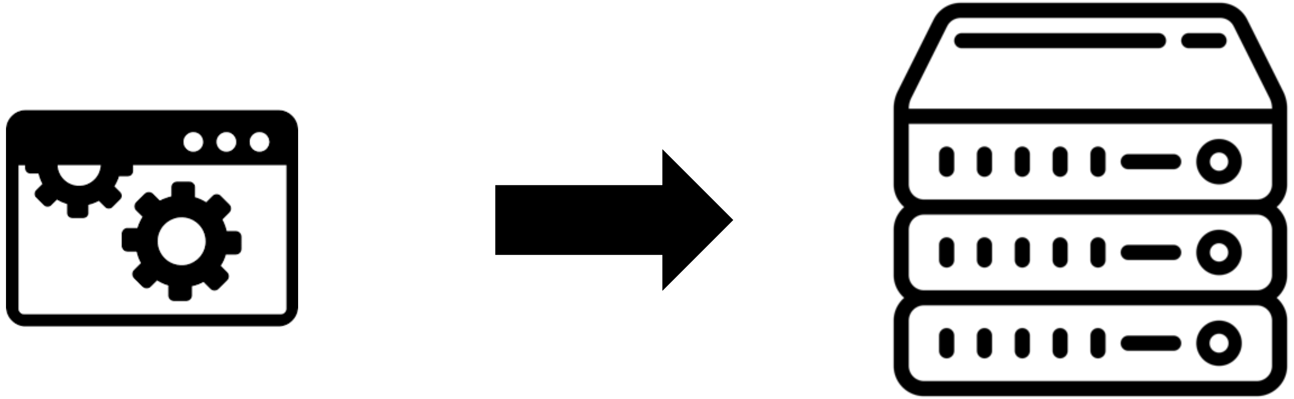
Tecnologie e applicazioni web

Docker containers

Filippo Bergamasco (filippo.bergamasco@unive.it)
http://www.dais.unive.it/~bergamasco/
DAIS - Università Ca'Foscari di Venezia
Academic year: 2023/2024

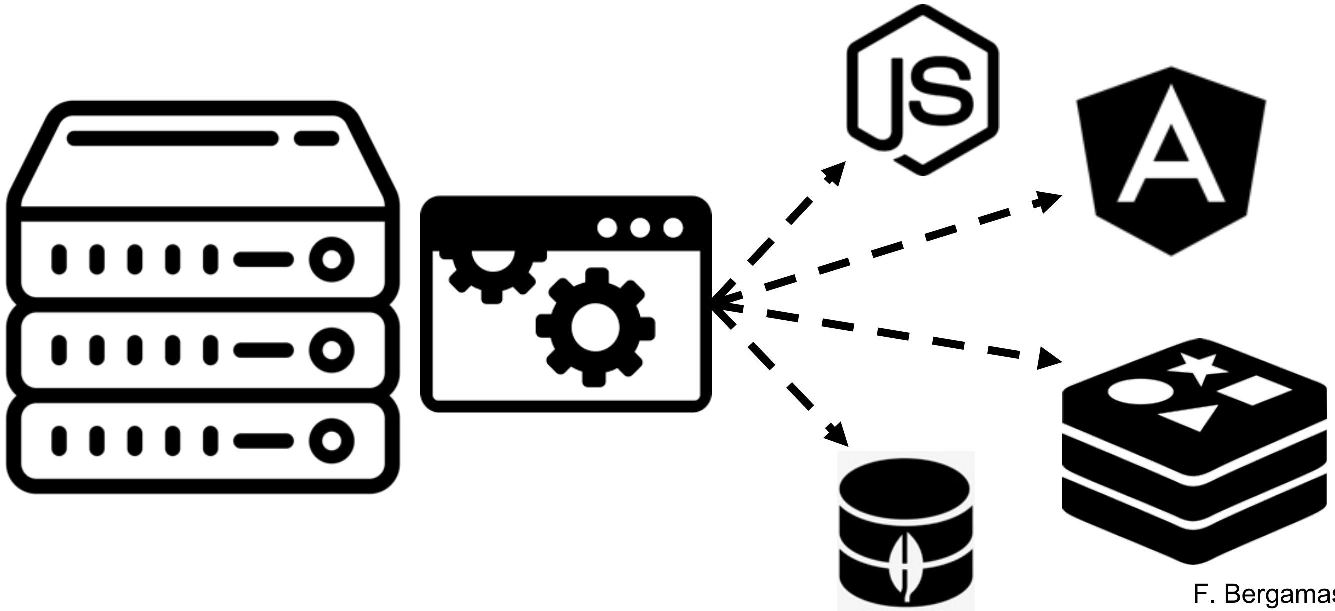
The problem

A team of developers creates a new application. Once completed, they install it on the production servers and get it running



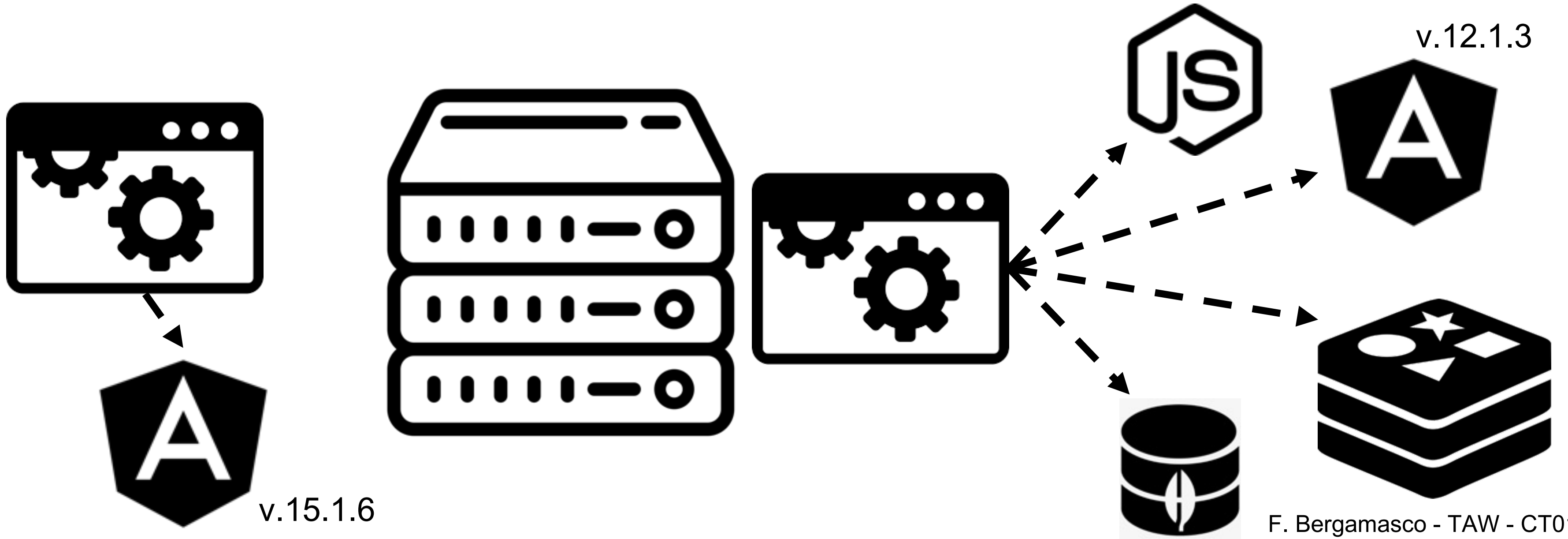
The problem

However, a modern application requires several external dependencies: frameworks, libraries, etc.



The problem

When a new application must be installed, there can be conflicts with some external dependencies



The problem

Sometimes, installing a new application (or a new version of an existing application) becomes a complex project on its own. It can take months of planning and testing

Besides, the release cycles of modern software become faster and faster. How to deal with it?

A first solution: use VMs

Instead of running multiple applications on the same server, package a single application on each VM so they can run without conflicts on the same server.

Drawback: VMs contain a full-blown operating system just for a single application.

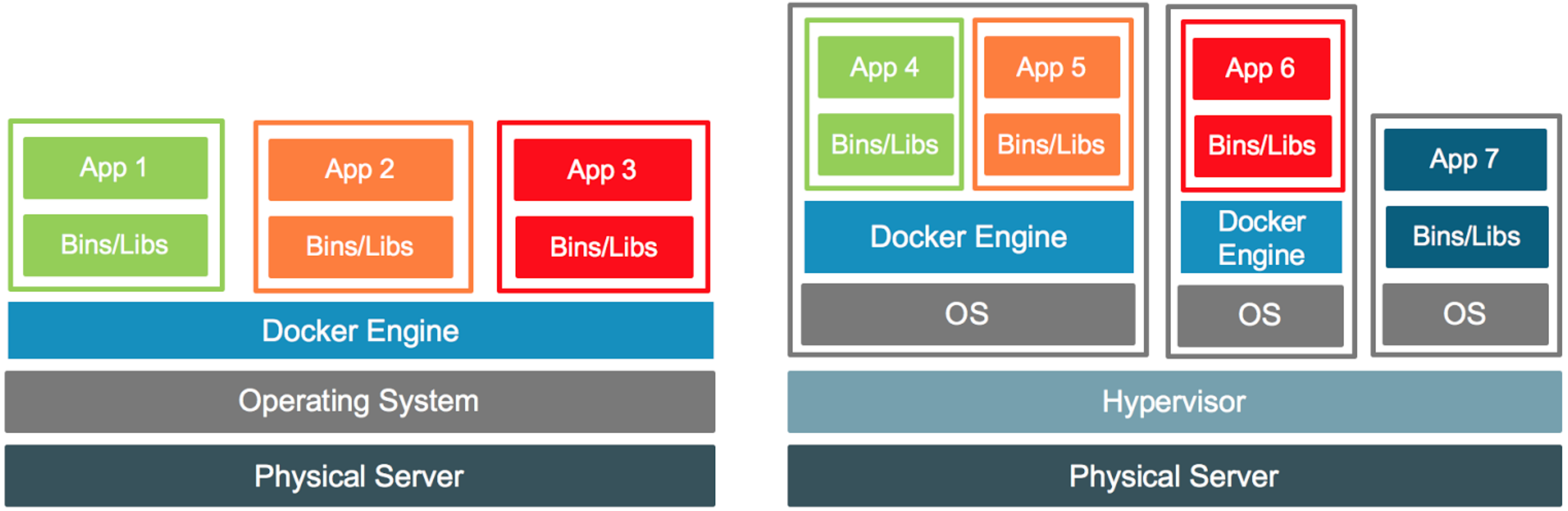
Lots of resources wasted!

A better solution: containers

A container is a “lightweight VM” that can run processes in a sandbox **completely isolated** from all other processes of the host machine in which the container is running.

Unlike VMs, containers share the same OS kernel with the host!

Containers vs. VMs



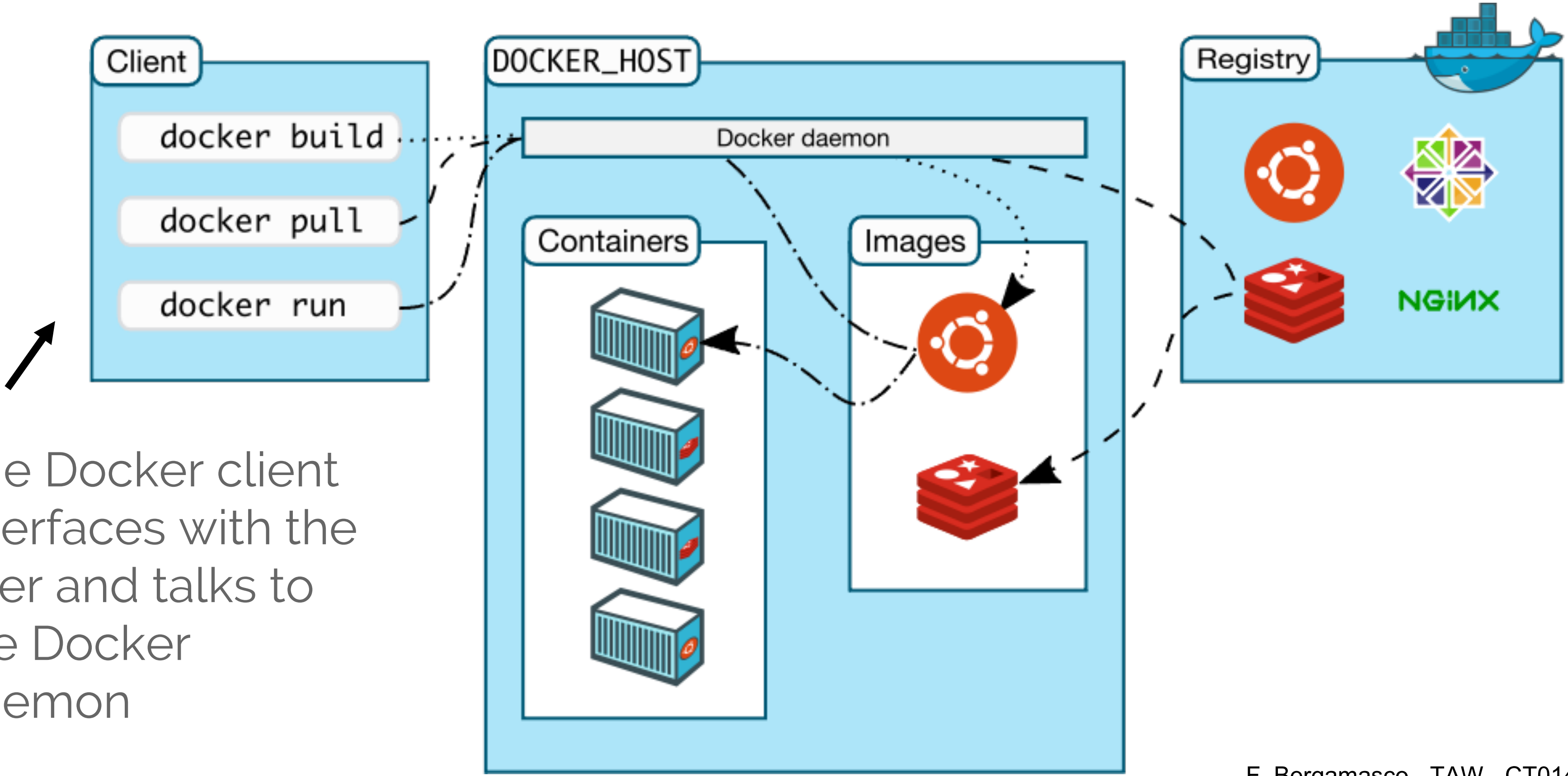
Containers: advantages

- **Simplified deployments:** a complex application can be packaged in a single component that can be executed without worrying about configuring the execution environment
- **Fast startup:** since the OS is not fully virtualized, the startup time is reduced

Containers: advantages

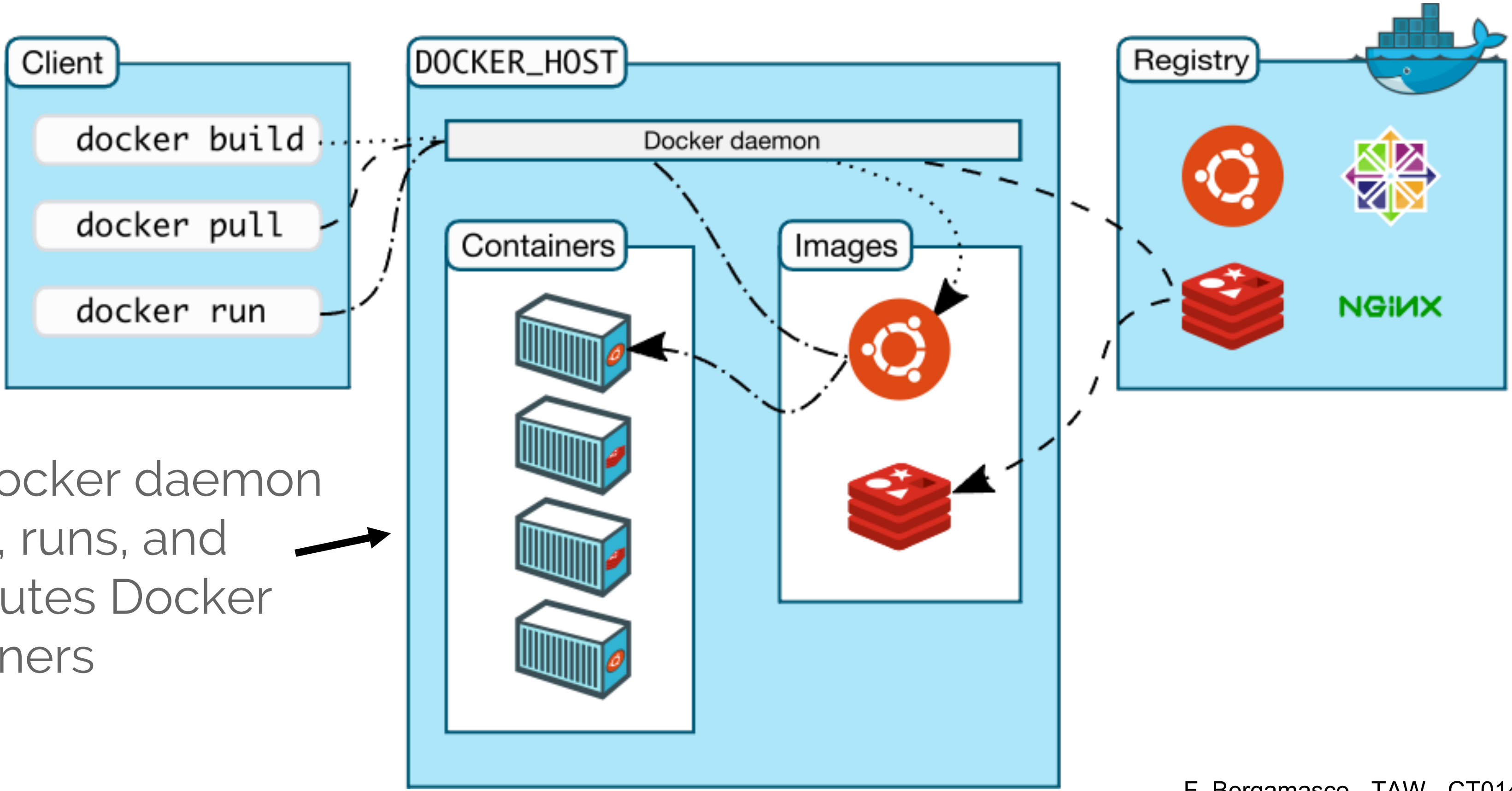
- **Portability:** a container can be created and configured on a machine and then executed on a production server without any modification
- **Fine-grained control:** A container can provide a single component to other containers, allowing the organization of an application in microservices.
- **Scalability:** containers can be executed and terminated quickly in accordance with the required workload

Docker architecture



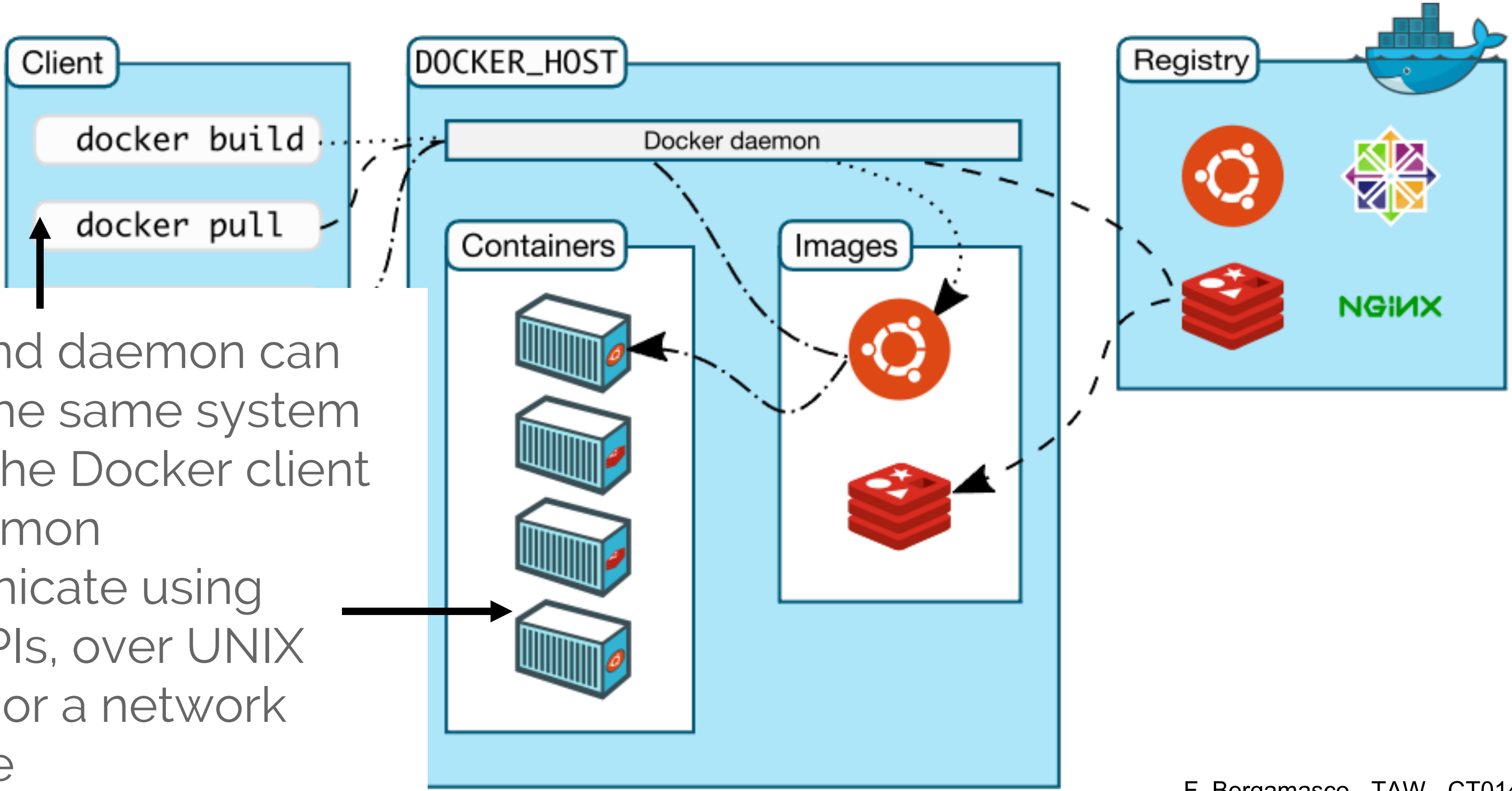
The Docker client interfaces with the user and talks to the Docker daemon

Docker architecture



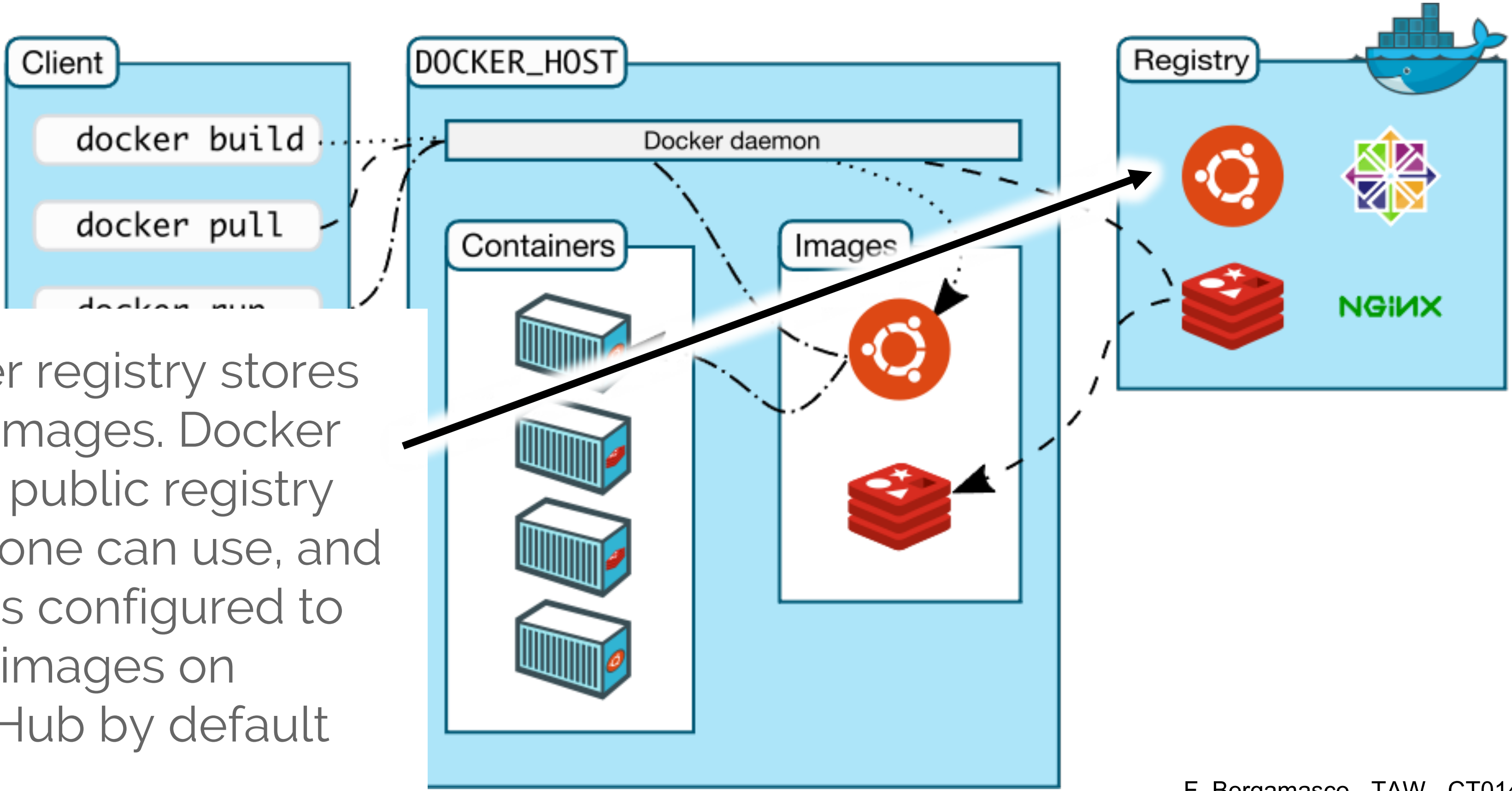
The Docker daemon builds, runs, and distributes Docker containers

Docker architecture



Client and daemon can run on the same system or not. The Docker client and daemon communicate using REST APIs, over UNIX sockets or a network interface

Docker architecture



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

Containers and Images

An **image** is a read-only template that is used to create a Docker container. An image is based on another image with some additional customization.

A file named *Dockerfile* defines a new image. It contains the instructions needed to create an image and run it as a container.

An image is like a **class** in OOP

Containers and Images

A **container** is a runnable instance of an image. Using the docker client, you can create, start, stop, move, or delete a container.

A running container can be connected to one or more networks, attached to storage, or used to create a new image based on its current state.

A container is like an **object** in OOP

Installing Docker

The easiest way for a developer to start working with Docker is to install Docker Desktop:

https://www.docker.com/products/docker-desktop/

Note: it will install a Linux VM to provide a consistent configuration across different platforms. It is meant for developers. In production servers, Docker EE is typically used.

Running your first container

```
$ docker run hello-world
```

It will run a container created from the image *hello-world:latest* available in the Docker HUB.

The process is executed, and its output is shown in the terminal. After the execution, the container changes its status from “Up” to “Exited”

List containers

```
$ docker container ls -a
```

It shows all the containers, along with their associated status, ID, name, and the image from which they were created.

The –q option is used to list just the ID. Useful to remove all the containers:

```
$ docker container rm –f $(docker
container ls –a –q)
```

Run a container in background

We typically run containers in background so they are detached from the terminal.

```
$ docker run -it -d --name myubuntu
ubuntu /bin/bash
```

When running, a container can be stopped and started again with the command

```
$ docker container start/stop <name or ID>
```

Exec into a running container

It is possible to run another process into an already-running container:

```
$ docker container exec –it <name>
<command>
```

It is useful to debug what’s happening inside it.

Single-host networking

Containers can communicate with the host by mapping internal ports (in the container) to host ports.

Example:

```
$ docker run -d --name nginx -p 8080:80 nginx:alpine
```

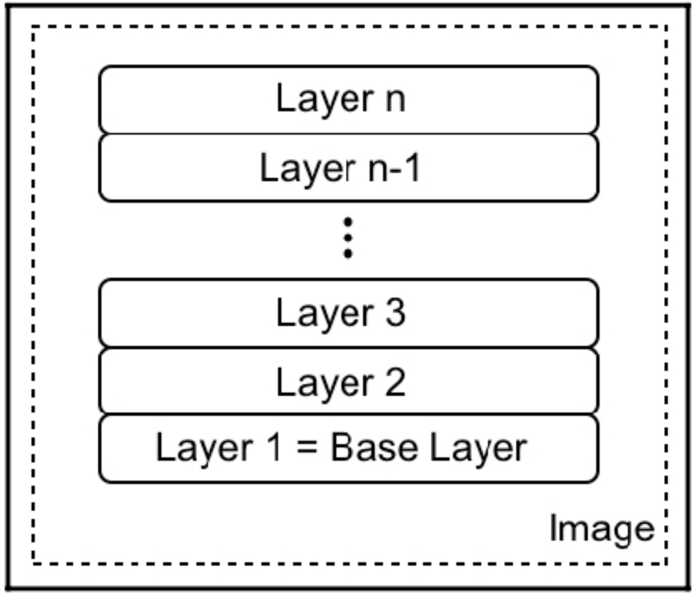
Runs an instance of the nginx web server, mapping port 80 of the container to 8080 of the host.

Images and the layered filesystem

The layered filesystem

At a lower level, a docker image is just a big tarball containing a **layered filesystem:**

- Each layer contains files and directories.
- Each layer contains the changes to the filesystem with respect to the underlying layers
- When instantiated in a container, the resulting filesystem is the union of all the layers

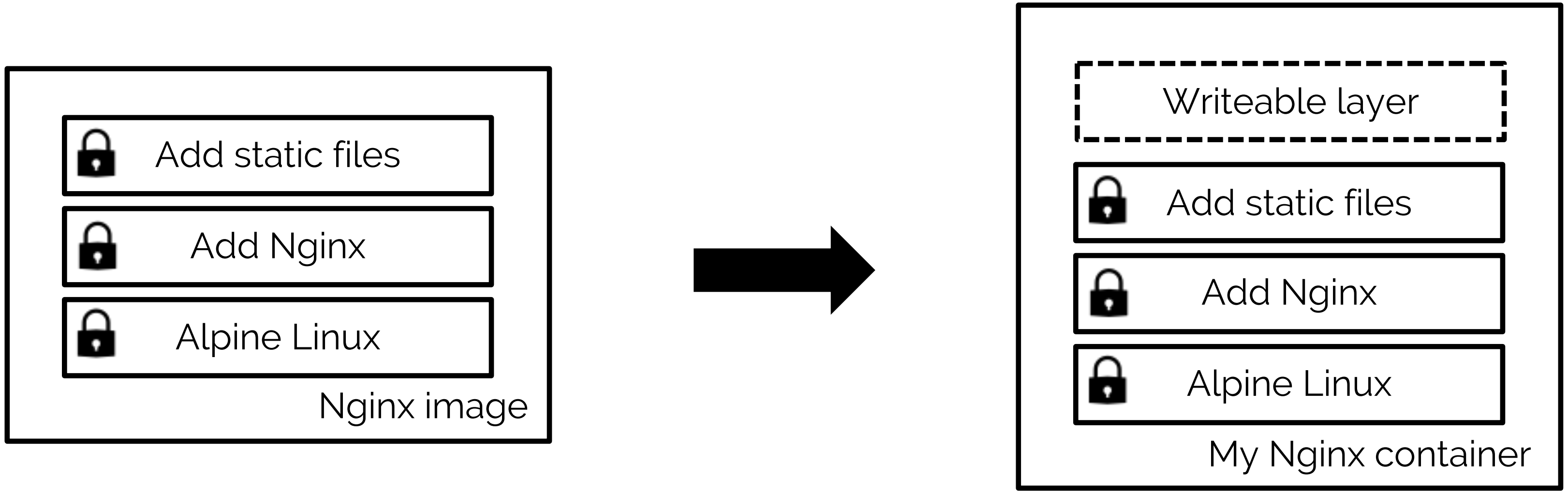


The layered filesystem

Layers of a container image are all **immutable** (read only). Once generated, each layer can only be deleted but not changed.

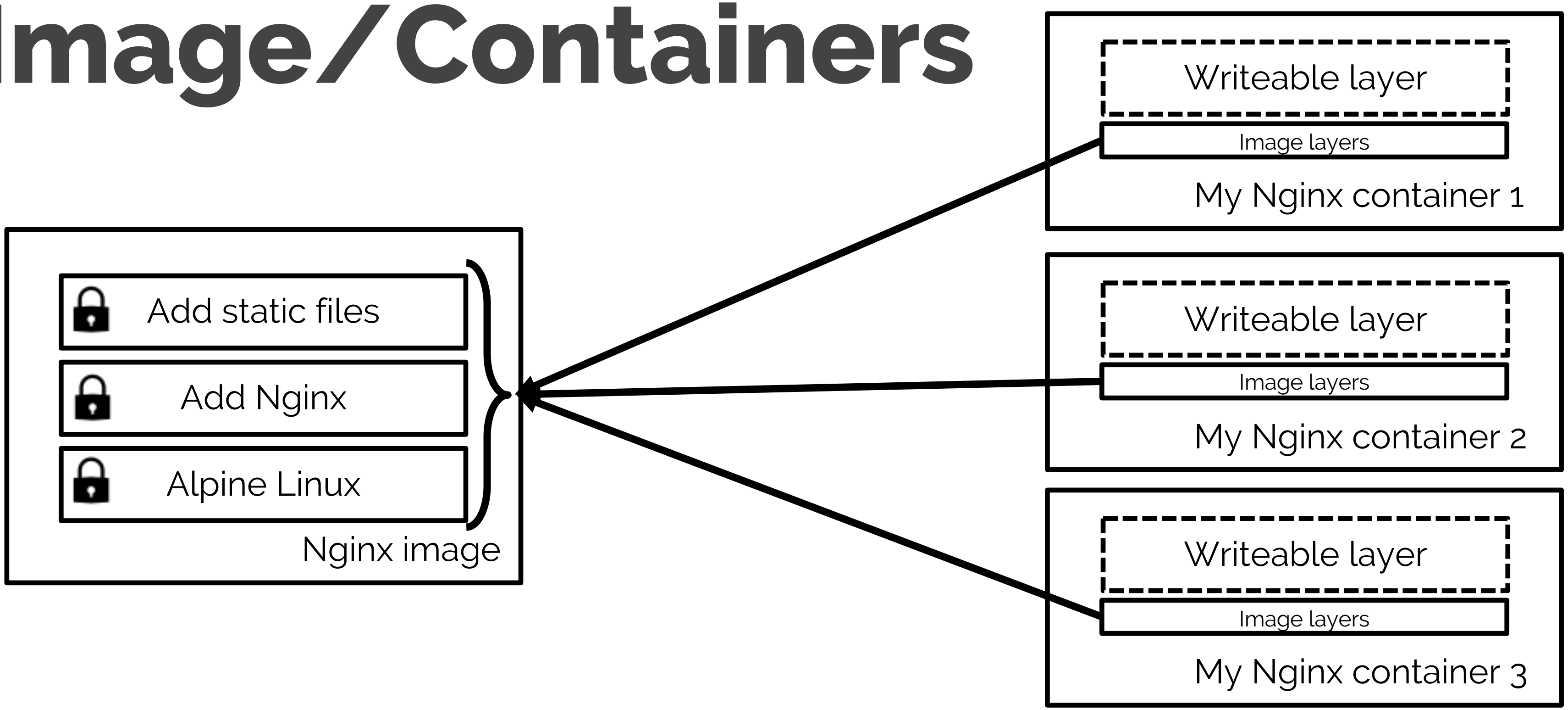
When a container is created from an image, a single writeable layer is added to the stack of immutable layers of the image.

Image/Containers



Since layers are immutable, they can be cached efficiently to maximize the performances

Image/Containers



All the containers started from the same image share the image layers. That’s how containers are so lightweight

Copy-on-write

- If a layer reads a file or folder that is available in one of the low-lying layers, then it just reads it
- If a layer wants to modify a file available in one of the low-lying layers, it first copies this file up to the target layer and then modifies it.

In a container, the top writeable layers will contain all the files modified during its execution

Container diff/commit

To see what was changed (with respect to the base image) during the execution of a container, we can use the command

```
$ docker container diff <container>
```

Such changes can be made permanent by creating a new image

```
$ docker container commit <container> <image>
```

The writeable layer of the container becomes a new readonly layer on top of the image stack

Volumes

Accessing a container fs

How to copy files in and out a container?

```
$ docker cp <container>:SRC DST
$ docker cp SRC <container>:DST
```

Notes: container paths are always relative to /

Host paths are relative to the current directory

It is not possible to copy files from an image. A container must be instantiated from an image first

Sharing/Persist data

How do you share/persist data among different containers?

Two options:

1. Using **volumes** (better performance, container only)
2. Using **bind-mount** (lower performance, mount host fs in a container)

Volumes

```
$ docker volume create myvolume
```

Mount a volume when starting a container:

```
$ docker run -v myvolume:<mountpoint>
<image>
```

Note: Mountpoint is relative to container root

Bind-mounts

When using a bind mount, a file or directory on the host machine is mounted into a container:

```
$ docker run –v
<host_path>:<container_path> <image>
```

When bind-mounting a directory into a non-empty directory on the container, the directory’s existing contents are obscured by the bind-mount

Creating and managing images

Interactive image creation

One way to create a new image consists of the following:

- Instantiate a container from a base image (like the official Ubuntu image)
- Install everything necessary, apply some changes
- Check the changes with container diff
- Commit the changes to a new image

The procedure is not repeatable or scalable…

Dockerfile

A Dockerfile is a text file containing instructions on how to build a custom container image.

Each line in a Dockerfile results in a layer in the resulting image filesystem.

After defining a Dockerfile, an image can be created by executing the command:

```
$ docker image build –t <imagename> .
```

in the directory containing the Dockerfile

Dockerfile keywords

- **FROM**: selects the base image we want to start building our custom image from
- **RUN**: execute a command inside a dummy container instantiated from the current image content
- **COPY**: add some content from the host to the current image content

Dockerfile keywords

- **WORKDIR**: defines the working directory that is used when a container is run from our custom image
- **CMD/ENTRYPOINT**: defines the command that should be executed when running a new container from this image with no parameters

Dockerfile reference

https://docs.docker.com/engine/reference/builder/

For a reference of all the available instructions that can be used in a Dockerfile.

Networking

Networks

Docker can create different networks to let groups of containers communicate.

To create a new network:

```
$ docker network create <network name>
```

When running a container, you can specify the network to use with the argument:

```
--network <network name>
```

Networks

To communicate with the host, ports opened in a container can be forwarded to ports in the host.

A container can communicate to some other container by using the container name as the host address. Docker will take care of translating names to the local IP addresses automatically