Cloud Computing Systems

Lecture 9-10

Nuno Preguiça

(nuno.preguica_at_fct.unl.pt)

OUTLINE

Containers

- Introduction
- Under the hood
  - Kernel namespaces
  - Cgroups
  - Copy-on-write File system
- Docker
- Docker compose
- Docker swarm
- Kuebernetes

Virtual machines: Pros

Efficiency. A virtual machine allows to efficiently use resource and provides isolation.

Flexibility. Resources can be allocated as needed.

Backup and recovery. Virtual machines can be stored as a single file that can be easily backed up on another source.

OS freedom. Different guest OSs can exist on the same hypervisor.

Performance and moving. Hypervisors support moving a virtual machine from one host to another in case of performance degradation on the host machine.

VIRTUAL MACHINES: CONS

Performance overhead. A VM stack includes the guest OS, the hypervisor and potentially the host OS (for type 2 hypervisors).

Efficient resource utilization. Using multiple OSs in the same hypervisor duplicates the used resources.

WHY NOT USING SIMPLE PROCESSES INSTEAD OF VMS?

Isolation. We want that an application does not affect other applications in any way.

- E.g.: be sure that a malicious user in a web application cannot gain access to the entire server.

Manage application dependencies. Different applications have different dependencies – libraries, library versions, etc. Sometimes it is complex to install all dependencies of an application or keep the dependencies of all application in the same system.

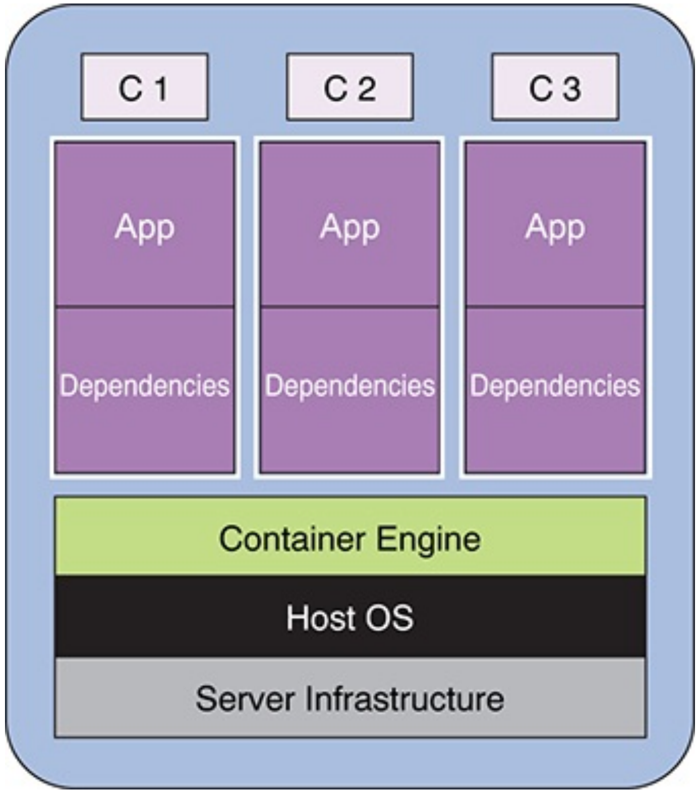
WHAT ARE CONTAINERS?

Containers provide OS-level virtualization.

Provides private namespace, network interface and IP address, file systems, etc.

Unlike VMs, containers share the host system's kernel with other containers.

- Application code needs to use the Host OS API.



Containers promises

Build once, run anywhere

- Faster deployment
- Portability across machines
- Version control
- Simplified dependency management

BEFORE CONTAINERS

Isolating applications is problem that exists for years.

chroot – Allows to specify a directory as the root directory for an application. This makes it impossible for an application to access other application files (and other resources, depending on the systems).

Chroot isolation not perfect. The process can still access the underlying IO devices, it can execute a second chroot if it has enough privileges.

All application dependencies need to be copied into the chroot directory.

OUTLINE

Containers

- Introduction
- **Under the hood**
  - **Kernel namespaces**
  - **Cgroups**
  - **Copy-on-write File system**
- Docker
- Docker compose
- Docker swarm
- Kuebernetes

DOCKER

Docker is the most popular container technology.

It builds on the following technologies:

- Kernel namespaces
- Cgroups
- Copy-on-write File system

Docker

Docker is the most popular container technology.

It builds on the following technologies:

- **Kernel namespaces**
- Cgroups
- Copy-on-write File system

KERNEL NAMESPACES

Kernel namespaces split kernel resources (processes, users, network stacks, etc.) into one instance per namespace.

A process only views the resources in its namespace.

There are currently 6 namespaces:

- mnt (mount points, filesystems)
- pid (processes)
- net (network stack)
- ipc (System V IPC)
- uts (hostname)
- user (UIDs)

KERNEL NAMESPACES: IMPLEMENTATION

Support for kernel namespaces added to the kernel.

New system calls:

- clone() - creates a new process and a new namespace;
  - The process is associated to the new namespace.
- unshare() - creates a new namespace and attaches the current process to it.
- setns() - allows for joining an existing namespace.

USE OF KERNEL NAMESPACES

Kernel namespaces are used to create isolated containers that have no visibility to objects outside the container.

The processes running inside a container share the underlying kernel with other containers.

SOME MORE INFO

UTS Namespace:

- Provides namespace-specific hostname and domain name.

Network Namespace:

- A network namespace is logically a copy of the network stack, with its own routes, firewall rules, and network devices.
- Each network namespace has its own IP addresses.
- A network device belongs to exactly one network namespace. A socket belongs to exactly one network namespace.
- Communicating between two network namespaces:
  - Veth (virtual ethernet) is used like a pipe between two namespaces
  - Sockets also work

SOME MORE INFO (2)

Mount Namespace:

- On creation, the file system tree is copied to new space, with all previous mounts visible.
- Future mounts/unmounts invisible to the rest of the system.

PID Namespace:

- Processes in different PID namespaces can have the same process ID.

User Namespace:

- A process will have distinct set of UIDs, GIDs and capabilities.

IPC Namespace:

- Each namespace gets its own IPC objects and POSIX message queues.

DOCKER

Docker is the most popular container technology.

It builds on the following technologies:

- Kernel namespaces
- **Cgroups**
- Copy-on-write File system

CONTROL GROUPS (CGROUPS)

Cgroups are a mechanism for applying hardware resource limits and access controls to a process or collection of processes.

The cgroup mechanism and the related subsystems provide a tree-based hierarchical, inheritable and optionally nested mechanism of resource control.

CGROUPS UNDER THE HOOD

The implementation of cgroups requires a few, simple hooks into the rest of the kernel: in boot phase, process creation and destroy.

All operations on cgroups are executed using operations on a VFS (virtual file system).

There are 11 cgroup subsystems: cpuset, freezer, mem, blkio, net_cls, net_prio, devices, perf, hugetlb, cpu_cgroup, cpuacct.

CGROUPS UNDER THE HOOD (2)

cpu, cpuacct, cpuset – allows to control the minimum and maximum CPU time of the processes in a cgroup, and to assign processes to a cgroup.

memory – allows to limit the memory used by a cgroup.

devices – allows to control which processes may create devices and open them for eeading and writing.

freezer – allows to suspend and restore all processes in a cgroup.

net_cls, net_prio – allows to give priorities, per network interface, and to place a classid on packets created in a cgroup; this classid can be used in firewall rules, shape traffic, etc. (does not apply to incoming traffic)

CGROUPS UNDER THE HOOD (2)

blkio – controls and limits access to specified block devices.

perf_event – allows perf monitoring of the set of processes in a cgroup.

hugetlb - supports limiting the use of huge pages by cgroups.

pids – allows limiting the number of process that may be created in a cgroup (and its descendants).

rdma - permits limiting the use of RDMA/IB-specific resources per cgroup.

USE OF CGROUPS

Cgroups are used to limit the memory and CPU consumption of containers. A container can be resized by simply changing the limits of its corresponding cgroup.

DOCKER

Docker is the most popular container technology.

It builds on the following technologies:

- Kernel namespaces
- Cgroups
- **Copy-on-write File system**

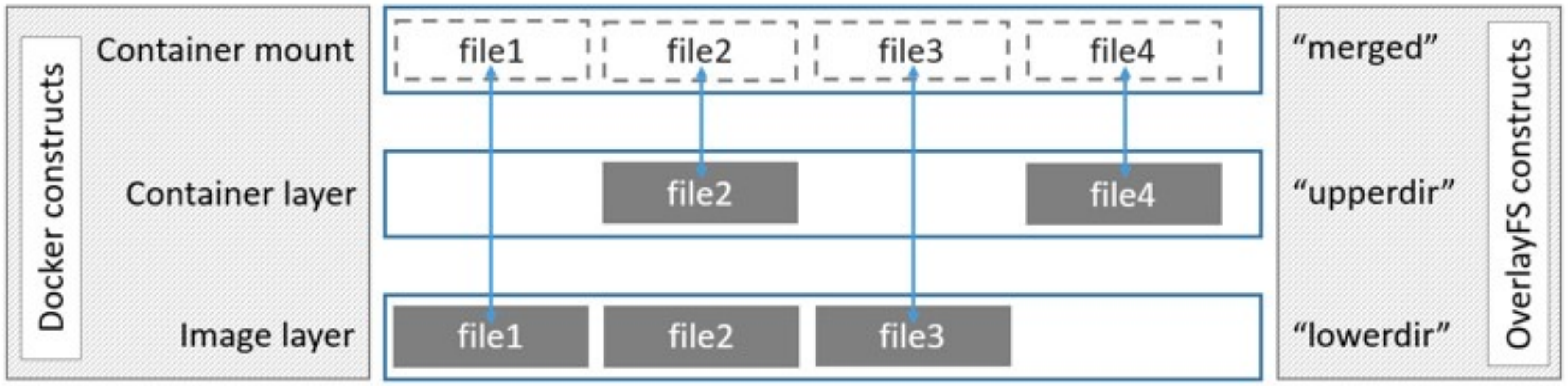
UNIONFS

UnionFS is a copy-on-write file system that is the union of existing file systems.

Gives a unified view of the file system, that combines all stacked file systems.

On write to the UnionFS, the overwritten data is saved to a new path, specific to that container.

- Thus, writes of one container do not affect reads of another container



USE OF UNIONFS

UnionFS allows several containers to share common data. Each layer is only stored once.

OUTLINE

Containers

- Introduction
- Under the hood
  - Kernel namespaces
  - Cgroups
  - Copy-on-write File system
- **Docker**
- Docker compose
- Docker swarm
- Kuebernetes

DOCKER

Built on top of kernel namespaces, cgroups, unionFS, and capabilities.

Each container gets its own set of namespaces and cgroups.

Namespaces isolate containers from each other: one container cannot even see the list of processes in another container.

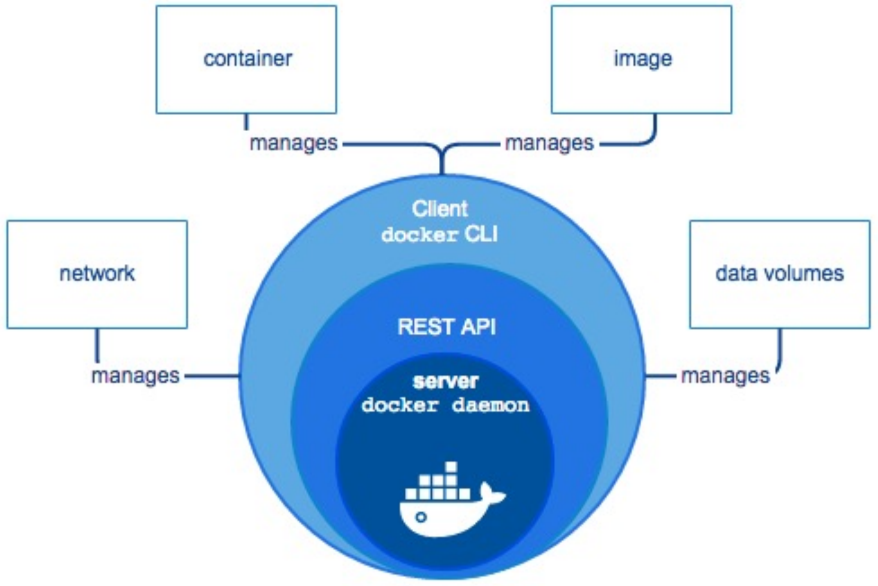
Cgroups allow the admin to isolate the resources used by each container and its children.

Running the docker daemon requires root privileges.

Docker provides a whitelist of capabilities to root users inside a container.

DOCKER ENGINE

- Docker daemon (dockerd) manages Docker objects such as images, containers, networks, and volumes.
- The docker client sends requests to docker daemon.

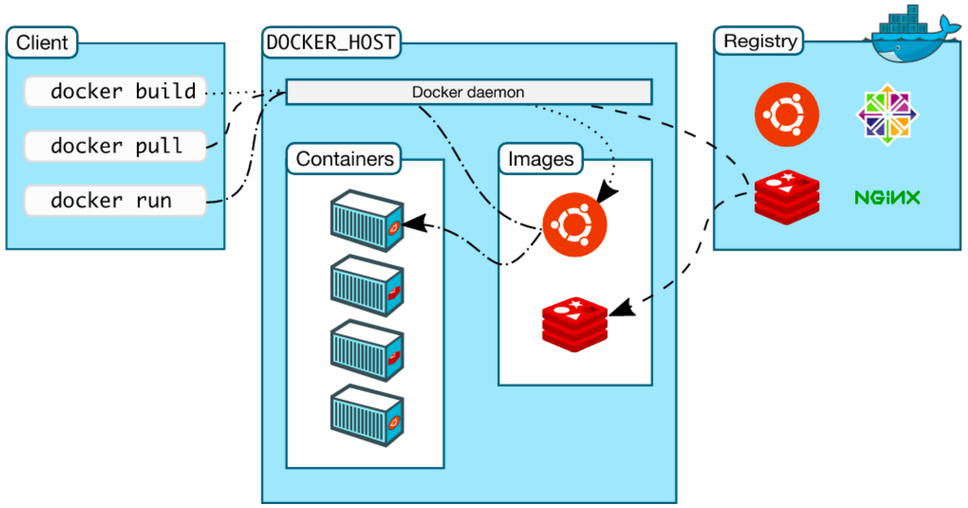


Docker engine (2)

A Docker *registry* stores Docker images. Docker is configured to search in Docker Hub by default.

An *image* is a read-only template with instructions for creating a Docker container. Often, an image is *based on* another image, with some additional customization.

A Docker image can be created from the specification in a Dockerfile.



CREATING DOCKER IMAGE

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

Initializes a new build stage and sets the Base Image.

`ADD <src>... <dest>`

`COPY <src>... <dest>`

Copies new files, directories from <src> and adds them to the filesystem of the image at the path <dest>.

ADD allows to use URL as src and unpacks (tar, bzip) archives.

This creates a new layer.

```
FROM jboss/wildfly:14.0.1.Final

COPY *.war
/opt/jboss/wildfly/standalone/deploymen
ts/

RUN /opt/jboss/wildfly/bin/add-user.sh
admin Admin#70365 --silent

EXPOSE 9990

CMD
["/opt/jboss/wildfly/bin/standalone.sh"
, "-b", "0.0.0.0", "-bmanagement",
"0.0.0.0"]
```

CREATING DOCKER IMAGE (2)

RUN <command>

RUN ["exec","param1","param2"]

Execute a commands in a new layer on top of the current image and commit the results.

As a RUN creates a new layer, cleanup should be made in the same command.

```
RUN apt-get update && \
  apt-get install -y \
  --no-install-recommends \
  g++ \
  gcc \
  libc6-dev \
  make \
  && rm -rf /var/lib/apt/lists/*
```

```
FROM jboss/wildfly:14.0.1.Final


COPY *.war
/opt/jboss/wildfly/standalone/deploymen
ts/


RUN /opt/jboss/wildfly/bin/add-user.sh
admin Admin#70365 --silent


EXPOSE 9990


CMD
["/opt/jboss/wildfly/bin/standalone.sh"
, "-b", "0.0.0.0", "-bmanagement",
"0.0.0.0"]
```

CREATING DOCKER IMAGE (3)

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

Informs Docker that the container listens on the specified network ports at runtime. By default, protocol is assumed to be TCP.

```
FROM jboss/wildfly:14.0.1.Final

COPY *.war
/opt/jboss/wildfly/standalone/deploymen
ts/

RUN /opt/jboss/wildfly/bin/add-user.sh
admin Admin#70365 --silent

EXPOSE 9990

CMD
["/opt/jboss/wildfly/bin/standalone.sh"
, "-b", "0.0.0.0", "-bmanagement",
"0.0.0.0"]
```

CREATING DOCKER IMAGE (4)

`CMD command param1 param2`

`CMD ["exec","param1","param2"]`

Sets the command to be executed when running the image.

Only one command per image is allowed. If more than one command is specified, only the last one is executed. When running the docker, the command can be overridden.

```
FROM jboss/wildfly:14.0.1.Final


COPY *.war
/opt/jboss/wildfly/standalone/deploymen
ts/


RUN /opt/jboss/wildfly/bin/add-user.sh
admin Admin#70365 --silent


EXPOSE 9990


CMD
["/opt/jboss/wildfly/bin/standalone.sh"
, "-b", "0.0.0.0", "-bmanagement",
"0.0.0.0"]
```

CREATING DOCKER IMAGE (5)

`ENTRYPOINT command param1 param2`

`ENTRYPOINT ["exec","param1","param2"]`

Sets the command to be executed when running the image. This cannot be overridden when starting the image.

Parameters specified when starting the image will be passed as parameters.

```
FROM jboss/wildfly:14.0.1.Final

COPY *.war
/opt/jboss/wildfly/standalone/deploymen
ts/

RUN /opt/jboss/wildfly/bin/add-user.sh
admin Admin#70365 --silent

EXPOSE 9990

CMD
["/opt/jboss/wildfly/bin/standalone.sh"
, "-b", "0.0.0.0", "-bmanagement",
"0.0.0.0"]
```

CREATING DOCKER IMAGE (6)

WORKDIR /path/to/workdir

Sets the environment variable <key> to the value <value>, for both the build process and when the container runs.

```
FROM jboss/wildfly:14.0.1.Final

WORKDIR /opt/jboss/wildfly

ADD scc-backend-aula4-0.1.war
standalone/deployments/

RUN bin/add-user.sh admin Admin#70365 -
-silent

EXPOSE 9990

CMD ["bin/standalone.sh", "-b",
"0.0.0.0", "-bmanagement", "0.0.0.0"]
```

Creating Docker Image (7)

`ENV <key> <value>`

`ENV <key>=<value> ...`

Sets the environment variable <key> to the value <value>, for both the build process and when the container runs.

`USER <user>`

Sets the user name (or UID) and optionally the user group (or GID) to use when running the image and for any RUN, CMD and ENTRYPOINT instructions that follow it in the Dockerfile.

```
FROM jboss/wildfly:14.0.1.Final

COPY *.war
/opt/jboss/wildfly/standalone/deploymen
ts/

RUN /opt/jboss/wildfly/bin/add-user.sh
admin Admin#70365 --silent

EXPOSE 9990

CMD
["/opt/jboss/wildfly/bin/standalone.sh"
, "-b", "0.0.0.0", "-bmanagement",
"0.0.0.0"]
```

CREATING DOCKER IMAGE (8)

```
VOLUME ["/data"]
```

Creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers.

```
FROM jboss/wildfly:14.0.1.Final

COPY *.war
/opt/jboss/wildfly/standalone/deploymen
ts/

RUN /opt/jboss/wildfly/bin/add-user.sh
admin Admin#70365 --silent

EXPOSE 9990

CMD
["/opt/jboss/wildfly/bin/standalone.sh"
, "-b", "0.0.0.0", "-bmanagement",
"0.0.0.0"]
```

BUILD DOCKER IMAGE

```
docker build [OPTIONS] PATH
```

Builds a docker image. PATH should specify a directory containing a Dockerfile and all resources to be copied.

Some options:

-t tag: specifies the tag for the built image.

Build docker image (9)

```
nmp$ docker build -t nunopreguica/ccs1920-app container
Sending build context to Docker daemon  20.94MB
Step 1/6 : FROM jboss/wildfly:14.0.1.Final
 ---> 8c9bcba630f0
Step 2/6 : WORKDIR /opt/jboss/wildfly
 ---> Using cache
 ---> d6992eeae570
Step 3/6 : ADD scc-backend-aula4-0.1.war  standalone/deployments/
 ---> Using cache
 ---> 46f3931aff1a
Step 4/6 : RUN bin/add-user.sh admin Admin#70365 --silent
 ---> Using cache
 ---> 24fd2f62ab29
Step 5/6 : EXPOSE 9990
 ---> Using cache
 ---> 025cf1e0321b
Step 6/6 : CMD ["bin/standalone.sh", "-b", "0.0.0.0", "-bmanagement", 
"0.0.0.0"]
 ---> Using cache
 ---> 4a7f4c112ff0
Successfully built 4a7f4c112ff0
Successfully tagged nunopreguica/ccs1920-app:latest
```

Docker run

```
docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]
```

Runs an image, downloading it if necessary.

Some options:

-P : Publish all exposed ports to the host interfaces

-p local_port:container_port

-p=[] : Publish a container's port or a range of ports to the host

```
$ docker run -p 9990:9990 -p 8080:8080 nunopreguica/ccs1920-app
```

-it : Runs the image in interactive mode.

```
$ docker run -it ubuntu:14.04 /bin/bash
```

DOCKER RUN (2)

```
docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]
```

Runs an image, downloading it if necessary.

Some options:

-v, --volume=[host-src:]container-dest[:<options>]: Bind mount a volume.

```
$ docker run -v $(pwd):/config -t nunopreguica/ccs1920-
test artillery run create-posts.yml
```

DOCKER RUN (3)

-c, --cpu-shares=0	CPU shares (relative weight)
--cpus=0.000	Number of CPUs. Number is a fractional number. 0.000 means no limit.
--cpu-period=0	Limit the CPU CFS (Completely Fair Scheduler) period
--cpuset-cpus=""	CPUs in which to allow execution (0-3, 0,1)
--cpuset-mems=""	Memory nodes (MEMs) in which to allow execution (0-3, 0,1). Only effective on NUMA systems.
--cpu-quota=0	Limit the CPU CFS (Completely Fair Scheduler) quota
--cpu-rt-period=0	Limit the CPU real-time period. In microseconds. Requires parent cgroups be set and cannot be higher than parent. Also check rtprio ulimits.
--cpu-rt-runtime=0	Limit the CPU real-time runtime. In microseconds. Requires parent cgroups be set and cannot be higher than parent. Also check rtprio ulimits.

DOCKER RUN (4)

-m, --memory=""	Memory limit (format: <number>[<unit>]). Number is a positive integer. Unit can be one of b, k, m, or g. Minimum is 4M.
--memory-swap=""	Total memory limit (memory + swap, format: <number>[<unit>]). Number is a positive integer. Unit can be one of b, k, m, or g.
--memory-reservation=""	Memory soft limit (format: <number>[<unit>]). Number is a positive integer. Unit can be one of b, k, m, or g.
--kernel-memory=""	Kernel memory limit (format: <number>[<unit>]). Number is a positive integer. Unit can be one of b, k, m, or g. Minimum is 4M.

Other docker commands

```
docker ps [OPTIONS]
```

Lists containers.

```
docker kill [OPTIONS] CONTAINER [CONTAINER...]
```

Kills one or more containers.

OTHER DOCKER COMMANDS (2)

```
docker images [OPTIONS] [REPOSITORY[:TAG]]
```

Lists images.

```
docker pull [OPTIONS] NAME[:TAG|@DIGEST]
```

Pulls an image from a registry.

```
docker push [OPTIONS] NAME[:TAG]
```

Push an image or a repository to a registry.

Docker networking

There are several options of networking.

Bridge networking allows to connect several dockers containers running in the same docker host.

A network can be created using docker network create.

```
$ docker network create my-net
```

When running a docker, you can specify it will be in the network. The following example would allow to run the server and artillery client in the same network.

```
$ docker run --network=my-net --name=server
nunopreguica/ccs1920-app
```

```
$ docker run --network=my-net  -v $(pwd):/config -t
nunopreguica/ccs1920-test artillery run create-posts.yml
```

DOCKER NETWORKING

The **overlay** network driver creates a distributed network among multiple Docker daemon hosts.

Docker storage

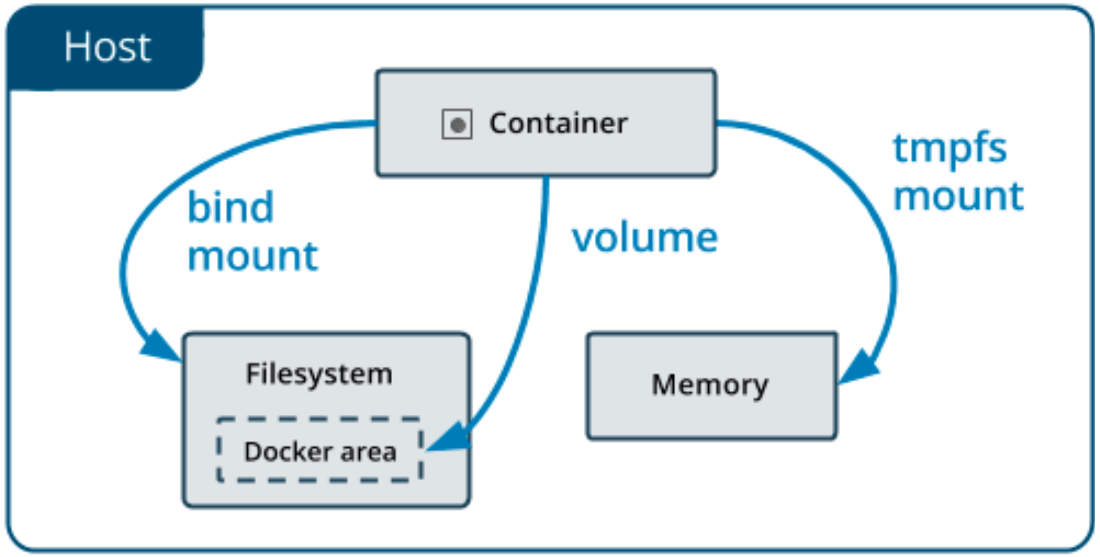
By default all files created inside a container are stored on a writable container layer that is not persisted.

Docker has two options for containers to store files in the host machine, so that the files are persisted even after the container stops: *volumes*, and *bind mounts*.

DOCKER STORAGE: VOLUMES

Volumes are stored in a part of the host filesystem which is *managed by Docker*.

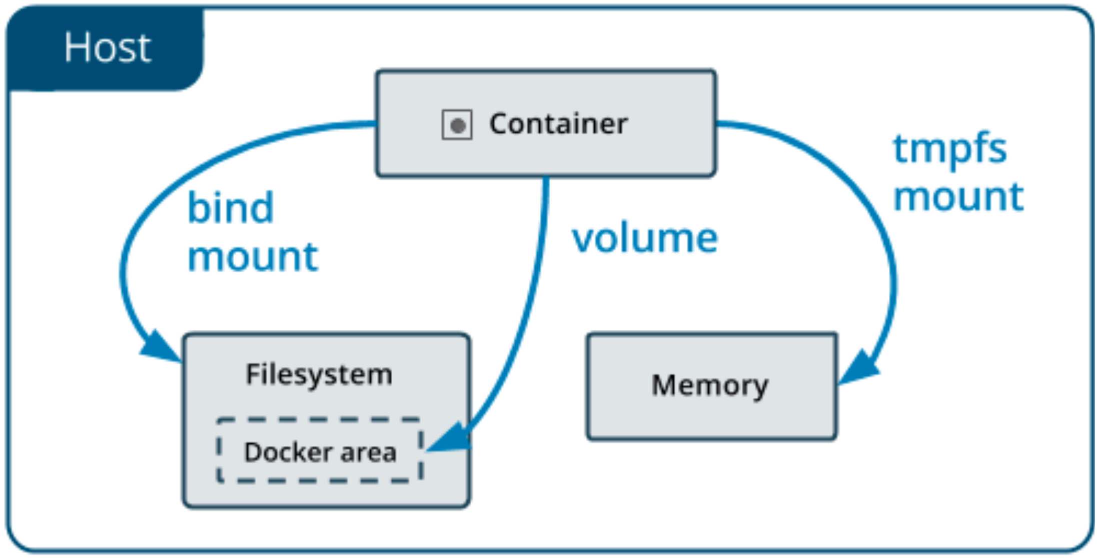
Non-Docker processes should not modify this part of the filesystem. Volumes are the best way to persist data in Docker.



DOCKER STORAGE: BIND VOLUMES

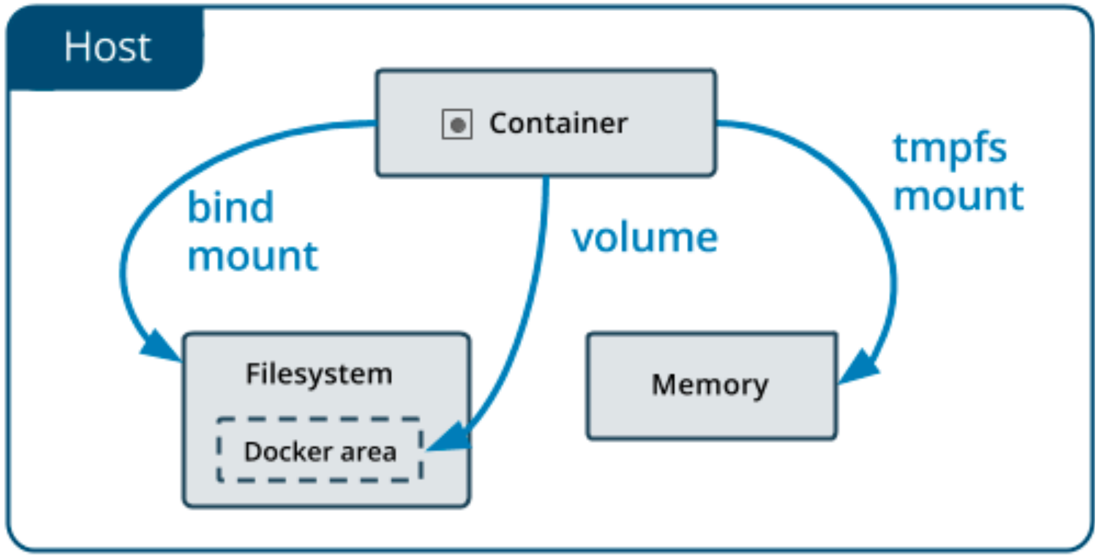
Bind mounts may be stored *anywhere* on the host system. They may even be important system files or directories.

Non-Docker processes on the Docker host or a Docker container can modify them at any time.



Docker storage: tmpfs

tmpfs mounts are stored in the host system's memory only, and are never written to the host system's filesystem.



OUTLINE

Containers

- Introduction
- Under the hood
  - Kernel namespaces
  - Cgroups
  - Copy-on-write File system
- Docker
- **Docker compose**
- Docker swarm
- Kuebernetes

DOCKER COMPOSE

Docker compose allows to define and run multi-container Docker applications.

The specification should be defined in a docker-compose.yml file.

Multi-container application started using:

```
$ docker-compose up -d
```

DOCKER COMPOSE: EXAMPLE

```
version: "3"
services:
  web:
    image: webserver
    depends_on:
      - "db"
    ports:
      - "8000:8000"
  db:
    image: postgres

networks:
  default:
    external:
      name: my-pre-existing-network
```

image : docker image to be used

depends_on : defines the order for starting up the services.

OUTLINE

Containers

- Introduction
- Under the hood
  - Kernel namespaces
  - Cgroups
  - Copy-on-write File system
- Docker
- Docker compose
- **Docker swarm**
- Kuebernetes

DOCKER SWARM

Swarm orchestrates a cluster of Docker instances.

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).

When creating 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.

DOCKER SWARM: NODES

A **node** is an instance of the Docker engine participating in the swarm. Typically, it consists of Docker nodes distributed across multiple physical and cloud machines.

The **manager node** receives service definitions and dispatches units of work called tasks to worker nodes

Manager nodes also perform the orchestration and cluster management functions required to maintain the swarm state.

Worker nodes receive and execute tasks dispatched from manager nodes.

A given Docker host can be a manager, a worker, or both.

Docker swarm: service

A **service** is the definition of the tasks to execute on the manager or worker nodes. A service is the primary root of user interaction with the swarm.

A service specifies which container image to use and which commands to execute.

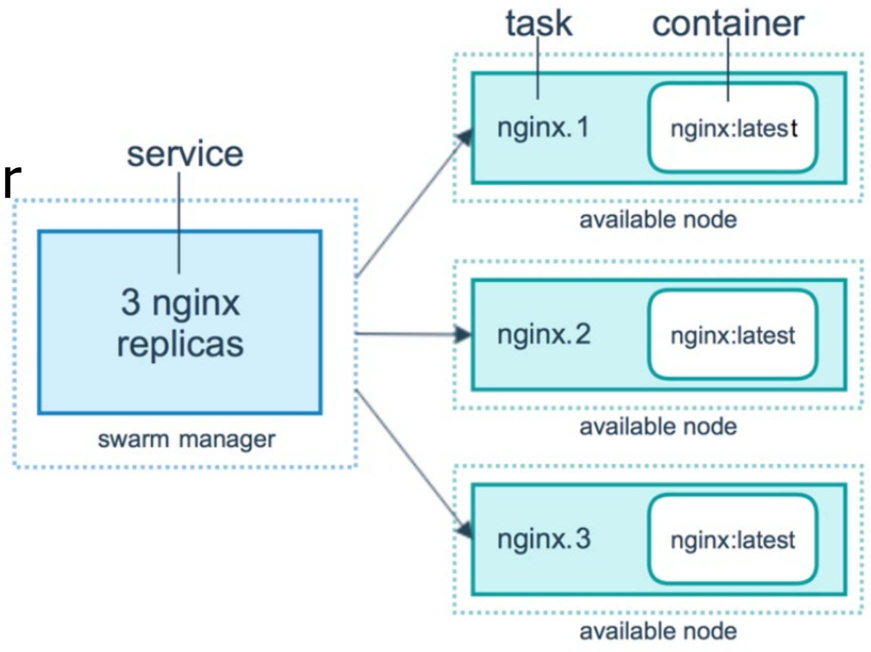
In the replicated services model, the swarm manager distributes a specific number of replica tasks among the nodes based on the scale set in the desired state.

For global services, the swarm runs one task for the service on every available node in the cluster.

DOCKER SWARM: TASK

A **task** is a running container which is part of a swarm service. It is the atomic scheduling unit of swarm.

- Manager nodes assign tasks to worker nodes according to the number of replicas set in the service scale.
- Once a task is assigned to a node, it cannot move to another node. It can only run on the assigned node or fail.



LOAD BALANCING

The swarm manager uses **ingress load balancing** to expose the services externally to the swarm. The swarm manager can automatically assign the service a **PublishedPort** or you can configure a PublishedPort for the service.

External components, such as cloud load balancers, can access the service on the PublishedPort of any node in the cluster. All nodes in the swarm route ingress connections to a running task instance.

Swarm mode 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 SERVICES VS. STANDALONE CONTAINERS

It is possible to modify a service’s configuration, including the networks and volumes it is connected to, without the need to manually restart the service.

Docker will update the configuration, stop the service tasks with the out of date configuration, and create new ones matching the desired configuration.

OUTLINE

Containers

- Introduction
- Under the hood
  - Kernel namespaces
  - Cgroups
  - Copy-on-write File system
- Docker
- Docker compose
- Docker swarm
- **Kubernetes**

KUBERNETES

Kubernetes is an open-source platform for automating deployment, scaling, and operations of application containers.

- Horizontal Scalability
- Self-healing
- Service Discovery
- Automated Rollbacks

From Google projects Borg and Omega.

KUBERNETES OBJECTS

Pod: encapsulates an application’s container (or multiple containers), storage resources, a unique network IP, and options that govern how the container(s) should run.

Service: a Service is an abstraction which defines a logical set of Pods and a policy by which to access them.

Volume: a volume is a directory which is accessible to the Containers in a Pod. A Kubernetes volume has the same lifetime of the Pod that encloses it.

Namespace: Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces.

KUBERNETES OBJECTS: POD

A ***Pod*** is the basic execution unit of a Kubernetes application – the smallest and simplest unit in the Kubernetes object model that can be created or deployed.

A Pod encapsulates an application's container (or multiple containers), storage resources, a unique network IP, and options that govern how the container(s) should run.

Docker is the most common container runtime used in a Kubernetes Pod, but Pods support other container runtimes.

KUBERNETES OBJECTS: POD (2)

Pods with a single container. In this case, a Pod is a wrapper around a single container. This is the most common Kubernetes use case.

Pods that run multiple containers that need to work together. A Pod might encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. Example: one container serving files from a shared volume to the public, while a separate "sidecar" container refreshes or updates those files.

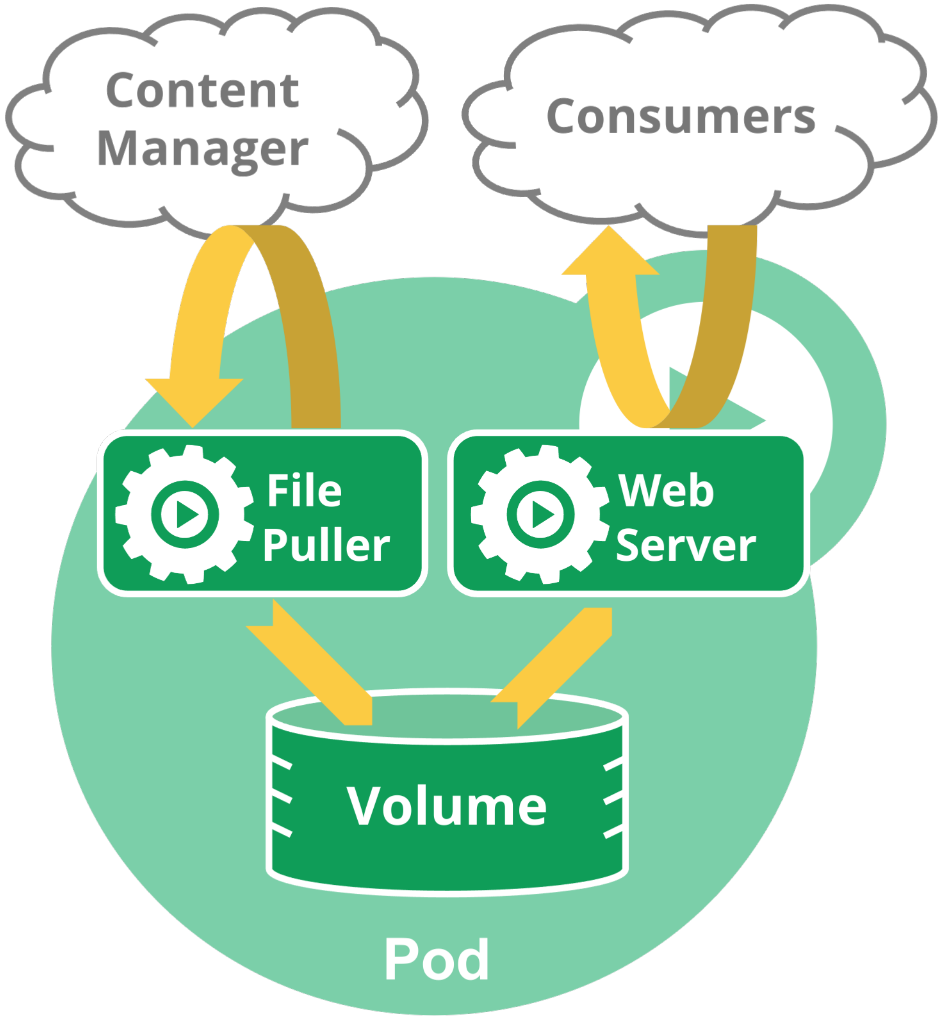
Each Pod runs a single instance of a given application. For scaling horizontally (e.g., run multiple instances), multiple Pods are used, one for each instance. **Replicated Pods** are usually managed as a group by an abstraction called a Controller.

INIT CONTAINERS

Some Pods have init containers as well as app containers.

Init containers run and complete before the app containers are started. If a init container fails, the Pod is reinitialized.

This can be used to make initializations before staring the main service.



Shared resources of a Pod

Networking

Each Pod is assigned a unique IP address. Every container in a Pod shares the network namespace, including the IP address and network ports. Containers inside a Pod communicate using localhost. When containers in a Pod communicate with entities outside the Pod, they must coordinate how they use the shared network resources (such as ports).

Storage

A Pod can specify a set of shared storage **Volumes**. All containers in the Pod can access the shared volumes, allowing those containers to share data. Volumes also allow persistent data in a Pod to survive in case one of the containers within needs to be restarted.

Pod template

```
apiVersion: batch/v1
kind: Job
metadata:
  name: hello
spec:
  template:     # This is the pod template
    spec:
      containers:
      - name: hello
        image: busybox
        command: ['sh', '-c', 'echo "Hello, Kubernetes!"
&& sleep 3600’]
      restartPolicy: OnFailure
    # The pod template ends here
```

Defines the name of the pod.

Defines what are the containers running in this pod.

Kubernetes objects: Services

Pods can fail and be restarted by the Kubernetes system.

For a given Deployment, the set of Pods running in one moment in time can be different from the set of Pods running that application a moment later.

This leads to a problem: if some set of Pods (call them “backends”) provides functionality to other Pods (call them “frontends”) inside your cluster, how do the frontends find out and keep track of which IP address to connect to, so that the frontend can use the backend part of the workload?

KUBERNETES OBJECTS: SERVICES (2)

A **Service** is an abstraction which defines a logical set of Pods and a policy by which to access them.

For example, consider a stateless image-processing backend which is running with 3 replicas. Frontends do not care which backend they use and whether they change.

The Service abstraction enables this decoupling.

Kubernetes APIs for service discovery allows to query Endpoints, that get updated whenever the set of Pods in a Service changes.

For non-native applications, **Kubernetes offers ways to place a network port or load balancer** in between your application and the backend Pods.

DEFINING A SERVICE

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - name: http
      protocol: TCP
      port: 80
    - name: https
      protocol: TCP
      port: 443
```

Defines a service that will be implemented by pods with the name MyApp.

Ports of the services. Outside applications will access the service using these ports.

SERVICE TYPE

ClusterIP: Exposes the Service on a cluster-internal IP. This is the default ServiceType.

NodePort: Exposes the Service on each Node’s IP at a static port (the NodePort). A ClusterIP Service, to which the NodePort Service routes, is automatically created.

LoadBalancer: Exposes the Service externally using a cloud provider’s load balancer. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created.

ExternalName: Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com).

KUBERNETES OBJECTS: VOLUMES

On-disk files in a Container are ephemeral. This leads to problems for applications that are not stateless.

First, when a Container crashes, it will be restarted, but the files will be lost - the Container starts with a clean state.

Second, when running Containers together in a Pod it is often necessary to share files between those Containers.

The Kubernetes **Volume** abstraction solves both of these problems.

KUBERNETES OBJECTS: VOLUMES (2)

A **volume** is a directory which is accessible to the Containers in a Pod.

A Kubernetes volume has the **same lifetime of the Pod that encloses it**. Consequently, a volume outlives any Containers that run within the Pod, and data is preserved across Container restarts.

When a Pod ceases to exist, the volume will cease to exist, too.

Kubernetes supports many types of volumes, and a Pod can use any number of them simultaneously. Some volumes are specific to a given cloud and some are persistent (in the sense that they outline a Pod).

KUBERNETES OBJECTS: NAMESPACES

Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces.

Namespaces are intended for use in environments with many users spread across multiple teams, or projects.

Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces. Namespaces can not be nested inside one another and each Kubernetes resource can only be in one namespace.

KUBERNETES CONTROLLERS

Deployments: A Deployment provides declarative updates for Pods and ReplicaSets.

ReplicaSet: A ReplicaSet’s purpose is to maintain a stable set of replica Pods running at any given time.

DaemonSet: A *DaemonSet* ensures that all (or some) Nodes run a copy of a Pod. As nodes are added/removed to the cluster, Pods are added to/deleted from them.

StatefulSet: StatefulSet is the workload API object used to manage stateful applications.

Job: A Job creates one or more Pods and ensures that a specified number of them successfully terminate

DEPLOYMENT

A ***Deployment*** provides declarative updates for Pods and ReplicaSets.

A *Deployment* describes the *desired state* of the system. The Deployment Controller changes the actual state to the desired state at a controlled rate.

DEPLOYMENT EXAMPLE

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Defines a deployment with three replicas of a Pod with name nginx.

Definition of the “nginx” Pod – includes a nginx container.

ReplicaSet

A ***ReplicaSet*** ensures that a specified number of pod replicas are running at any given time. A Deployment manages ReplicaSets.

DaemonSet

A ***DaemonSet*** ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.

Some typical uses of a DaemonSet are:

- running a cluster storage daemon, such as glusterd, ceph, on each node.
- running a logs collection daemon on every node, such as fluentd or logstash.
- running a node monitoring daemon on every node.

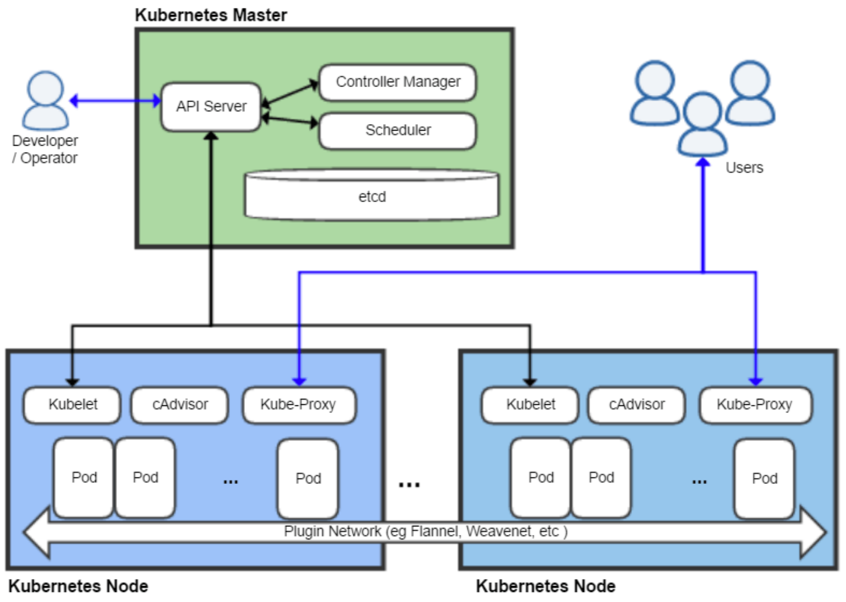
StatefulSets

StatefulSet is used to manage stateful applications, and is useful for applications that require one or more of the following:

- Persistent, unique network identifiers.
- Persistent, persistent storage.
- Ordered, graceful deployment and scaling.
- Ordered, automated rolling updates.

KUBERNETES CONTROL PLANE

The ***Kubernetes Control Plane*** makes the cluster's current state match the desired state, by performing a variety of tasks automatically – such as starting or restarting containers, scaling the number of replicas of a given application, and more.

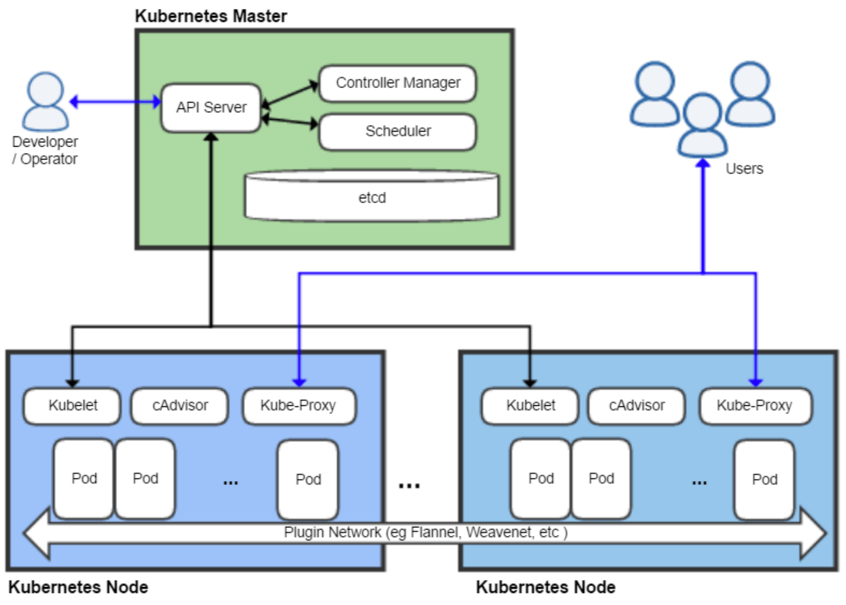


Kubernetes Control Plane (2)

The **Kubernetes Master** runs on a single node of a cluster, which is designated as the master node.

A **Kubernetes Node** runs:

- kubelet, which communicates with the Kubernetes Master.
- kube-proxy, a network proxy which reflects Kubernetes networking services on each node.



KUBERNETES VS. SWARM

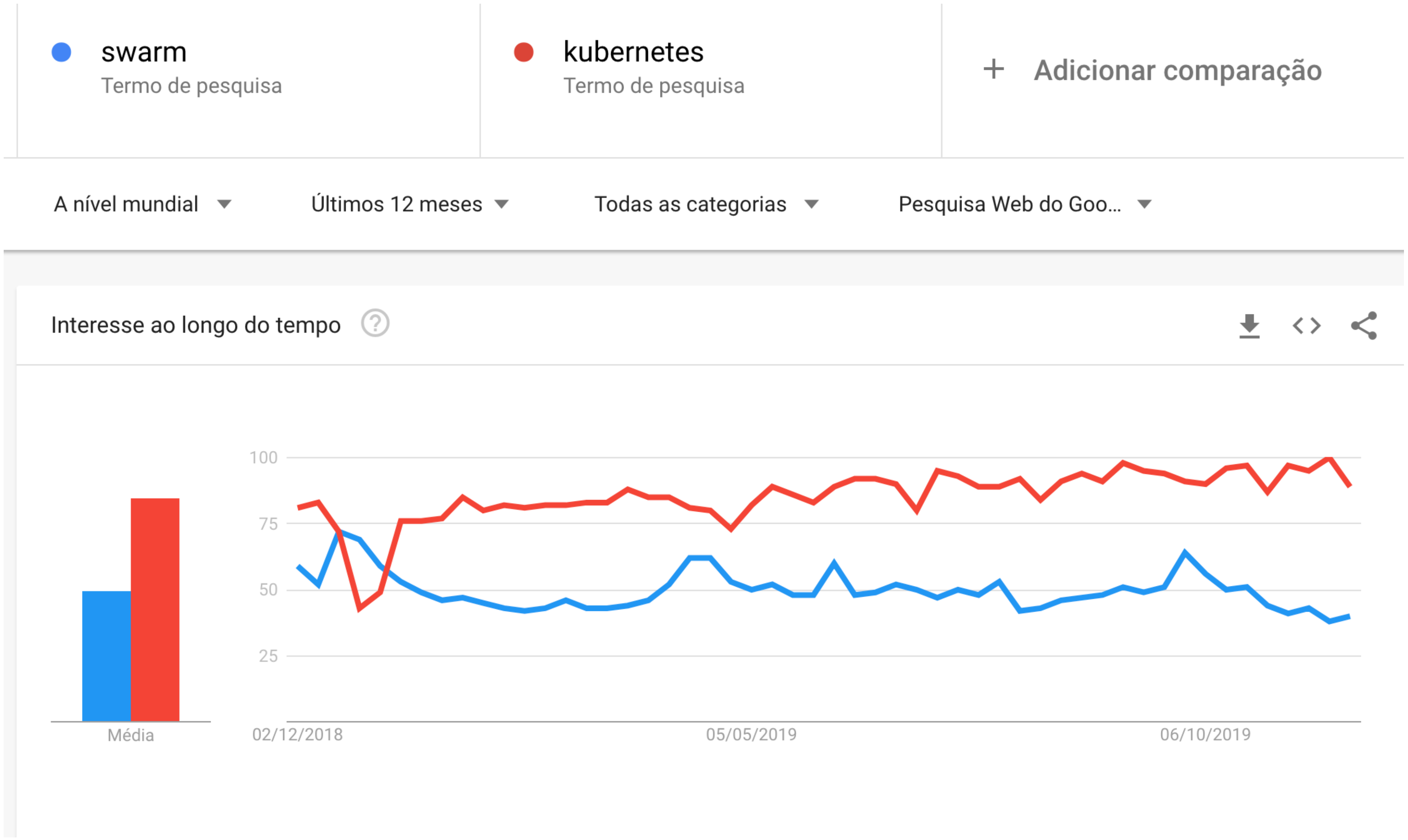
Kubernetes is more sophisticated than Swarm.

Swarm works only with Docker, Kubernetes can work with other container services also.

Kubernetes is more complex to deploy and manage compared to Swarm.

Kubernetes is reported to have better scalability.

KUBERNETES VS. SWARM: USAGE? (SOME INDICATIONS FROM GOOGLE TRENDS)


● swarm
Termo de pesquisa

● kubernetes
Termo de pesquisa

+ Adicionar comparação

A nível mundial ▾ Últimos 12 meses ▾ Todas as categorias ▾ Pesquisa Web do Goo... ▾

Interesse ao longo do tempo

OUTLINE

…

Container @ Azure

CONTAINERS @ AZURE

Azure has a comprehensive offer related to containers.

IF YOU WANT TO...	USE THIS
Simplify the deployment, management, and operations of Kubernetes	Azure Kubernetes Service (AKS)
Quickly create powerful cloud apps for web and mobile	App Service
Easily run containers on Azure without managing servers	Container Instances
Cloud-scale job scheduling and compute management	Batch
Develop microservices and orchestrate containers on Windows or Linux	Service Fabric
Store and manage container images across all types of Azure deployments	Container Registry

TO KNOW MORE

Docker: Lightweight Linux Containers for Consistent Development and Deployment. Petros Koutoupis. Linux Journal 2015.

https://www.linuxjournal.com/content/everything-you-need-know-about-containers-part-iii-orchestration-kubernetes

Containers and Docker

https://docs.docker.com/engine/docker-overview/

Kubernetes

https://kubernetes.io/docs/concepts/#kubernetes-objects

https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/

https://kubernetes.io/docs/concepts/services-networking/service/

https://kubernetes.io/docs/concepts/storage/volumes/

https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

ACKNOWLEDGMENTS

Some text and images from Docker, Kubernetes and Microsoft online documentation.

Some slides based on a previous version by Paulo Lopes and Vitor Duarte.