Swimming with whales

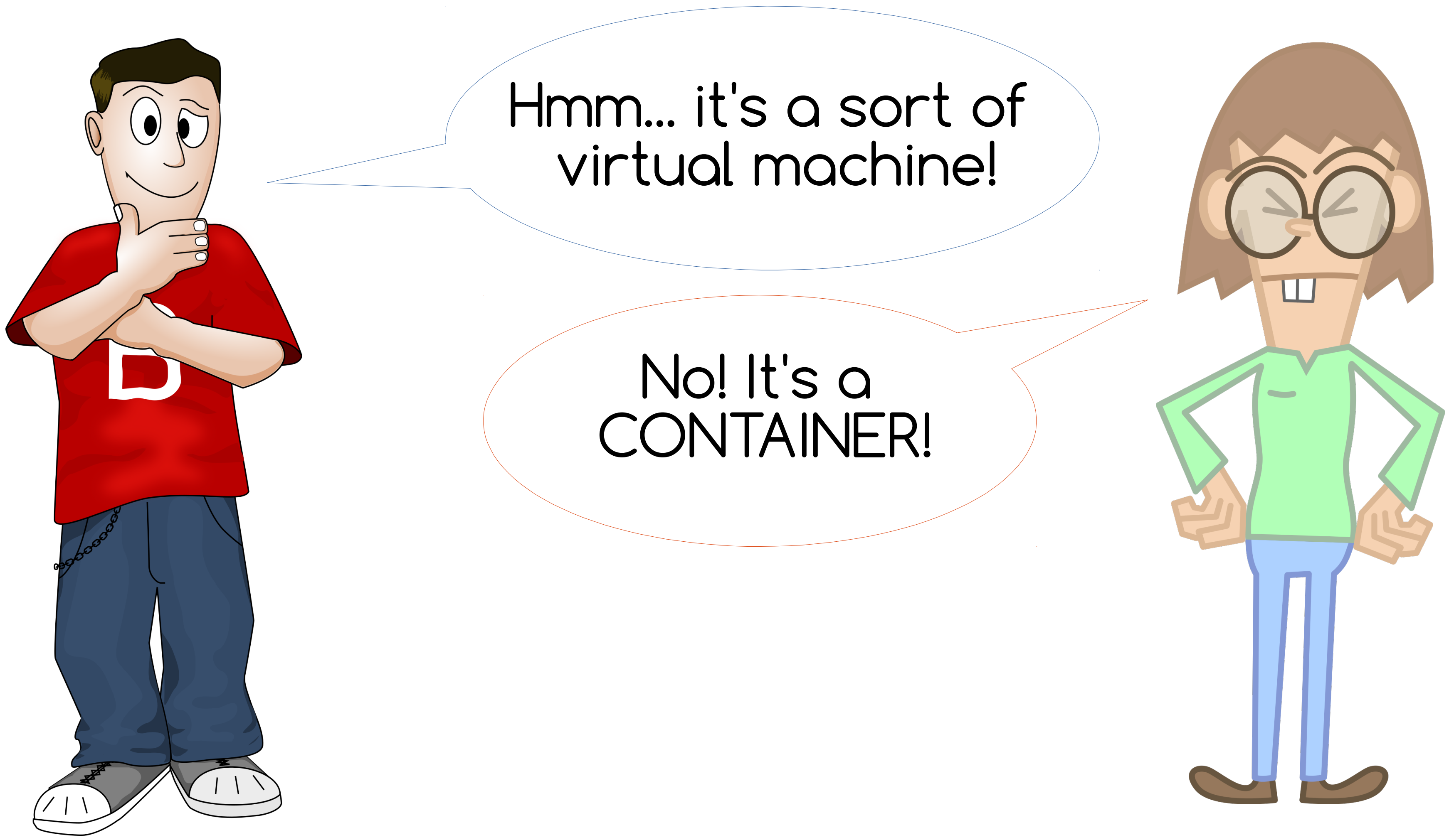
A “gentle” introduction to docker

Stefano Piani
eXact-lab s.r.l.

Outline

- What is Docker?
- How Docker works
- The ingredients
- A first hands-on
- The Docker daemon
- The Docker images
- A real example
- Brief summary

What is Docker?



What is Docker?

It just feels like a virtual machine, because:

- You interact with a container using a shell
- You can install libraries (even as root) and change files inside a container and nothing changes in the "host"
- You can freely start and stop the containers
- Inside a container, there is no way to tell what is happening on the host

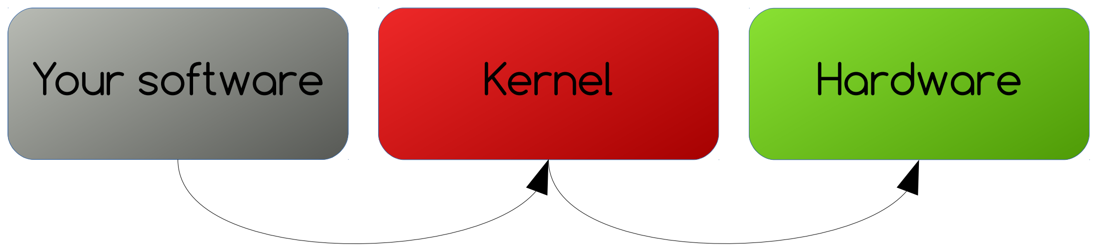
What is Docker?

A Docker container is an isolated environment where a process can run

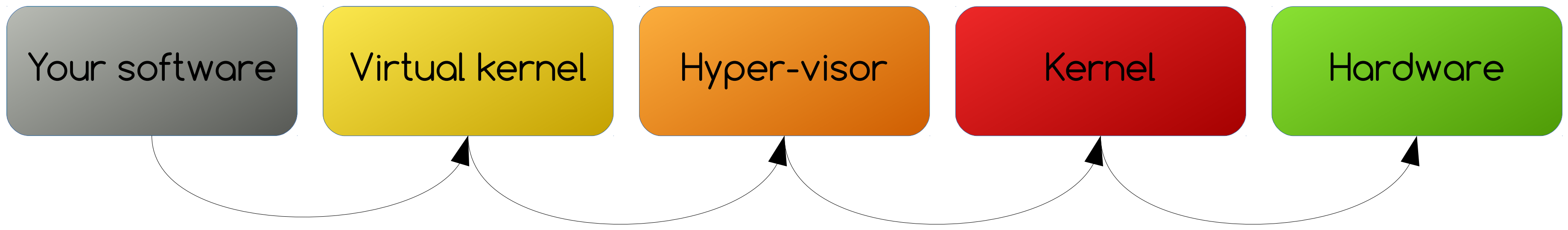
Why is this different from a virtual machine?

How Docker works

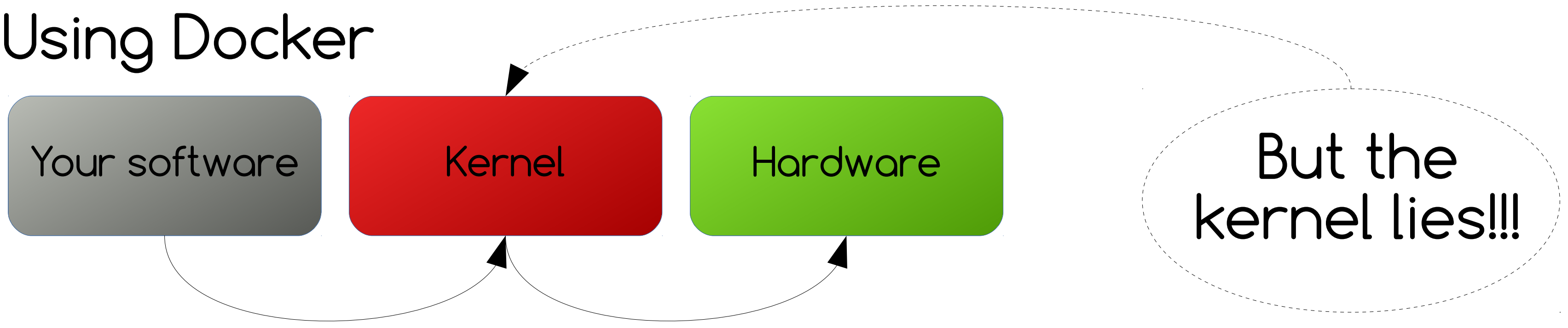
The usual work flow



On a virtual machine



Using Docker



How Docker works

When you are inside a container:

If you ask how many processes are running

If you ask the PID of a process

The kernel lies!

If you ask the amount of free space

If you ask how many network interfaces are up

How Docker works

When you are inside a container, the kernel isolate you from the other processes that are running on the machine (and from its filesystem, its devices and so on...) making you believe that you are in a complete different environment!

But your code runs natively!!! (which means: almost no overhead)

How Docker works

Using Docker:

- All containers share the same kernel (the one of the host)
- You can not run containers with different OS
- You can not use different kernel modules among containers
- In a docker container usually there are just a few processes, not an entire operating system (no cron, no init...)

The ingredients

How can Docker work?

There are actually 3 main ingredients

- cgroups: limit what you can do
- namespaces: limit what you can see
- overlay: the peculiar filesystem that Docker uses

The ingredients

*“**cgroups** (abbreviated from **control groups**) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.*

Engineers at Google (primarily Paul Menage and Rohit Seth) started the work on this feature in 2006 under the name "process containers"

Wikipedia

The ingredients

- The control groups functionality was merged into the Linux kernel mainline in kernel version 2.6.24, which was released in January 2008
- Using cgroups it is possible to limit the resources of a particular set of processes (like memory, cpu, I/O, ...)
- Cgroups uses a complex hierarchical system to decide what a process can or can not do. Luckily, Docker manage it for you!

The ingredients

- **Namespaces** are a feature of the Linux kernel that isolates and virtualizes system resources of a collection of processes
- Implemented since 2002 in the 2.4.19 kernel but some features were added later
- Everything that we are going to use has already been implemented in the 3.10 kernel

The ingredients

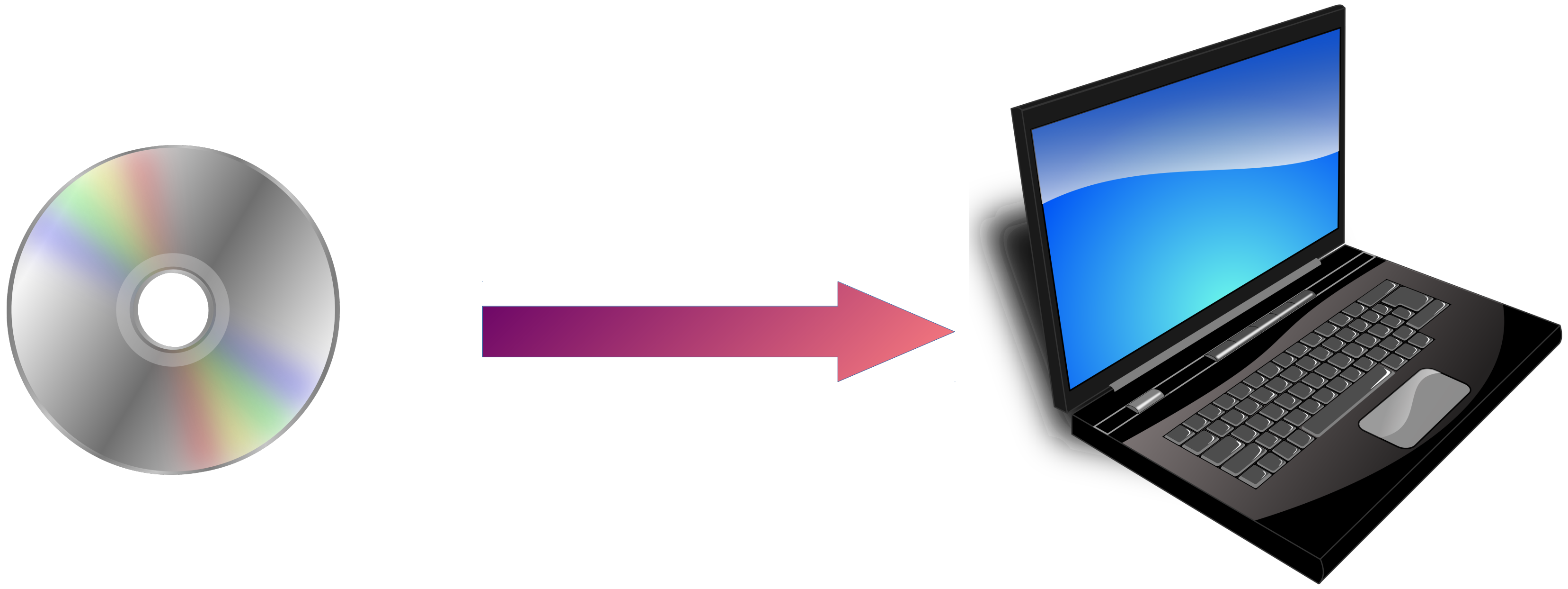
There are several kind of namespaces for different attributes:

- Mount
- Process id
- Network
- Interprocess communication
- UTS
- User ID
- ...

The ingredients

The ingredients

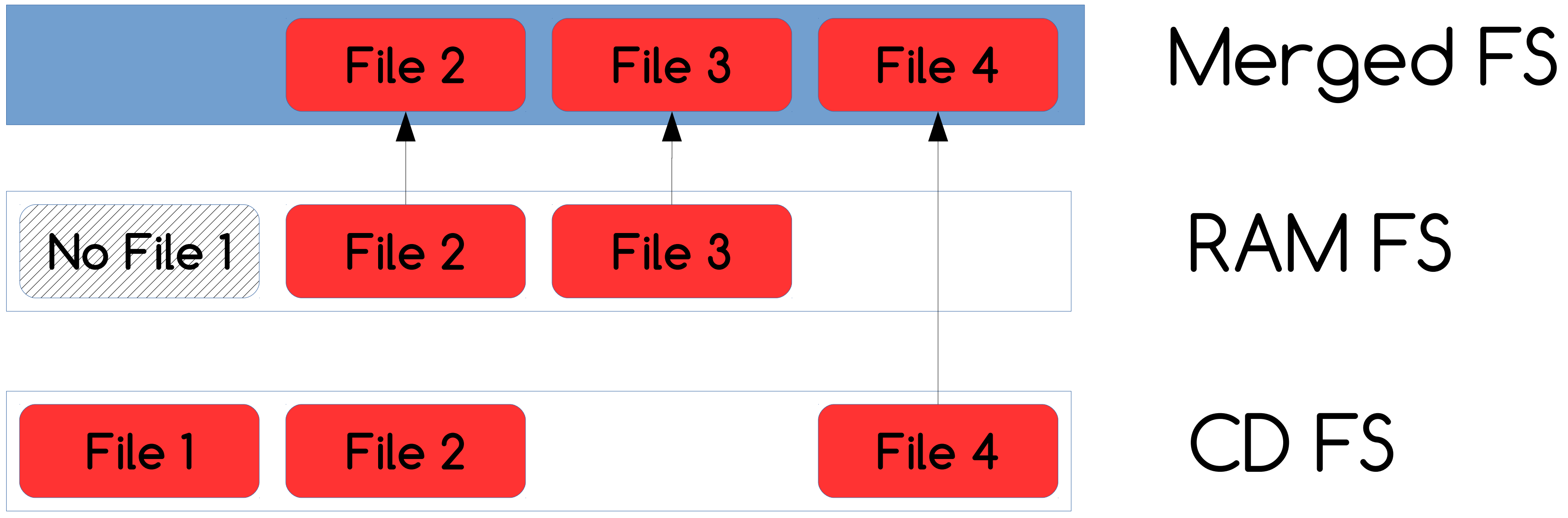
How does a live CD work?



What happens if I write a file on the disk?

The ingredients

Two filesystem, one over the other!



What is the connection with Docker???

The ingredients

- A docker container is just a process of your machine isolated from the network stack, the other processes and the filesystems
- But, maybe, my process relies on some libraries that are on the filesystem!
- How can I use these libraries with my isolated process? YOU DON'T! (even if you can)

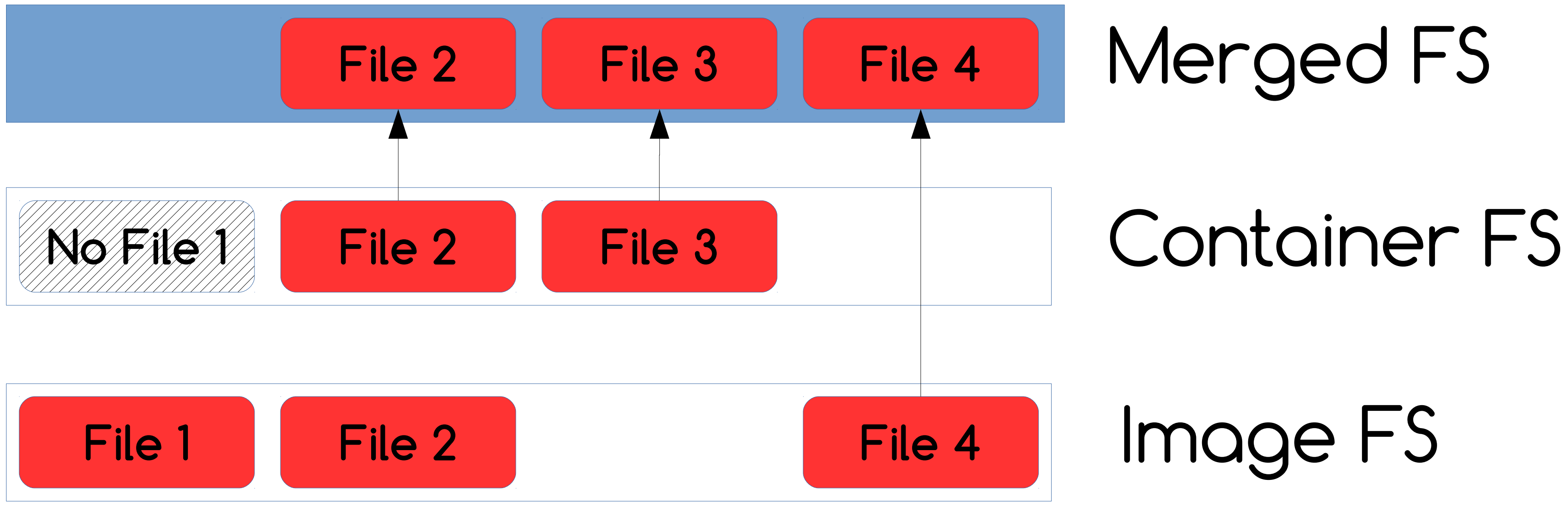
The ingredients

- What you do is put inside a “directory” everything you need to run your software
- In Docker, these directories are called images

The ingredients

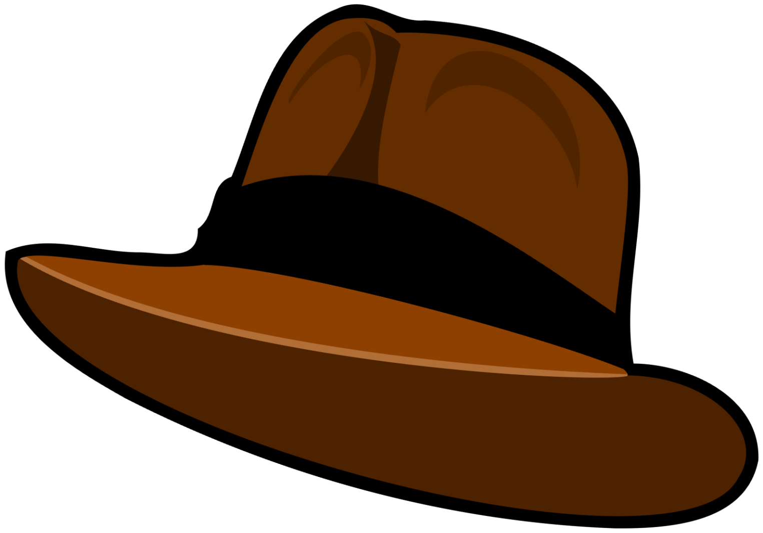
- A docker image contains a “mini operating system” with all the libraries and the utilities that we need to run and manage our process
- The image is created once and then is immutable. Each container is started from an image
- The situation is analogue to the previous one (the live cd)

The ingredients



A first hands-on

Let's explore Docker with our hands!



A first hands-on

We want to see if there are some images

- `docker images`

You can think as repository as the “name” name of the image.

Tag is a string that indicate the version of the image

Usually, you refer to an image as “repository:tag”. If you do not specify nothing, the “tag” is automatically “latest”

A first hands-on

Let's try to start our first container

- `docker run -it IMAGENAME`

Use the image `docker/whalesay:latest`

Now you are inside your first container! (and you are root inside it)

This image has been build to run an “improved” version of cowsay.

A first hands-on

Have a look around! Whatever you will do will not affect the original image or any other file on your system!

Try to install a package like python using apt-get!

And execute a long-running script like the following:

```
>>> i=2
>>> while True:
...     i = i*2
```

A first hands-on

Open a shell on your host machine and run `top`. What do you see?

Docker isolates your processes from inside to outside, not the other way around!

If you have root permission on your host machine, you can kill the python process from outside, otherwise use Ctrl+C from inside the container to stop the python script!

A first hands-on

Launch `top` inside your container! What do you see? What is the PID of your bash?

The python process that you have launched before was born inside your container and died there!

This is fine because that process was a child of the bash process!

But your bash process is different! Its PID is 1. The bash process is the main process of the container!

A first hands-on

The process with PID 1 behaves like the init process in a linux machine!

The main process is running

↕

The container is running

Use `Ctrl+P, Ctr+Q` to detach from your container!

A first hands-on

Let's see how many containers are running

➢ `docker ps`

We have not specified a name for our container when we created it. Docker, therefore, has chosen a random one.

`COMMAND` is the name of the main process of the container

We can go inside the container with the command:

➢ `docker attach DOCKERNAME`

A first hands-on

Use `exit` to stop the bash process from inside your container

Now you are outside your container! Use again

- `docker ps`

to see if your container is still there!

Try again with

- `docker ps -a`

A first hands-on

Now you have two choices:

- You can `run` another container from the image
- You can `start` the old container again

If you use `start`, then you must also attach to your container with

- `docker attach CONTAINERNAME`

Will python be already installed?

A first hands-on

- `docker start` simply runs again the same command using the filesystem of the container
- A stopped container does not use any resource beyond the disk space occupied by its filesystem.
- You can stop a container terminating its main process or with the command
  - `docker stop CONTAINERNAME`

A first hands-on

CLEAN UP!

Stop all your containers and delete them with

- `docker rm CONTAINERNAME`

A first hands-on

Usually, you do not use bash as the main process for your containers!

Each image has a “COMMAND” specified inside itself, but you can override this value. For example, try to do something like that

-
  ```
  docker run -it
  docker/whalesay:latest cowsay
  Greetings from Docker!
  ```

Why are not we inside the container any more? Is the container still running? If not, can you start it again and attach to it?

A first hands-on

Time to take a step back!

The run command is just a shortcut for three actions:

- `docker create` (create a container from an image)
- `docker start` (start the container)
- `docker attach` (attach your standard input, your standard output, and your standard error to the container)

A first hands-on

If you use the `-d` flag with the `run` command, you can avoid the attach phase.

A container runs even if you are not attached to it.

The “problem” with your image is that your process is executed and then it terminates. And, when your main process terminates, the container does not run any more!

A first hands-on

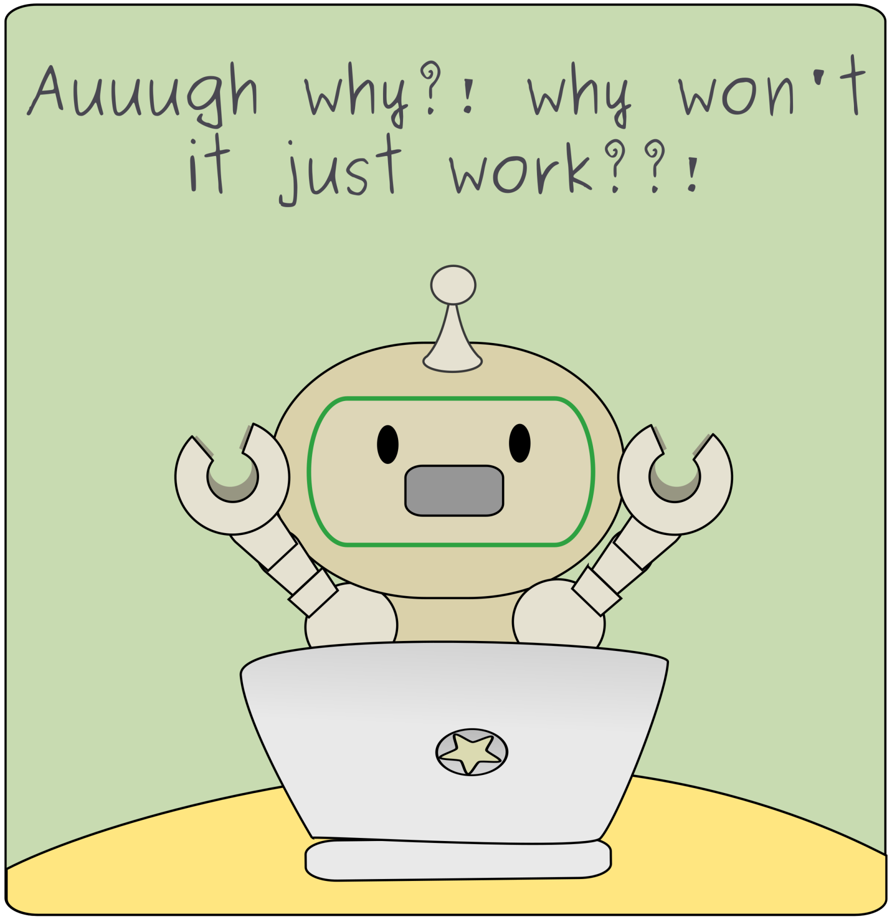
Let's clean all the containers and start again!

This time, we will use '/bin/bash' as COMMAND. To make everything more comfortable, we will give a name to the container:

- `docker run --name test01 docker/whalesay:latest /bin/bash`

Is everything fine?

A first hands-on



A first hands-on

Let's see what is wrong this time! What is the exit status of the container?

You have executed the following operations:

- create
- start
- attach

When you have started the container, you were not attached to the container and, therefore, the standard input file was closed.

A first hands-on

If bash realise that there is not standard input, it stops its execution because no more command can be submitted, and if bash stops its execution...

We must keep the standard input open even if we are not attached to the container! Use the `-i` flag!

- `docker run -i --name test02 docker/whalesay:latest /bin/bash`

A first hands-on

Now it works, but you are missing your terminal!

Use the `-t` flag:

- `docker run -i -t --name test03 docker/whalesay:latest /bin/bash`

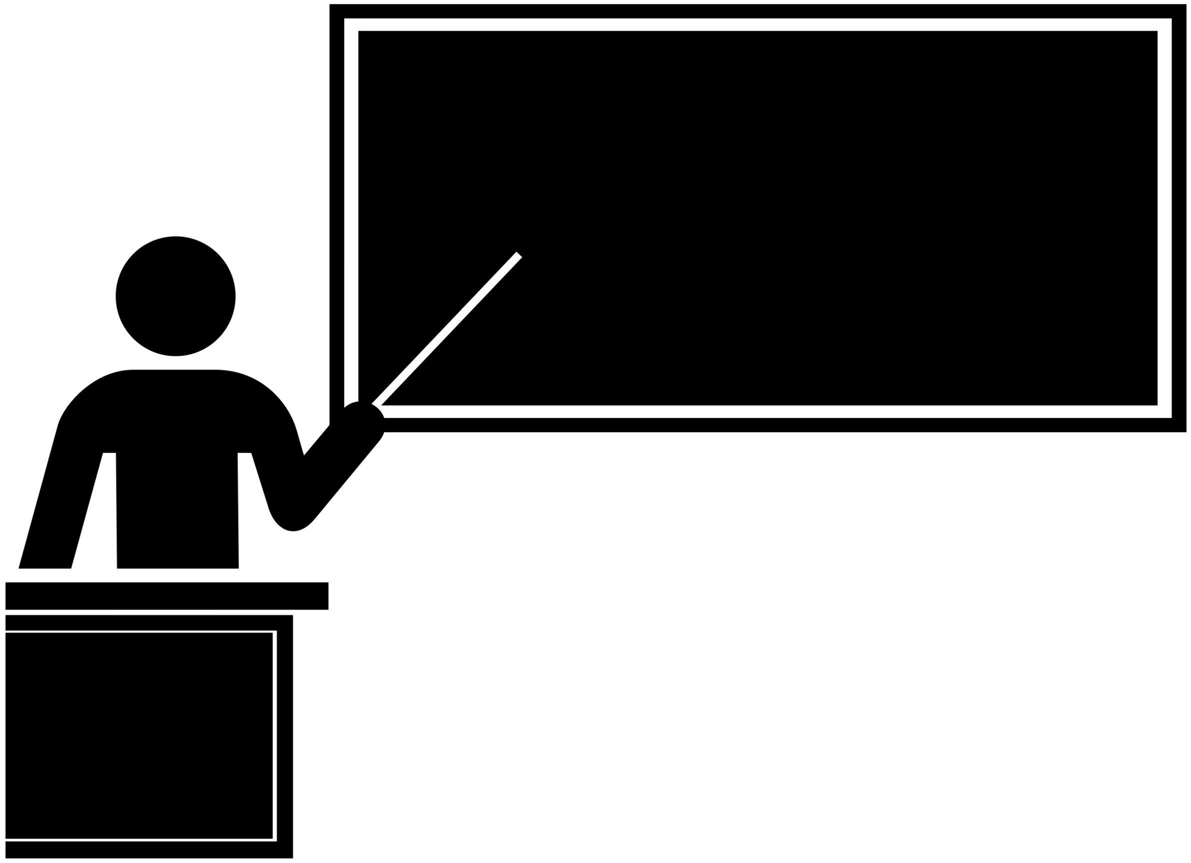
or (is the same):

- `docker run -it --name test03 docker/whalesay:latest /bin/bash`

A first hands-on

If you think that you will enter inside a container, always use the -it flags!

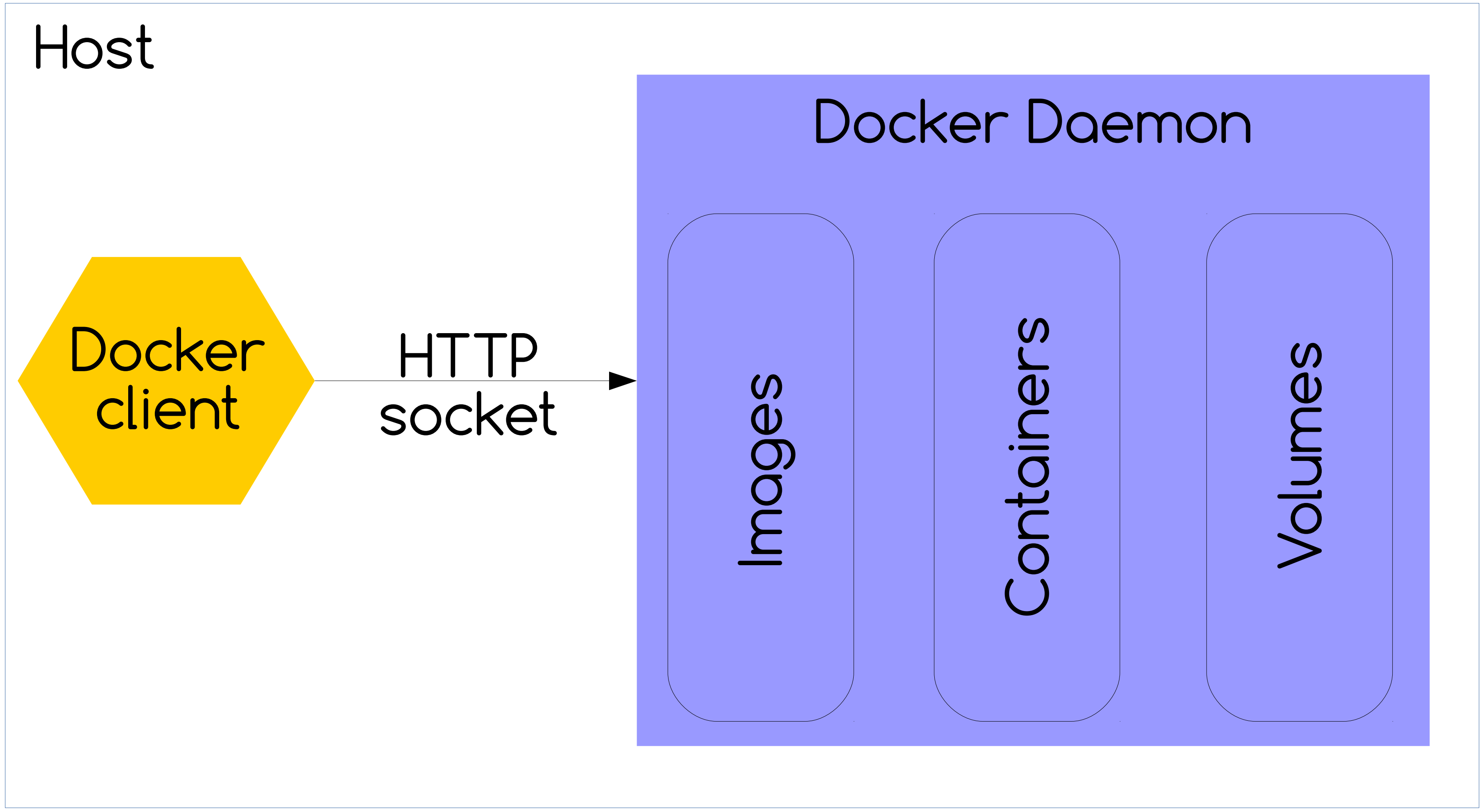
The Docker daemon



Time to learn the theory

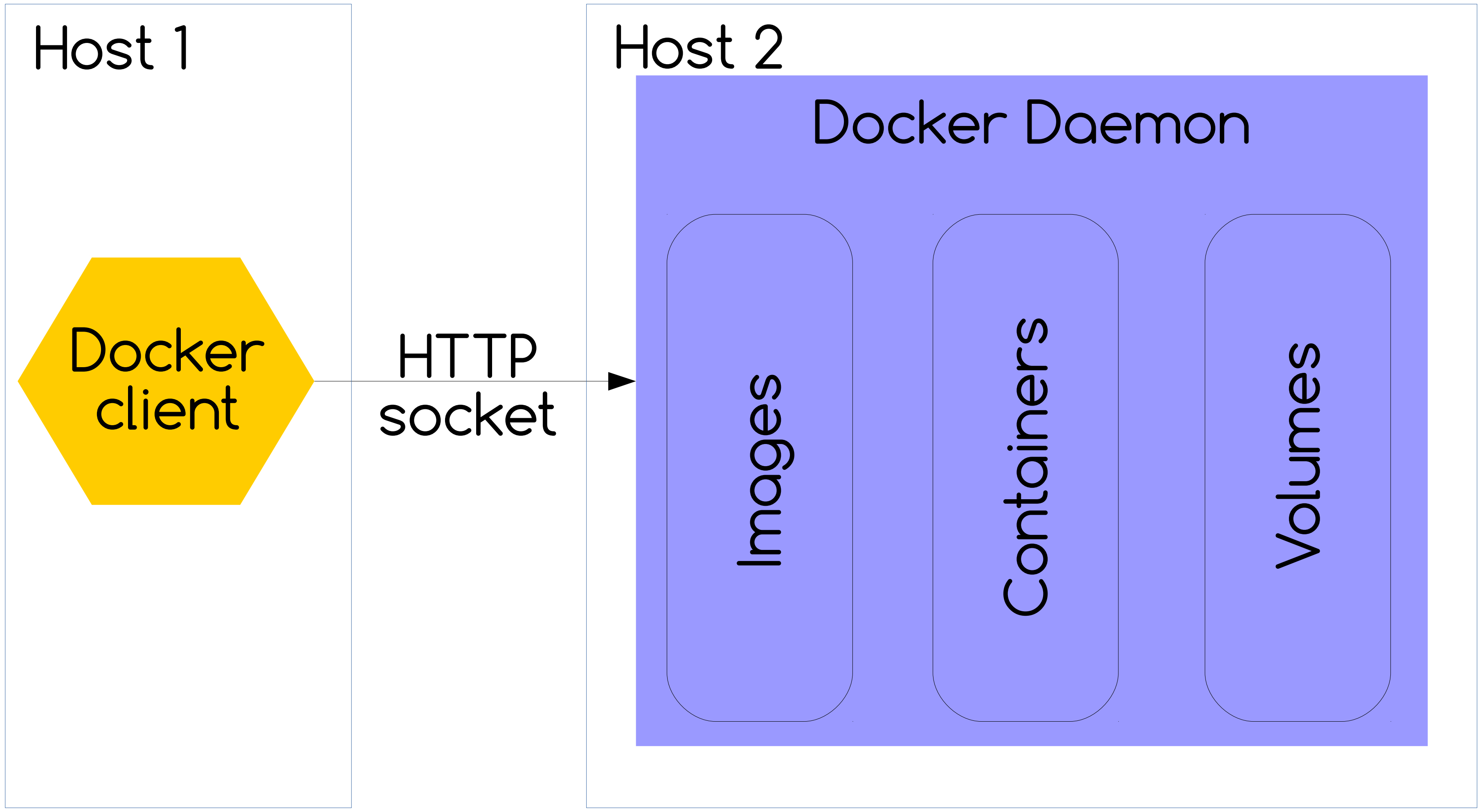
The Docker daemon

When you write `docker`, who are you talking with?



The Docker daemon

When you write `docker`, who are you talking with?



The Docker daemon

- The Docker daemon waits for instructions from a HTTP socket that can be exposed on the network
- Docker gives you a small handy client to talk with the docker daemon
- If you are REALLY brave, you can avoid to use the client and talk directly to the daemon

The Docker daemon

- Or, maybe, you can hope that somebody has already written an interface for your programming language!
- Take a look at docker-py!

The Docker images

If for every container I need an image with an entire operating system, will I run out of space in no time?

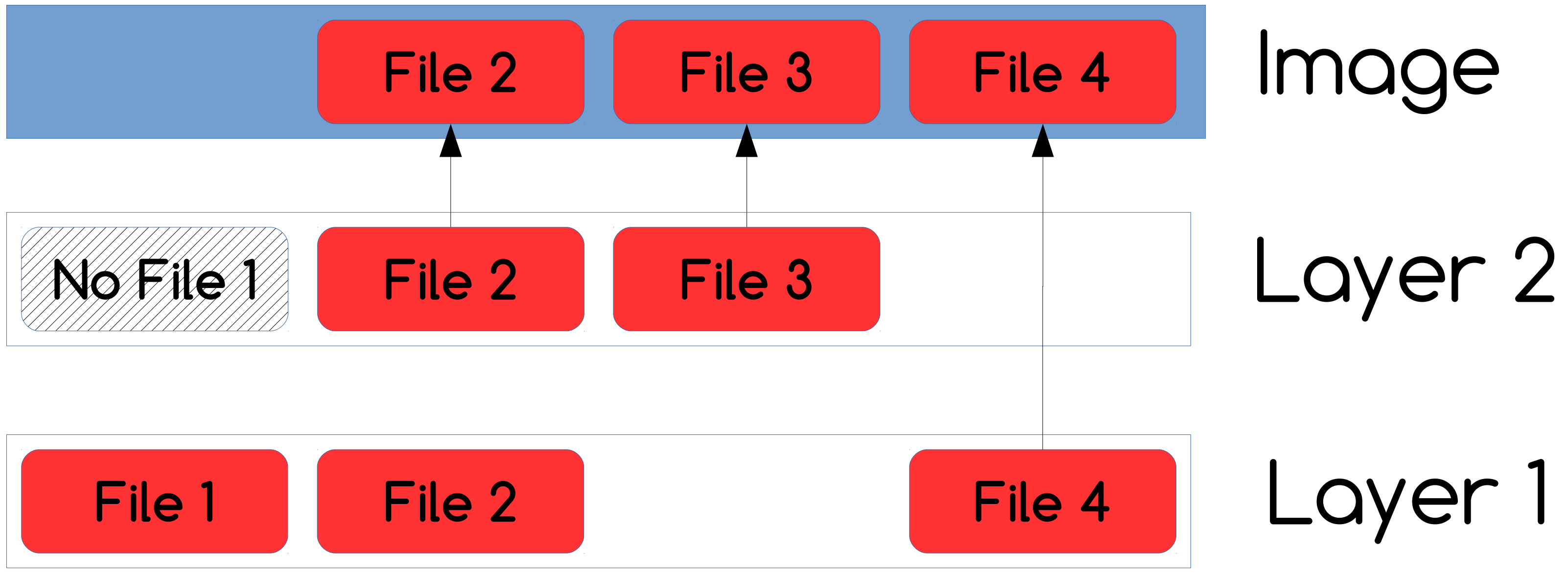
No, because of the layers!

The Docker images

- Every Docker image is made by several layers (up to 127)
- Different images can share they layers
- The layers can be implemented in different ways (depending on the version and configuration of the docker daemon)

The docker images

In the most common one, each layer is mounted over the others using the same technique described before!



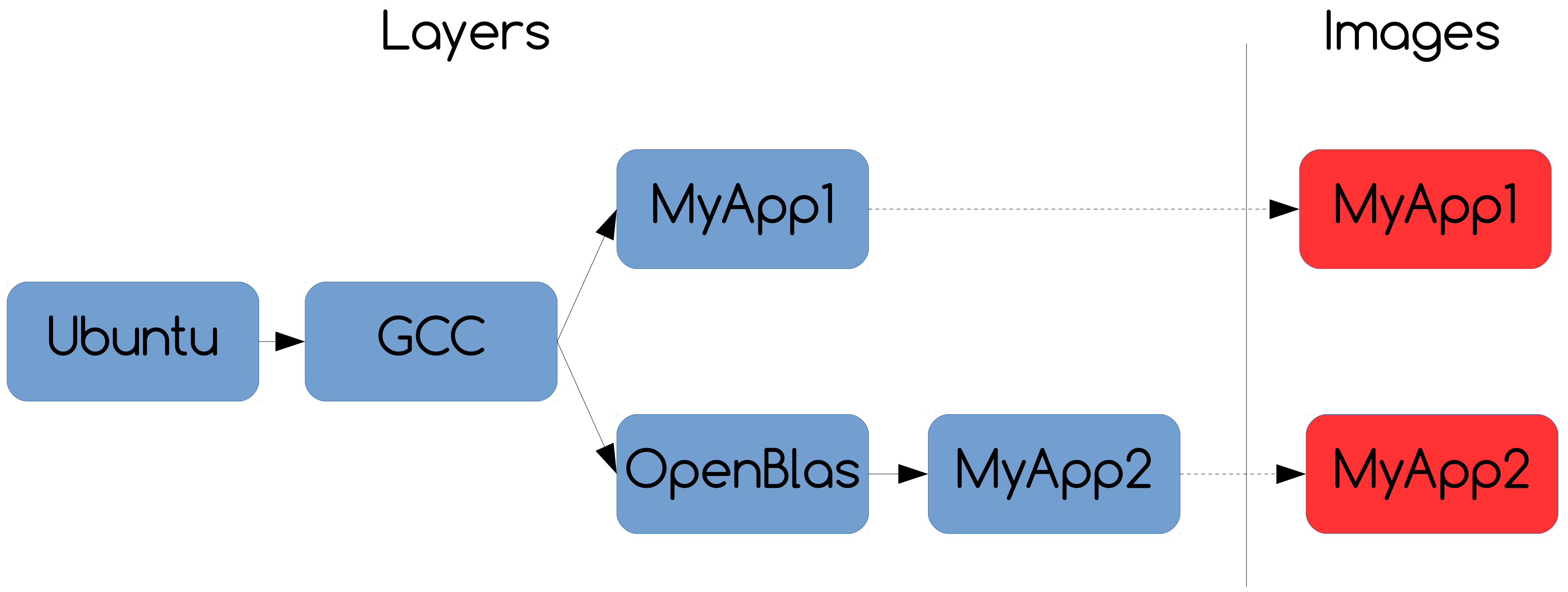
But for hundreds of layers!!!

The Docker images

- **Drawback**: check if a file exists, in a container, may require to inquire hundred of filesystems!
- If you have written a program that opens and closes files continuously, you may see a significant overhead!
- Actually, if you have written a program that opens and closes files continuously, you deserve all the overhead you may encounter!

The Docker images

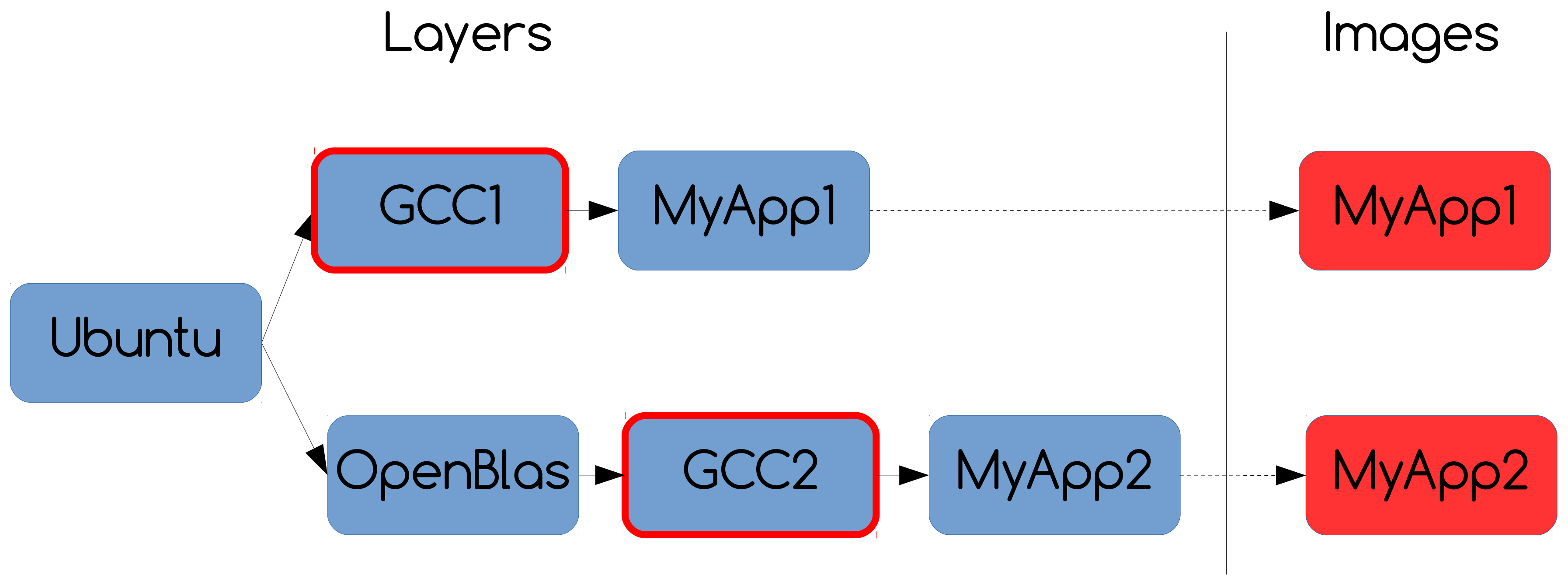
The following is an example about how the layers are managed in Docker:



If MyApp2 was already installed, MyApp1 occupies only the space of its files!

The Docker images

But watch out for the order!!!



If you install OpenBlas before than GCC for the MyApp2 image, then you will have two different layers with GCC (increasing the disk space)

The Docker images

- To create an image, you have to give a “recipe” to docker!
- This recipe is called Dockerfile and is just a small text file with a list of commands.
- Let's try with an example!

The Docker images

- We will build an image with Ipython! It will also be the default COMMAND for our image!
- Create an empty dir somewhere on your machine!
- Create a file named Dockerfile in the directory with an editor
- Write the following line:

```
FROM ubuntu:16.04
```

The Docker images

- The `FROM` command specifies an image that will be used as starting point. Your image, therefore, will be created from another image
- Your image will also inherit all the layers of the ubuntu image
- An image that has not been created from another image is said to be a base image. If you want to create a base image... well... good luck!

The Docker images

- Add the following lines:
  ```
  RUN apt-get update
  RUN apt-get install -y ipython
  ```
- The RUN command execute its arguments like they were typed into a shell
- The `-y` flag after `apt-get install` guarantees that apt-get will not stop its execution asking for a confirm
- Every command that you write in the Dockerfile add one layer!

The Docker Images

- Now we need to tell to our image that we want to execute ipython when the container is started!
- Use the following line:
  `CMD ipython`
- Now its time to build your image! From your shell, use the following command:
  `docker build -t IMAGENAME DIRPATH`
- Use `ipython` as `IMAGENAME`. If your working dir is the directory where the Dockerfile is saved, you can use "." as `DIRPATH`

The Docker images

- Now test if everything works as expected!

  ```
  docker run -it --name ipy-cont
  ipython
  ```

- When you are done, delete the container

  ```
  docker rm ipy-cont
  ```

The Docker images

Exercise time!

I realized that I want also nano (the text editor) in the image! Please, create another image named “ipythonano” with nano installed

PS. For apt-get, nano is in the package “nano”

The Docker images

- There are different ways to solve the exercise!
- First question: Have you changed the old Dockerfile or have you used another one?
- Second question: Have you added a new line like the following

  ```
  RUN apt-get install -y nano
  ```

  or have you changed the old install line in the following way?

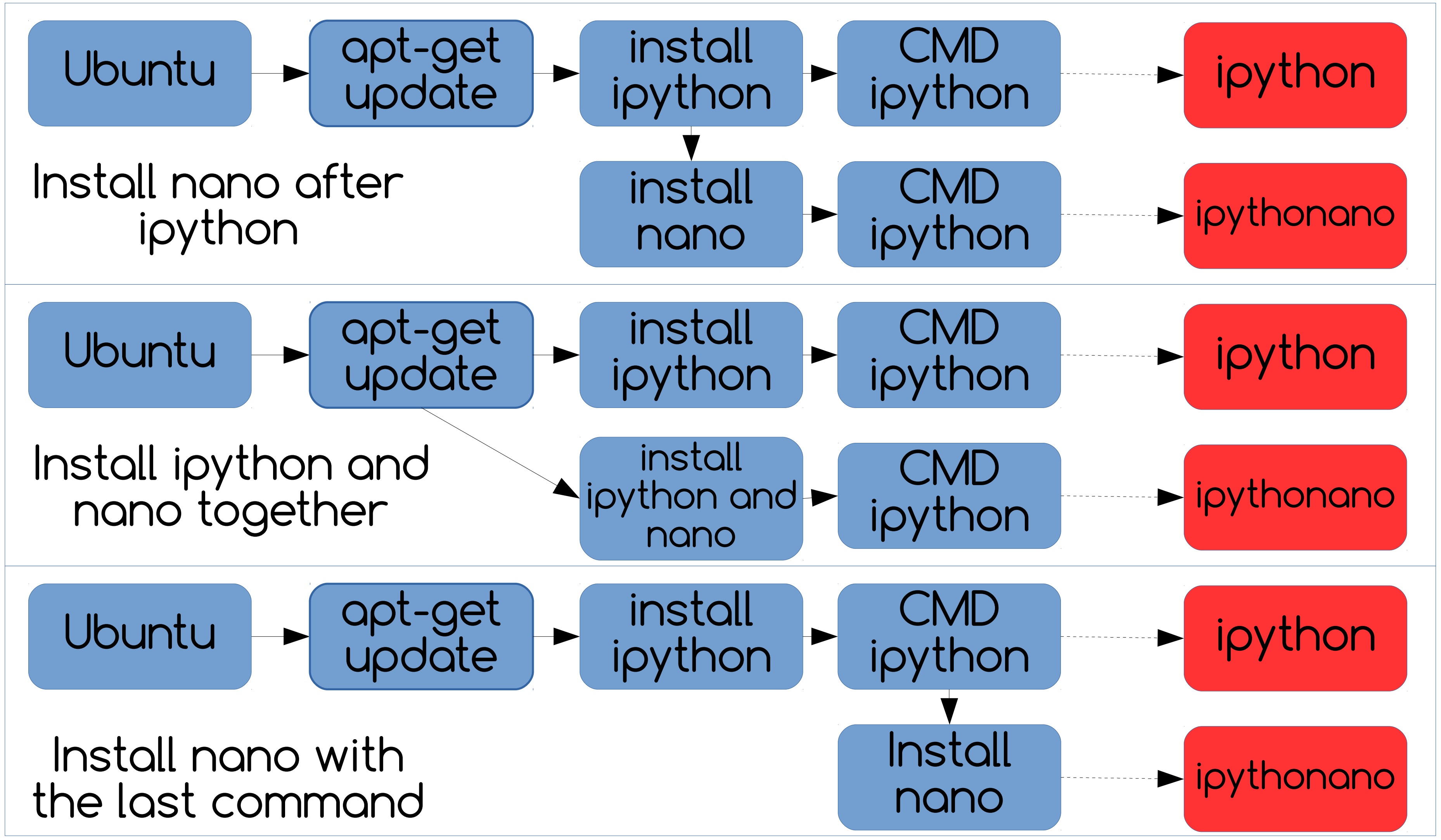
  ```
  RUN apt-get install -y ipython nano
  ```
- If you added a line, where have you put it?

The Docker images

- If you used another Dockerfile, nothing changes!

- The second and third questions, instead, are more subtle!

The Docker images



The Docker images

Exercise time!

Copy your Dockerfile in a new directory. Add only the following line at the end:

- `CMD /bin/bash`

and build a new image with the name “ipythonbash”.

How much time does it take to build the image?

The Docker images

- Docker does not execute a command for which it already has a layer!
- This can be a problem if you use commands that can change their output in time. For example:
  - `RUN apt-get update`
  - `RUN git pull myrepo`
  - `RUN wget http://beautiful/file.tar.gz .`
- Add a comment (with #) to avoid this:

```
RUN git pull myrepo #31-01-2017
```

A real example

- It's time to try to “dockerize” your first application!
- Being in a HPC master, we will use the LINPACK benchmark
- In this exercise, we will not try to achieve the highest result with LINPACK! The aim of the exercise is learn how to create am image to distribute scientific software.
- Therefore, we will use pre-compiled libraries and we will not try to optimize the linpack parameters

A real example

- The `ADD` command, used inside a Dockerfile, adds a file inside an image
- The syntax is the following:

```
ADD file path/inside/the/docker
```

- `file` in the previous example can be a local file but also a URL to a remote file
- `ADD` uses the cache in a clever way: it checks if the files is changed and, if this is the case, it creates a new layer (much better than `RUN wget`)

A brief summary

- This is a good moment to summarize what we have learned about docker so far!
- Docker is different from a virtual machine because:
  - No overhead when you use CPU or memory (but you can have overhead accessing disk if you have too many layers) (GOOD)
  - Starting a container is way faster than starting a virtual machine (GOOD)
  - You can't use containers to try different kernels or different kernel modules (BAD)

A brief summary

- Docker is different from a virtual machine because:
  - Because of the fact that the container uses the same kernel of the real machine, it offers more possibilities to exploit kernel bugs to get control of the physical machine (BAD). <u>Don't run services as root inside a docker container</u>!
  - Docker manages different images in a very efficient ways, saving a lot of disk space (GOOD)

A brief summary

- Docker makes easy sharing applications!
- A docker container has no dependencies (beside Docker itself) and somebody says that its a fully deterministic environment (but this is not entirely true, because you have no control on the kernel space)
- From the host machine, the docker containers are completely transparent: you see the applications running in a container as they were launched natively on the host. You can even use a debugger on an application inside a container!