
CMD ["python3", "./file.py"]
```

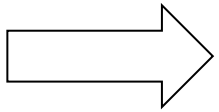
ENTRYPOINT

- An ENTRYPOINT allows you to configure a container that will run as an executable.
- The difference between ENTRYPOINT and CMD is that, if you try to specify default arguments in the *docker run* command, it will not ignore the ENTRYPOINT arguments.
- The exec form of an ENTRYPOINT command is − ENTRYPOINT [“<executable-command>”, “<parameter 1>”, “<parameter 2>”, ….]

Dockerfile - Best Practices

- As we saw earlier, a docker image can be considered as a layered structure, containing some base image which might be pulled from *dockerhub* and over which some modifications can be carried out.
- As a best practice when creating a Dockerfile, **the order of statements matters.**
  - You need to order your steps in such a way that the least frequently changing statements appear first.
  - This is so because when you change or modify a line in the dockerfile and its cache gets invalidated, the subsequent line's will break due to these changes.
  - Hence, you need to keep the most frequently changing lines as last as possible.

```
FROM python:3

#set working directory
WORKDIR /usr/src/myapp

#copying all the files in the container
COPY . .

RUN apt-get -y update
RUN apt-get -y install vim

#specify the port number to be exposed
EXPOSE 8887

# run the command
CMD ["python3", "./file.py"]
```



```
FROM python:3

#set working directory
WORKDIR /usr/src/myapp

RUN apt-get -y update
RUN apt-get -y install vim

#copying all the files in the container
COPY . .

#specify the port number to be exposed
EXPOSE 8887

# run the command
CMD ["python3", "./file.py"]
```

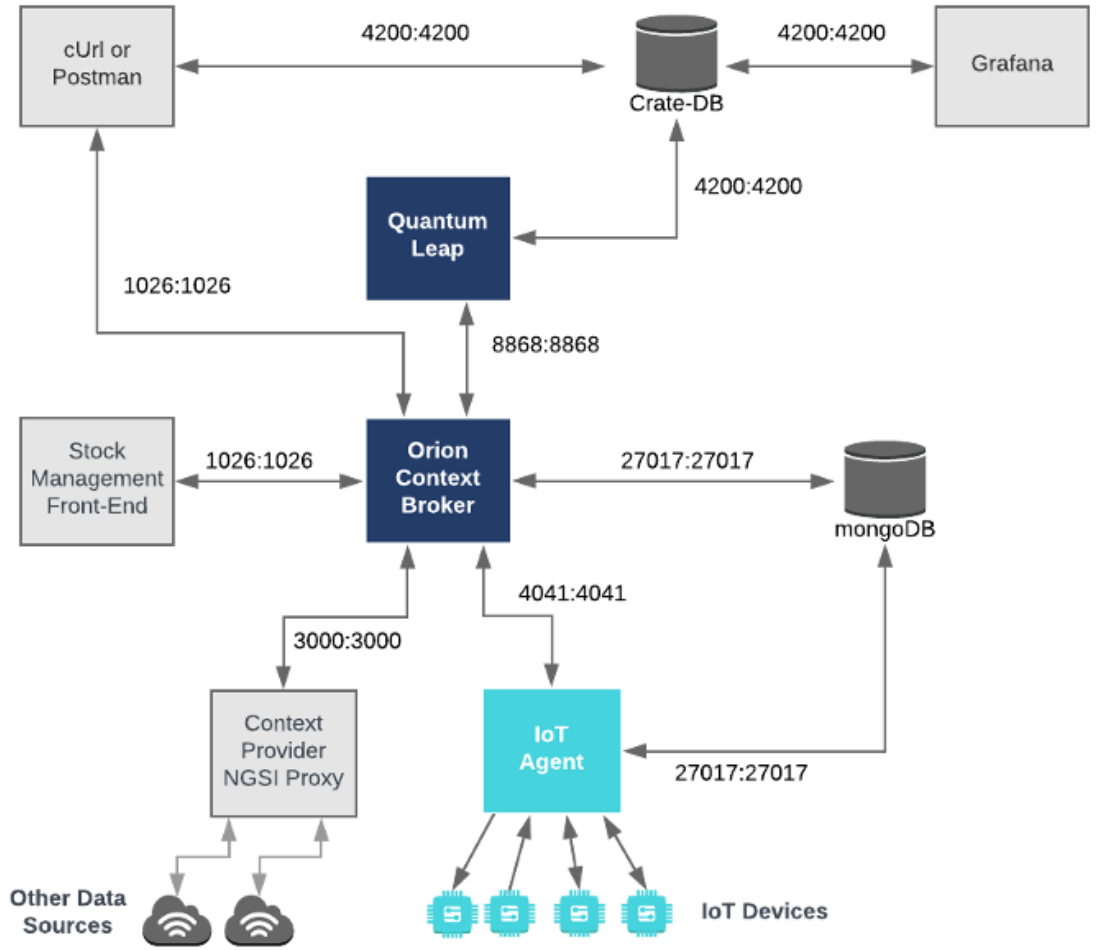
Multi-container Applications

& Docker Networking

Multi-container Applications

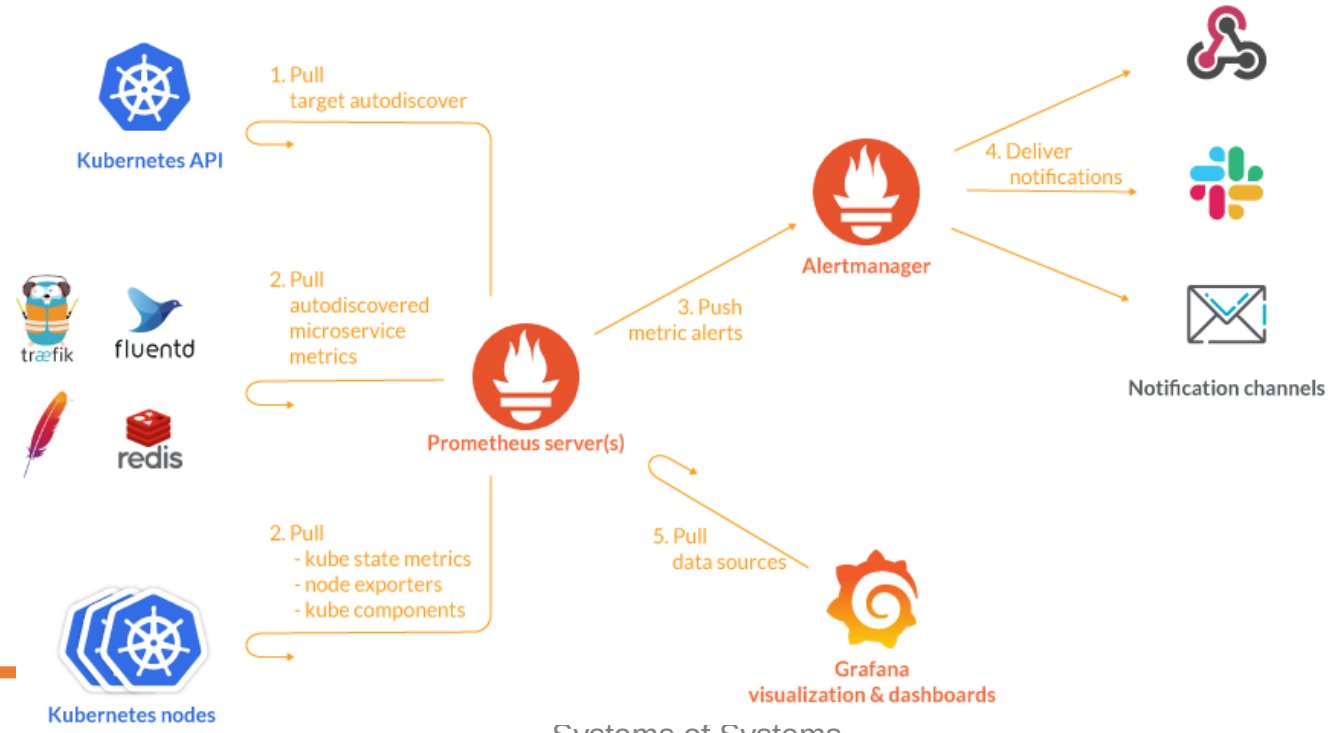
- Often you will need to set up multiple containers cooperating among themselves.

- Example: a FIWARE ecosystem that can be fully deployed in containers:
  - https://fiware-tutorials.readthedocs.io/en/latest/time-series-data.html



Monitoring - Prometheus

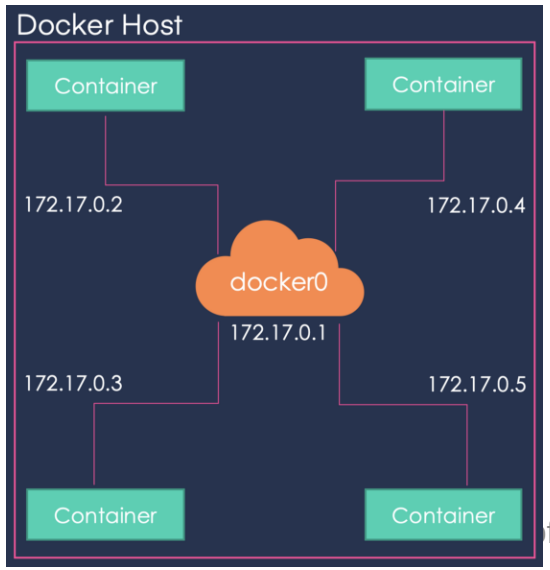
- Multi-container applications can be hard to maintain.
- Monitoring tools for containers are necessary to keep track of what's happening.
- **Prometheus** is an open-source systems monitoring and alerting toolkit, that has been widely adopted as a monitoring tool for containers.
- It collects and stores its metrics as time series data, i.e. metrics information is stored with the timestamp at which it was recorded, alongside optional key-value pairs called labels.

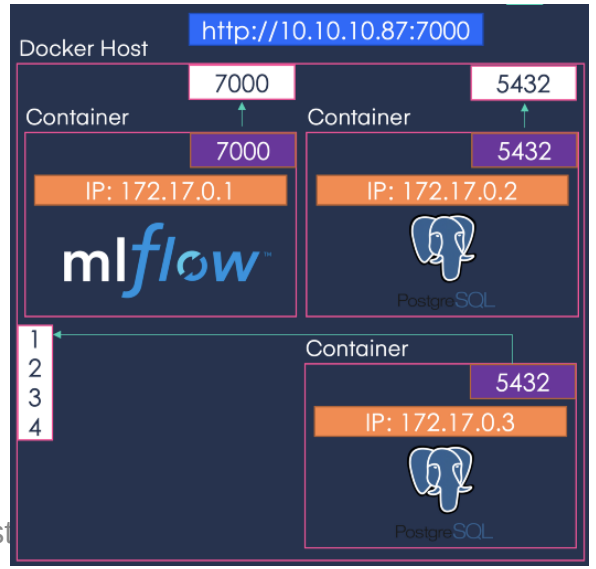


Docker Networking

- You can connect Docker containers and services, or connect them to non-Docker workloads.
- Docker's networking subsystem is pluggable, using drivers. Several drivers exist by default, and provide core networking functionalities:
  - bridge: The default network driver. If you don't specify a driver, this is the type of network you are creating. Bridge networks are usually used when your applications run in standalone containers that need to communicate.
  - host: For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly. At setup time, Container ports are mapped in Host ports.
  - none: For this container, disable all networking. Usually used in conjunction with a custom network driver. none is not available for swarm services.

Bridge





Host

Docker Compose

- Compose is a tool for defining and running multi-container Docker applications.
- You use a YAML file to configure your application's services.
- Using Compose is basically a three-step process:
  1. Define your app's environment with a Dockerfile so it can be reproduced anywhere.
  2. Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
  3. Run docker compose up and the Docker compose command starts and runs your entire app. You can alternatively run docker-compose up using the docker-compose binary.
- Example:

```
version: "3.9"  # optional since v1.27.0
services:
  web:
    build: .
    ports:
      - "8000:5000"
    volumes:
      - .:/code
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

Container Orchestration

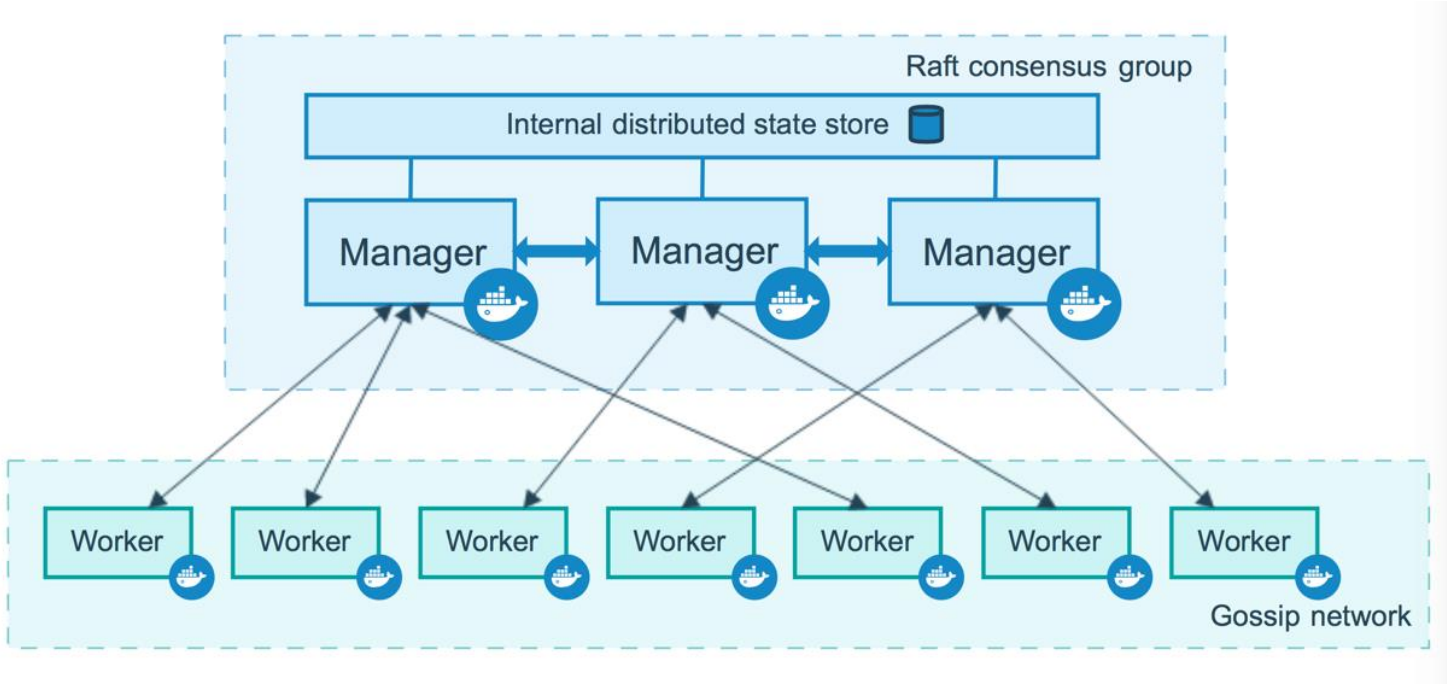
Container Orchestration

- Containers guarantee that applications run the same way across clouds and data centers.
- As we scale our applications up, we need tools to help automate the maintenance of those applications, enable the replacement of failed containers automatically, and manage the rollout of updates and reconfigurations of those containers during their lifecycle.
- Tools to manage, scale, and maintain containerized applications are called **orchestrators**, and the most common examples of these are Kubernetes and Docker Swarm.

- A container orchestrator really does two things:
  - Dynamically schedules container workloads within a cluster of computers.
  - Provides a standardized application definition file (kube *yaml*, docker *compose*, etc)
- The above two features provide many capabilities:
  - Allows containers within an application to be scheduled completely separately.
    - Allows the utilization of large clusters of Container Hosts
    - Failure of individual containers (process hang, out-of-memory), container hosts (disk, network, reboot), or container engines (corruption, restart)
    - Individual containers need to be scaled up, or scaled down
  - It's easy to deploy new instances of the same application into new environments.

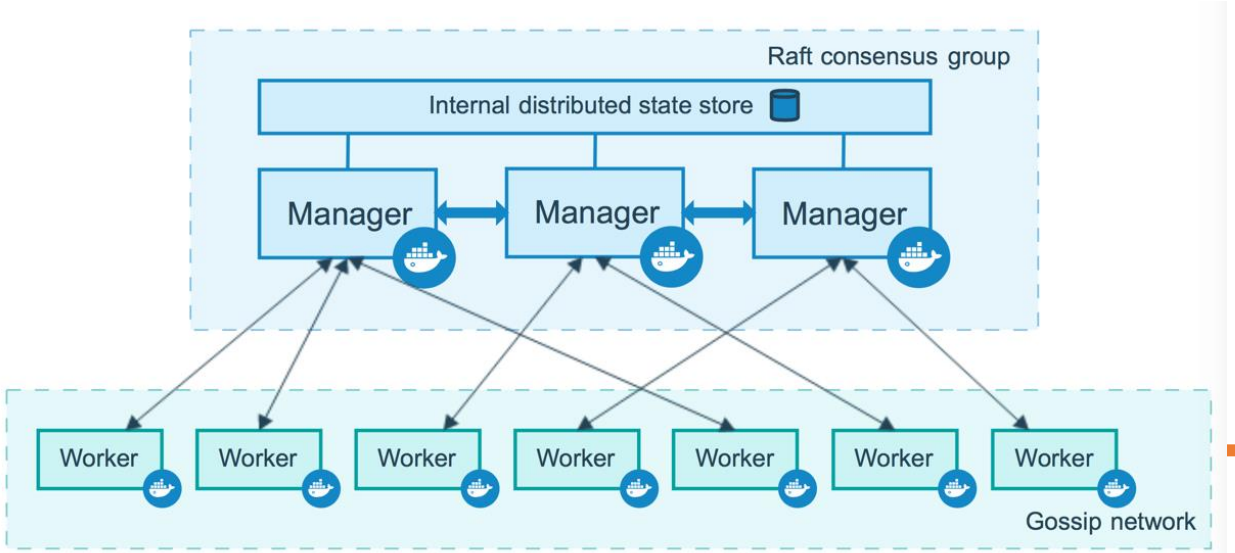
Docker Swarm

- Swarm provides many tools for scaling, networking, securing and maintaining your containerized applications, above and beyond the abilities of containers themselves.
- Docker Engine 1.12 introduces **swarm mode** that enables you to create a cluster of one or more Docker Engines called a swarm. A swarm consists of one or more nodes: physical or virtual machines running Docker Engine 1.12 or later in swarm mode.
- There are two types of nodes: **managers** and **workers**.



What is a Swarm?

- **What is a Docker Swarm?**
  - 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.
- **Why:**
  - 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 running containers on other nodes.
- **Analogy:** In the same way that you can use Docker Compose to define and run containers, you can define and run Swarm service stacks.



Swarm Nodes

- A node is an instance of the Docker engine participating in the swarm.
- Typically you can find Docker nodes distributed across multiple physical and cloud machines.

Manager nodes:

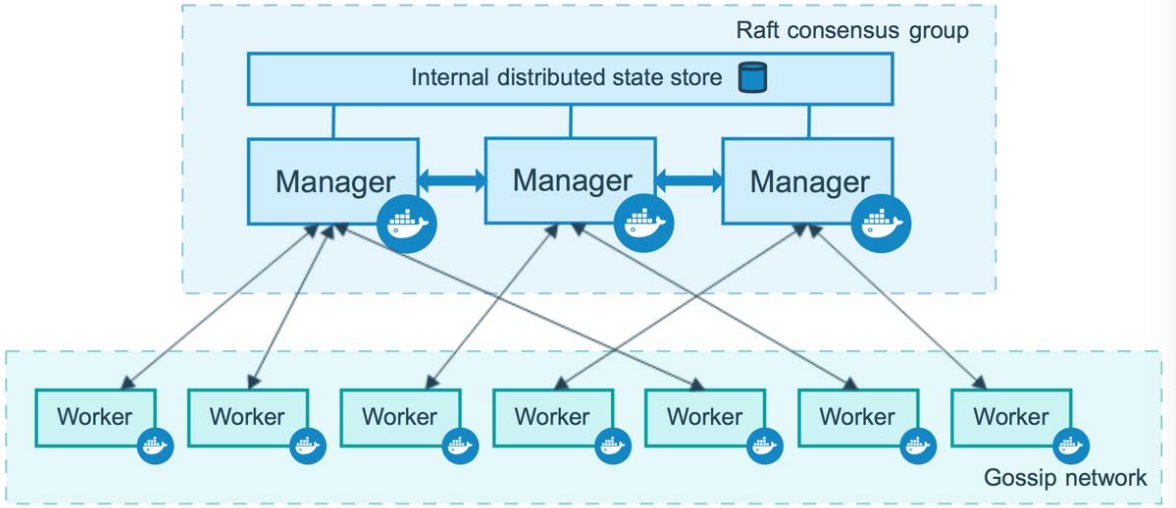
- To deploy your application to a swarm, you submit a service definition to a manager node.
- The manager node dispatches units of work called tasks to worker nodes.
- Manager nodes also perform the orchestration and cluster management functions required to maintain the desired state of the swarm.
- Manager nodes elect a single leader to conduct orchestration tasks.

Worker nodes:

- Worker nodes are instances of Docker Engine whose sole purpose is to execute containers.
- Worker nodes receive and execute tasks dispatched from manager nodes.
- By default, all managers are also workers, i.e., manager nodes also run services as worker 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.

Fault-tolerance

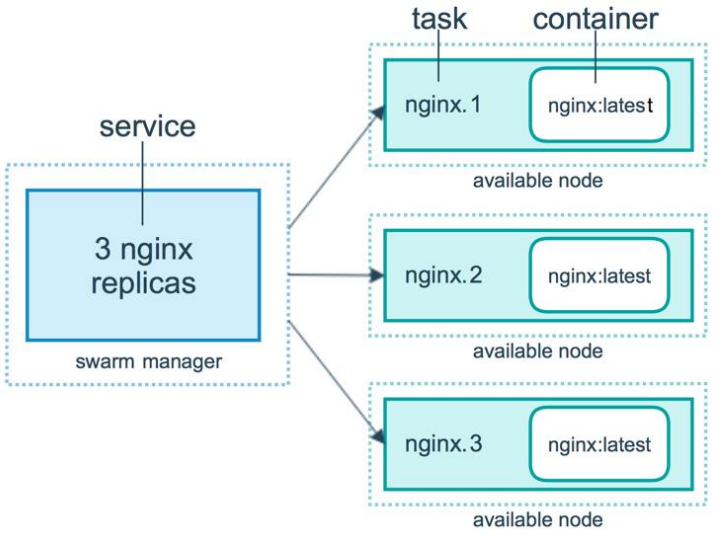
- Manager nodes handle cluster management tasks:
  - maintaining cluster state
  - scheduling services
  - serving swarm mode HTTP API endpoints



- The managers maintain a consistent internal state of the entire swarm and all the services running on it using Raft, a consensus algorithm (consensus involves multiple servers agreeing on values).
- To take advantage of swarm mode's fault-tolerance features, Docker recommends you implement an odd number of nodes.
- When you have multiple managers you can recover from the failure of a manager node without downtime. Cases:
  - 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.

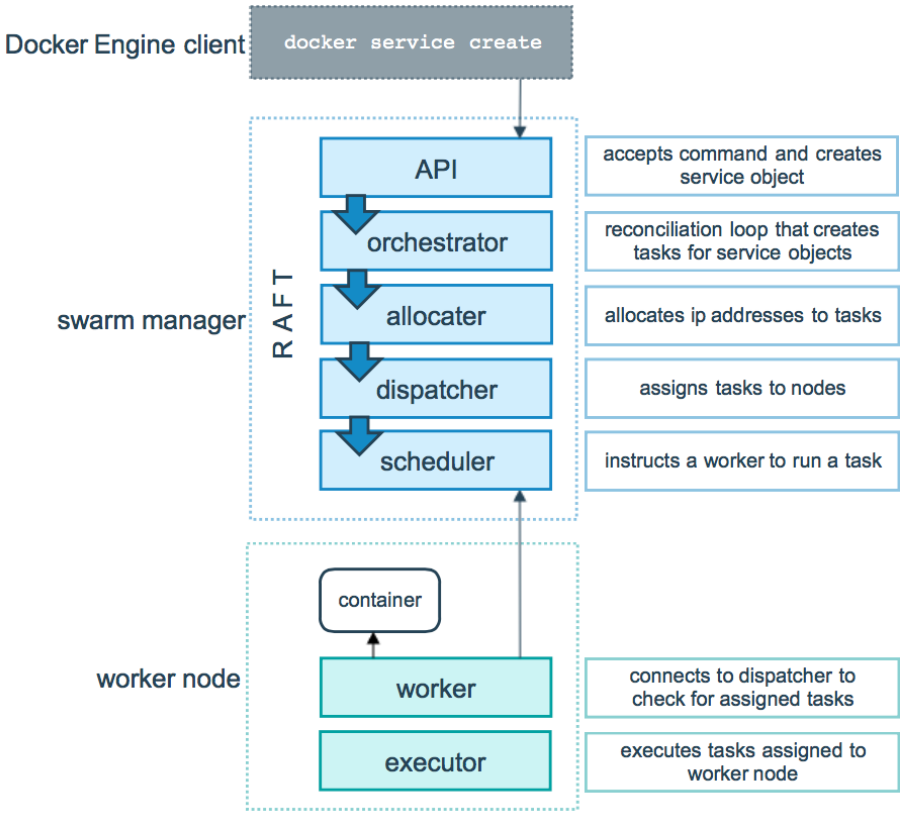
Services and tasks

- Service:
  - A service is the definition of the tasks to execute on the manager or worker nodes.
  - When you create a service, you specify which container image to use and which commands to execute inside running containers.
- Task:
  - A task is the atomic unit of scheduling within a swarm.
  - When you declare a desired service state by creating or updating a service, the orchestrator realizes the desired state by scheduling tasks.
  - If the task fails the orchestrator removes the task and its container and then creates a new task to replace it according to the desired state specified by the service.
- Example:
  1. For instance, you define a service that instructs the orchestrator to keep three instances of an HTTP listener running at all times.
  2. The orchestrator responds by creating three tasks.
  3. Each task is a slot that the scheduler fills by spawning a container. The container is the instantiation of the task.
  4. If an HTTP listener task subsequently fails its health check or crashes, the orchestrator creates a new replica task that spawns a new container.



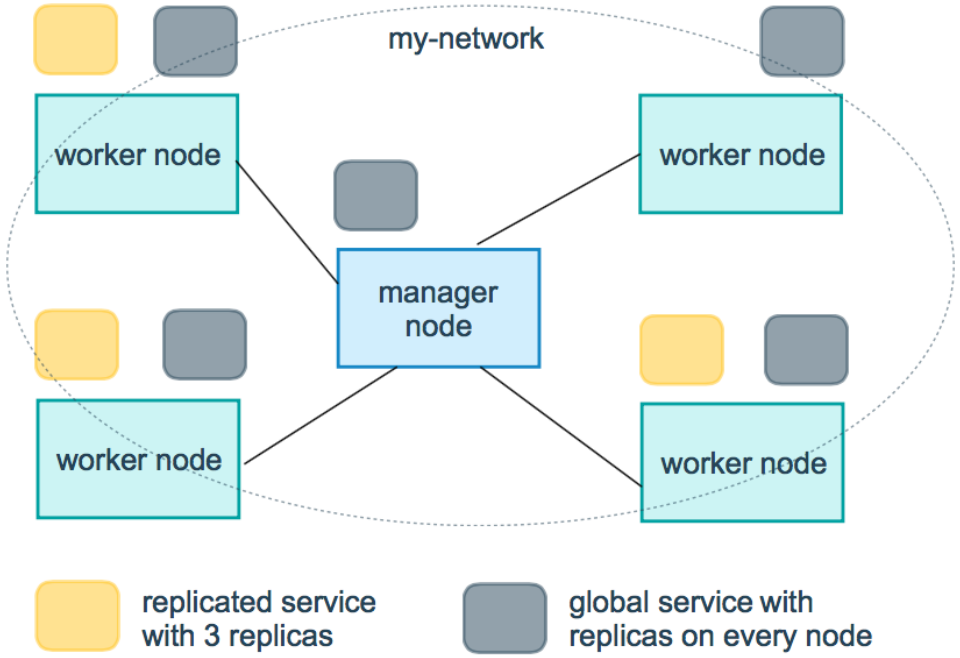
Tasks and scheduling

- The underlying logic of Docker swarm mode is a general purpose scheduler and orchestrator.
- The service and task abstractions themselves are unaware of the containers they implement.
- Hypothetically, you could implement other types of tasks such as virtual machine tasks or non-containerized process tasks.
- The scheduler and orchestrator are agnostic about the type of task. However, the current version of Docker only supports container tasks.



Global Services

- For global services, the swarm runs one task for the service on every available node in the cluster.
- A task carries a Docker container and the commands to run inside the container.



Other Orchestrators

- **Kubernetes** has become the de facto standard in container orchestration, similar to Linux before it.
- There are many container schedulers being developed in the community and by vendors.
- Historically, Swarm, Mesos, and Kubernetes were the big three, but recently even Docker and Mesosphere have announced support for Kubernetes - as has almost every major cloud service provider.

Thank You

Build

Build an image from the Dockerfile in the current directory and tag the image
```
docker build -t myimage:1.0 .
```

List all images that are locally stored with the Docker Engine
```
docker image ls
```

Delete an image from the local image store
```
docker image rm alpine:3.4
```

Share

Pull an image from a registry
```
docker pull myimage:1.0
```

Retag a local image with a new image name and tag
```
docker tag myimage:1.0 myrepo/myimage:2.0
```

Push an image to a registry
```
docker push myrepo/myimage:2.0
```

Run

Run a container from the Alpine version 3.9 image, name the running container "web" and expose port 5000 externally, mapped to port 80 inside the container.
```
docker container run --name web -p 5000:80 alpine:3.9
```

Stop a running container through SIGTERM
```
docker container stop web
```

Stop a running container through SIGKILL
```
docker container kill web
```

List the networks
```
docker network ls
```

List the running containers (add `--all` to include stopped containers)
```
docker container ls
```

Delete all running and stopped containers
```
docker container rm -f $(docker ps -aq)
```

Print the last 100 lines of a container's logs
```
docker container logs --tail 100 web
```

Docker Management

All commands below are called as options to the base `docker` command. `Run docker <command> --help` for more information on a particular command.

`app*`	***Docker Application***
`assemble*`	***Framework-aware builds (Docker Enterprise)***
`builder`	***Manage builds***
`cluster`	***Manage Docker clusters (Docker Enterprise)***
`config`	***Manage Docker configs***
`context`	***Manage contexts***
`engine`	***Manage the docker Engine***
`image`	***Manage images***
`network`	***Manage networks***
`node`	***Manage Swarm nodes***
`plugin`	***Manage plugins***
`registry*`	***Manage Docker registries***
`secret`	***Manage Docker secrets***
`service`	***Manage services***
`stack`	***Manage Docker stacks***
`swarm`	***Manage swarm***
`system`	***Manage Docker***
`template*`	***Quickly scaffold services (Docker Enterprise)***
`trust`	***Manage trust on Docker images***
`volume`	***Manage volumes***

*Experimental in Docker Enterprise 3.0.