Universitat de Lleida

Docker and Kubernetes

Distributed Computing - Grau en Enginyeria Informàtica

Artur Cullerés Cervera
Roger Fontova Torres

October 30, 2022

Contents

List of Figures

1 Docker

1.1 What is Docker

Docker is a software platform that enables you to build, run, deploy and manage your applications grouping the source code, libraries and dependences required to run it in any environment (Linux, MacOS, Windows, etc.).

This grouping facilitates the delivery of distributed applications since it is easier to manage the different parts of the application thanks to containers, which will be explained in section 1.3.

1.2 Docker or Virtual Machines?

Some readers might be thinking that Docker is a kind of a Virtual Machine, however, that statement is incorrect. They have quite a different architecture and purposes.

1.2.1 Virtual Machine architecture

Virtual Machines allows your current computer to simulate an entire OS as a different environment.

As we can see on the figure 1, each Virtual Machine run a full independent OS that communicates with the host OS and hardware through the Hypervisor. In addition, the startup and shutdown of a Virtual Machine can be slow if your purpose is only to run, deploy or test your application.

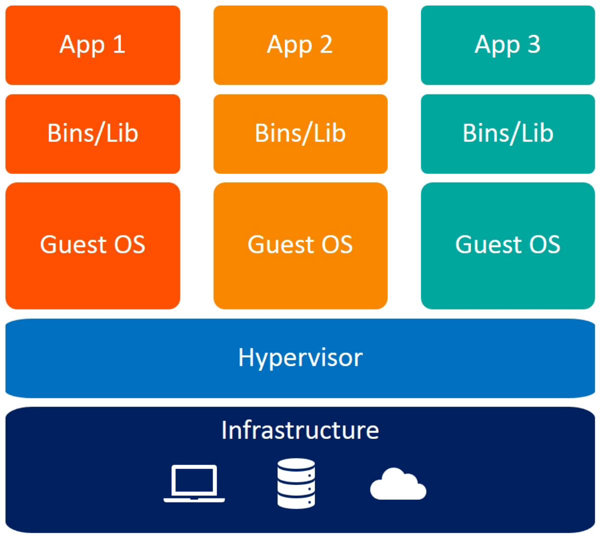

Figure 1: Virtual Machine architecture

1.2.2 Docker architecture

Docker architecture is based in client-server architecture, where the client talks with the server through an API to manage life cycle of the containers.

A container is only a set of libraries and software required to run the application.

As Docker uses the host operating system, the instructions are not required to be traduced, they are compatibles with the operating system. So, if we are using a linux operating system, all those containers executed on docker engine, wouldn't be able to be ran at other operating systems.

The main elements of the Docker architecture are the Docker Engine and the Docker Register.

The Docker Engine (or Container Engine, as we can see on figure 2) is the client-server application implemented by Docker, and is composed by three components:

- **Server**: It is the main Docker process. The responsible of managing all the objects kept at Docker, like images, containers, network, etc. which we are going to talk about in section 1.3.

- **REST API** [1]**:** It allows us accessing to network capabilities and execute commands on it.
- **Client:** It talks through the API Rest for executing the commands. Is what are we going to use for manage life cycle of our images and containers.

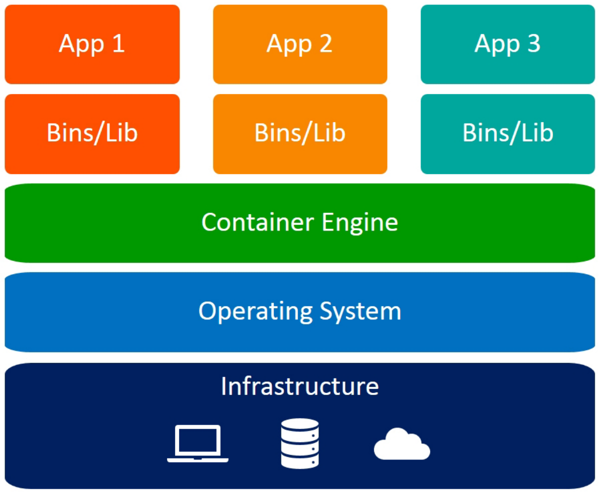

Figure 2: Docker architecture

[1]To see more about REST API's visit the following webpage: https://www.redhat.com/en/topics/api/what-is-a-rest-api

1.2.3 Benefits and drawbacks

Virtual Machine	Docker
Each Virtual Machine runs its own OS	All containers share the ame Kernel of the host
Slow startup and shutdown (can be even minutes)	Very fast startup and shutdown (seconds, and even tenths of a second if is a small container)
Cannot run a lot of Virtual Machines in an average PC without performance issues	Can run many containers in an average PC.
Fully isolated, hence more secure	Process-level isolation, hence can be less secure

Table 1: Differences between Virtual Machine and Docker

1.3 How does Docker works?

In section 1.2.2 we have mentioned images, containers, networks, etc. but what are they. Hereinafter in this section we will explain how do they work and how to create, manage and run them.

A Docker image is a file that contains our code and the needed libraries that we have specified in the Dockerfile. This image will be executed in a container.

In order to create an image file we have to execute the Dockerfile, which will execute the commands that we have specified on it. The following Dockerfile is a simple example to copy a text file into an image.

```
FROM ubuntu
COPY file-outside-docker.txt file-inside-docker.txt
RUN echo text inside file > file-inside-docker.txt
CMD ["cat", "file-inside-docker.txt"]
```

In order to build the Dockerfile we will run the following command:

```
docker build [image-name] .
```

Once it is created, we run it in a container with the following command:

```
docker run --name [container-name] [image-name]
```

A container is an instance of an image that isolates it from the environment that is being executed.

What if we wanted to run many docker images at once? What we should do is instead of running the previous commands one by one, we should use docker-compose.

This allows us to run many docker containers with the specifications that we want.

Also, it is useful since all the containers created with the same docker compose, will be on a same virtual network, so they will be able to communicate between them.

The following yaml file specifies what docker-compose has to do:

```
version: "3.9"
services:
  web:
    build: .
    ports:
      - "8000:5000"
    volumes:
      - .:/code
  redis:
    image: "redis:alpine"
```

In order to execute a docker-compose file we will execute the following command:

```
docker-compose up
```

This piece of code will run two containers named web and redis:

- **web:** This service will build a docker file located on the same directory, will bind the 8000 host port to the 5000 container port and will copy the content from the actual directory to /code inside the container. Also, once you shutdown the container, everything saved in the /code, will also be saved on the actual directory.
- **redis:** This service will run a container with the image redis:alpine.

As we said before, all containers that are run by the same docker-compose file, will be on the same virtual network, this means that we can ping from web to redis. But that's not everything. Also, thanks to the magic [2] of Docker, you can use the container instead of using the container IP.

1.4 Docker Alternatives

All technologies have disadvantages and Docker is not an exception. There are different tools for the containers creation and managing:

- **RKT**: One of the biggest Docker competitors. It is harder to install, but in security terms, RKT is better than Docker.

- **PODMAN**: As RKT it allows running containers without the need of being root user. It also works awesome with Kubernetes, and is substituting Docker in this field.

- **SINGULARITY**: Is perfect for shared workspaces because of its isolation level. We can import and use our images from Docker that we have already used. SINGULARITY containers are made for being the most portable possible.

- **OPENVZ**: Available for Linux operating system, and it allows us to execute each container as a unique server. The installation and usage are harder than Docker. One disadvantage is that there is very little information about it on internet.

[2]To know more about that, visit https://docs.docker.com/config/containers/container-networking/

2 Kubernetes

2.1 What is Kubernetes?

Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications.

Is in steady growth, its tools, support and services are widely available to all people.

Kubernetes offers a container-centric management environment. It orchestrates, computes and storage distributed applications so users don't have to.

It is not a traditional PaaS system. Since it operates with containers and not directoy with hardware, it provides some applicable features common to PaaS (deployment, scaling, load balancing, etc).

2.2 Why do we need it?

When we have a high production context, we should be able to organize and manage all those container we have created so they auto-scale and manage possible bugs or erros by themselves.

Thanks to Kubernetes, it's possible to solve problems as inactivity of the services, high yield scalability, the possibility to easily backup a server if it fails, etc.

Using kubernetes for a monolithic application with a static user base is useless since the potential kubernetes offers will not be exploided.

2.3 Kubernetes architecture

Kubernetes has to big blocks: master and workers (also known as nodes).

The master block is composed of:

- **API server:** Exposes an interface so different clients can interactuate with kubernetes.
- **Controller manager:** Manages the running containers and those that should be running.
- **Scheduler:** Receives and executes the controller manager orders.
- **etcd:** Data base that store all the information and configurations of the cluster

Each node is composed of:

- **Kubelet:** Is an agent
- **Pod:** Is a set of containers (usually one), under the same IP. Those pods are volatile, so they can be destroyed and recreated any time.

Pods in different nodes can communicate with each other, however, each time a pod is destroyed and recreated, it's IP changes. This makes it difficult for a pod to know another pod IP. This problem is solved with something called "services". Some types of them are Cluster IP, Load Balancer, Ingress, Node Port, etc.

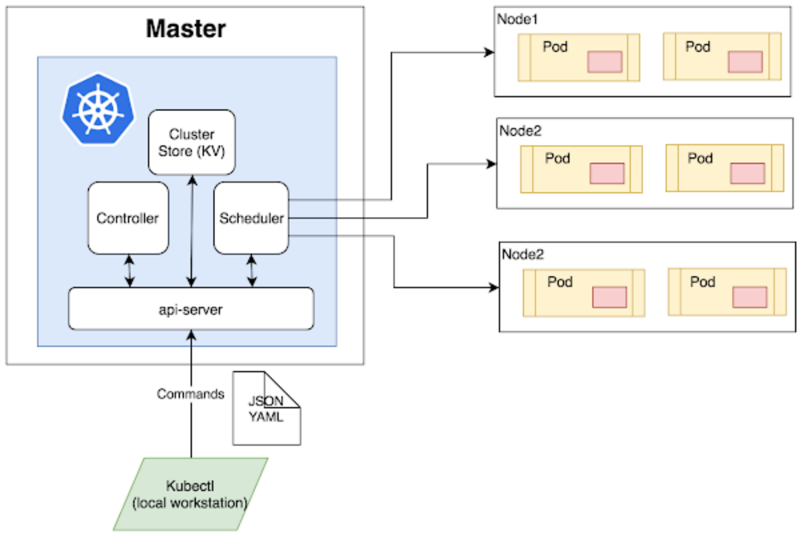

Figure 3: Kubernetes Master-Worker architecture

3 Practical example of docker

In this section we are going to explain briefly how to run an example web page we made locally with docker.[3] The frontend block is written with Angular 2, the backend with Springboot, and finally, the database will be a MySQL database.

Our intention with this example is to understand better how to use docker, not only the theoretical part.

First of all, we created a Dockerfile for the frontend and a Dockerfile for the backend.

In the backend Dockerfile we specify to build the image above one that already has openjdk11, so we dont need to run a command and set up everything. Then, we specify to create a folder inside the new image where we will copy a the .jar (the compiled file). Once we run the image, it will execute the java command to run a .jar file.

[3]If you want to see the code, visit our github repository

In the frontend Dockerfile we specify to build the image avobe one that already has node installed. In this case, it will create a new folder where all the files from our angular app will be copied at. Next we will install the needed dependencies to run the Angular app and finally build the project. Once we run the image, it will execute the Angular command to run the server.

For the database we dont need a Dockerfile, since we can use the images that we can find on DockerHub[4].

Once we have all the Dockerfiles, we can proceed to create de docker-compose.yaml.

In this file, all the services to execute are specified, also, it is specified the order of execution. And the ports that we want to bind for each container.

To execute this file, we must the command ”docker compose up”, and suddenly all 3 images will run in a container and we will be able to access the webpage on localhost:4200.

4 Bibliography

- https://www.docker.com/
- https://guiadev.com/top-5-alternativas-a-docker/
- https://kubernetes.io/
- https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/
- https://thenewstack.io/do-i-really-need-kubernetes/

[4]Visit the following link to see the MySQL images: https://hub.docker.com/search?q=mysql