Containers

Linux Kernel and Docker

Roberto Masocco
roberto.masocco@uniroma2.it

University of Rome "Tor Vergata"
Department of Civil Engineering and Computer Science Engineering
Intelligent Systems Lab

May 5, 2022

TOR VERGATA
UNIVERSITY OF ROME
School of Engineering

Disclaimer

What follows is heavily based on specific features of the Linux kernel.
Compatibility with different platforms cannot be guaranteed.

Roadmap

Roadmap

1. Containers
2. Docker
3. Docker Compose

Why Containers?

Example: Packaging Applications

Suppose you are ready to distribute your new application:

- you need to be sure that it is compatible with all the **platforms** you chose to support;
- you need to figure out a way to deal with **dependencies**;
- you want to publish some kind of self-contained, easily-identifiable **package**.

Why Containers?

Example: Isolating Applications

Suppose you are deploying applications on a server:

- you want to define **resource quotas** and **permissions** for each;
- you want to be sure that each module has what it needs to operate, but **nothing more**;
- you want to **isolate** each module for security reasons, in case something goes wrong.

Why Containers?

Example: Replicating Environments

Suppose you are developing applications for a specific system (maybe with a different architecture):

- you want to have a **local copy** of such system without carrying one with you;
- you want to have all **libraries** and **dependencies** installed without tainting your own system;
- you would like to **deploy** the entire installation with just a few commands, without running any script but simply copying data.

Why Containers?

A possible solution to many of the previous situations could be a set of **virtual machines**.
However, virtual machines are **slow**, hypervisors take up **system resources** and guest kernels must always be **tweaked**.
In each of the above scenarios something simpler would be enough, especially since **the OS is not involved**, only applications are.
This is what a **container** is.

Figure 1: FreeBSD jail logo

Containers in the Linux kernel

Support for containers was added to the Linux kernel with a set of **features** starting from kernel 2.6 (2003), mainly:

- **control groups** (`cgroups`): defining different resource usage policies for groups of processes;
- **namespaces**: isolating processes and users in different "realms", both hardware (e.g. network stack) and software (e.g. PIDs);
- **capabilities**: defining what a process can do, with both hardware and software resources.

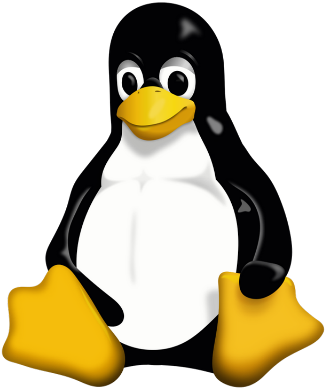

Figure 2: Tux

Containers in the Linux kernel

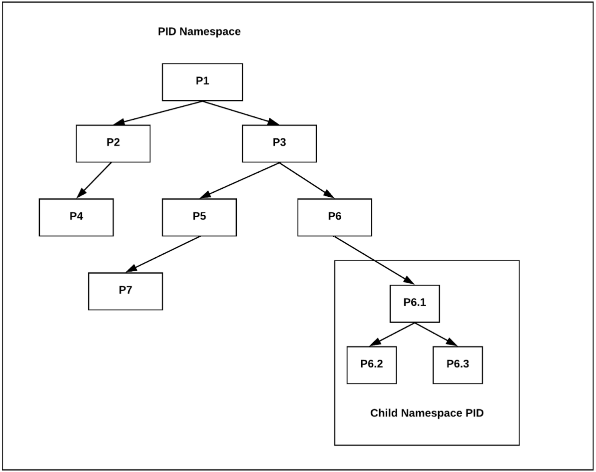

Figure 3: Nested PID namespaces

Roadmap

1. Containers
2. Docker
3. Docker Compose

Docker Engine

Docker is the currently de-facto standard for building, managing and distributing **multiplatform** containers.
It is an engine (i.e. a collection of **daemons**) that automates the management of the kernel subsystems in order to set up, store and run containers.

Figure 4: Docker logo

Docker Engine

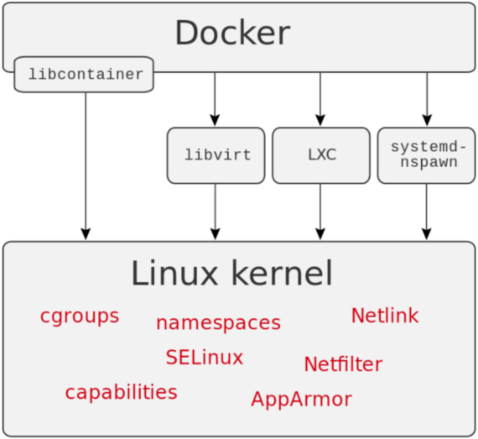

Figure 5: Docker Engine scheme

Containers in Robotics

Containers can be of help in some classic scenarios:

- **deploying** applications or whole control architectures, solving issues like **dependencies** and **configurations**;
- configuring and distributing **development environments**;
- expanding the capabilities of **(partially) closed-source** hardware solutions (e.g. Nvidia Jetson...);
- working with **multiple architectures** at the same time: Docker fully supports **QEMU** to build and run containers.

Building a Docker Container

1. A **Dockerfile** specifies a set of rules to build an **image**, just like a script.
2. **Images** are the binary archives from which a **container** can be started: they can be stored, pulled or simply built locally.
3. A **container** can be built from an image and then started, stopped and managed by the Docker daemon.
4. Processes started inside the container are subject to its limitations, e.g. **filesystem jails** prevent them to climb up to the hosts's filesystem.

Images are built **incrementally**: each Dockerfile directive defines a new **layer**, and the Docker engine stores the differences between each build step thanks to filesystem capabilities: this allows to efficiently **cache build stages**.

Dockerfiles

```
ARG VERSION=20.04
FROM ubuntu:$VERSION # Note the tag!

ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update && \
    apt-get install -y --no-install-recommends \
    build-essential \
    git && \
    rm -rf /tmp/*

ENV DEBIAN_FRONTEND=dialog
LABEL maintainer.name="Roberto Masocco"
CMD ["bash"]
```

Listing 1: Minimal example of a Dockerfile running an Ubuntu image in a container

Dockerfile commands

Just to name a few (see the Dockerfile reference for more):

- `FROM repository/image:tag`
  Specifies a base image to pull.
- `RUN command`
  Runs the following command in a new shell inside the container.
- `COPY source target`
  Copies a file into the container.
- `ENV variable=value`
  Sets an environment variable inside the container.
- `ARG name=value`
  Declares a build argument.
- `CMD ["command", "arg1", ...]`
  Specifies the command to run when the container is started.

Docker Commands

Again, just a few (each with a gazillion of options):

- `docker build`
  Builds a new image from a Dockerfile.
- `docker run`
  Builds and starts a container.
- `docker ps`
  Lists active containers.
- `docker exec`
  Runs a command inside a container (e.g. a shell).
- `docker start`
  Starts a container.

Docker Commands

- `docker stop`
  Stops a container.
- `docker images`
  Lists available images.
- `docker rm`
  Removes a container.
- `docker rmi`
  Removes an image.

Active containers are usually referenced by their **ID**.

Roadmap