Inside the roboticist's toolbox

Linux kernel, Docker, and more

Roberto Masocco
roberto.masocco@uniroma2.it

University of Rome Tor Vergata
Department of Civil Engineering and Computer Science Engineering

June 5, 2024

TOR VERGATA
UNIVERSITY OF ROME
School of Engineering

Roadmap

1. Containers

2. Docker

3. Docker Compose

Roadmap

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 **software copy** of such system without having to carry it with you;
- you want to have all **libraries** and **dependencies** installed without tainting your own;
- you would like to **deploy** the entire installation with just a few commands;
- you would like to **avoid reinstalling** or **reflashing** the OS every time something changes.

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**: granting some of the superuser's permissions to unprivileged threads with very fine-grained control (*e.g.*, network protocols, scheduling policies...).

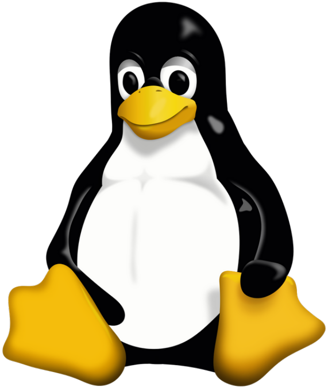

Figure 2: Tux.

Containers in the Linux kernel

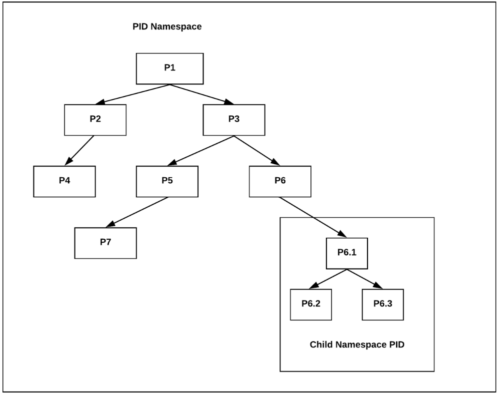

Figure 3: Nested PID namespaces: processes in each namespace can only address processes in their own namespace, but the parent process of a namespace can address also all those in its child.

Roadmap

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.

Today, it is not the only option. It has contributed to the birth of the **Open Container Initiative**, which standardizes many of the concepts behind containers, making them **interoperable** with other containerization solutions.

Figure 4: Docker logo.

Docker Engine

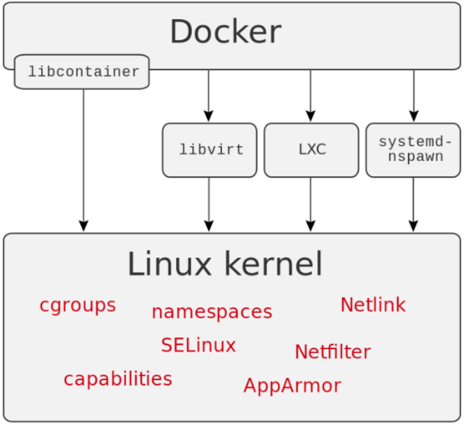

Figure 5: Docker Engine scheme.

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**;
- working with **multiple architectures** at the same time: Docker fully supports **QEMU** to build and run containers;
- expanding the capabilities of **(partially) closed-source** hardware solutions (*e.g.*, Nvidia Jetson...);
- avoiding continuous **reinstallations** and **reflashings** of the OS.

Building a Docker container

Step by step

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 from a remote **registry** 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 host 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 the **OverlayFS union filesystem**.

For every new container, its filesystem will be in a **new top layer**.

This allows to efficiently **cache and share build stages**, which will then be stacked together to form images, but operating in a **copy-on-write** fashion (*i.e.*, modifications to the lower levels are **slower** than those to the uppermost ones).

```
ARG VERSION=22.04
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 /var/lib/apt/lists/* /tmp/* /var/tmp/*/apt/lists/*

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

Listing 1: Minimal example of a Dockerfile running a Bash shell in a Ubuntu container.

Dockerfile commands

- `FROM repository/image:tag`
  Specifies a base image to pull.
- `RUN command`
  Runs the following command in a new `sh` shell inside the container.
- `COPY source target`
  Copies a file into the image (see also `ADD`).
- `ENV variable=value`
  Sets an environment variable inside the container from that line on.
- `ARG name=value`
  Declares a build argument (can be specified from the CLI).
- `CMD ["command", "arg1", ...]`
  Specifies the command to run when the container is started (see also `ENTRYPOINT`).

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, optionally overriding the command (`CMD`).
- `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.

Containers and images are usually referenced by their **ID** (*e.g.*, `abae6cae4648`).

See the Dockerfile reference and the Docker CLI reference for more.

Containers on real robots

Best practices

To run containers on embedded systems and robot SBCs, some configurations are **suggested** which **would not apply in traditional containerization scenarios**:

- the host **network stack** should be fully exposed to allow for **ROS** and other network-based **middleware** to work properly (`--network host`), otherwise, virtual NAT will block you;
- the **IPC namespace** should be shared to allow for **shared memory** and **IPC** to work properly (`--ipc host`) (*e.g.*, Fast DDS uses shared memory when possible by default);
- to allow access to the **hardware** mounted on the host, one should grant full privileges to the container (`--privileged`) and mount `/dev` and `/sys` inside it (`-v /dev:/dev -v /sys:/sys`);
- your development directory should be **mounted as a volume** inside the container, so that file manipulations happen on the host filesystem (`-v /your/code:/workspace`).

We would like some utility to **automate** this process...

Roadmap

Composing services

Managing multiple, interdependent **containerized services** can become quite a tedious task.

Each container may take multiple options, some have to be started in sequence or built in a particular way...

Compose is a utility that helps to **build**, **run** and **manage** multiplatform containers by parsing all such settings from **YAML configuration files**.

For instance, **our code repository** cointains development containers managed by Compose.

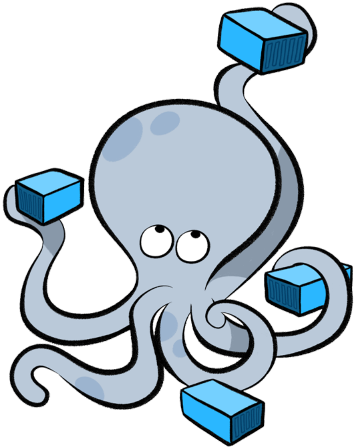

Figure 6: Docker Compose logo.

Compose files

```yaml
services:
  development:
    build:
      context: .
      args:
        TARGET: dev
    image: devenv:latest
    environment:
      TERM: xterm-256color
    network_mode: host
    command: ["/bin/zsh"]
    volumes:
      - ~/.ssh:/home/user/.ssh
```

Listing 2: Minimal example of a Compose file.

Refer to the Compose reference for more.

Compose commands

Pretty much the same that Docker has, but invoked with

```
docker compose
```

instead of `docker` (previously `docker-compose`), and oriented only towards services specified in the local Compose file.

See the Compose CLI reference for more.

Installation instructions for Docker and Compose can be found here and here, respectively, and also in the provided shell script `bin/docker_install.sh`.
On systems with an Nvidia GPU, the NVIDIA Container Toolkit is suggested.
Installation on Windows and macOS requires Docker Desktop, but **lacks many features**.

Example

`ros2-examples`

Remember ros2-examples?
It is a self-contained **ROS 2 development environment built with Docker**!

It also supports the **automated build** of **development containers** in **VS Code**.
Such containers are based on our **Distributed Unified Architecture** project, which is part of Roberto Masocco's PhD thesis.

Their inner workings are totally transparent, but if you're curious see `dua_template.md`.