

Roboticist 101

Software and middleware for robotics

Roberto Masocco
roberto.masocco@uniroma2.it

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

May 17, 2023



TOR VERGATA
UNIVERSITY OF ROME

School of Engineering

Roadmap

- 1 Introduction
- 2 Middleware in robotics
- 3 ROS 2 overview
- 4 Containers
- 5 Docker
- 6 Docker Compose

Roadmap

1 Introduction

2 Middleware in robotics

3 ROS 2 overview

4 Containers

5 Docker

6 Docker Compose

Information

- **Calendar:** from May 17 to June 14, 2023
- **Topics** (both with practical examples):
 - ① Middleware for robotics
 - ② 3D printing basics (with Simone Mattogno)
- **Materials** (for my part):
 - ▶ Code repository: [github.com/ros2-examples](https://github.com/robmasocco/ros2-examples) (humble branch)
 - ▶ Lectures: github.com/robmasocco, Teams directory (this lecture is [here](#))
- **Prerequisites** (for my part):
 - ▶ Basics of C and Python programming
 - ▶ Basics of Git workflow (check out [this tutorial](#) by Atlassian)
 - ▶ Everyday Linux commands and a Unix-like system

Information

- **Calendar:** from May 17 to June 14, 2023
- **Topics** (both with practical examples):
 - ① Middleware for robotics
 - ② 3D printing basics (with Simone Mattogno)
- **Materials** (for my part):
 - ▶ Code repository: github.com/ros2-examples (humble branch)
 - ▶ Lectures: github.com/robmasocco, Teams directory (this lecture is [here](#))
- **Prerequisites** (for my part):
 - ▶ Basics of C and Python programming
 - ▶ Basics of Git workflow (check out [this tutorial](#) by Atlassian)
 - ▶ Everyday Linux commands and a Unix-like system
- **Exam:** ... ask Prof. Carnevale ☺

Program

Middleware for robotics

- ① Roboticist 101 - Software and middleware for robotics
- ② ROS 2 - Workflow and basic communication
- ③ ROS 2 - Advanced communication
- ④ ROS 2 - Node configuration
- ⑤ ROS 2 - Sensor sampling and image processing
- ⑥ ROS 2 - Interfacing with the data space
- ⑦ microROS - Bridging the gap

The roboticist

A new path for control engineers

A **control engineer** is a specialist in the **design** of controllers to drive dynamical systems; the implementation was traditionally left to other specialists.

A **roboticist** is a specialist capable of designing, **building**, and **programming** complex autonomous systems; with skills ranging from computer science to other disciplines.

The former can be very effective as the latter.

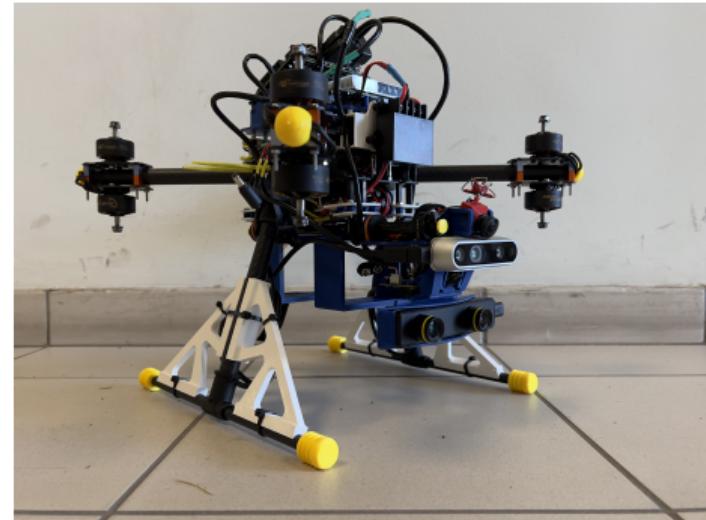


Figure 1: Stanis autonomous drone prototype.

The roboticist

A new path for control engineers

A **roboticist** can usually:

- **design** solutions to complex problems;
- **develop** parts of a robot, or entire **control architectures**;
- **deploy** and test software and hardware solutions;
- make use of modern **hardware accelerators** and robotics-oriented hardware.

Industries are looking for roboticist for their **versatile skill set**.



Figure 2: Nvidia Jetson TX2 developer kit.

Roadmap

1 Introduction

2 Middleware in robotics

3 ROS 2 overview

4 Containers

5 Docker

6 Docker Compose

What is middleware?

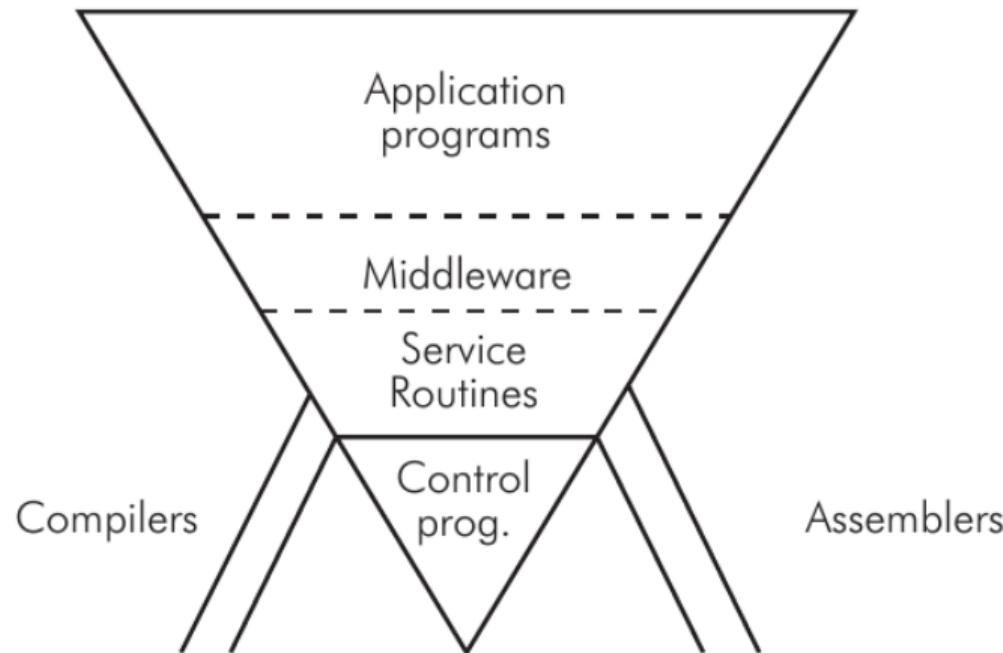


Figure 3: Software organization in a general-purpose computer system.

What is middleware?

Definition of middleware

The term **middleware** identifies a kind of software that offers common services and functionalities to applications in addition to what an operating system usually does.

Middlewares are usually implemented as **libraries** that application programmers can use via appropriate **APIs**.

Middleware in robotics

New problems arising when developing software for modern autonomous systems:

- integration of **sophisticated hardware** (microcontrollers, hardware accelerators, SBCs);
- **software** organization and maintenance;
- **communication** (involves both hardware and software!);
- debugging and **testing**.

Middlewares can help to tackle and solve each one!

Definition of DDS

A DDS is a **publish-subscribe middleware** that handles communications between **real-time** systems and software over the network.

DDSS are currently used in automotive, aerospace, military...

Their implementations follow an **open standard** that defines:

- **serialization** and **deserialization** of data packets (RTPS Wire Protocol);
- **security protocols** and cryptographic operations;
- enforcing of **Quality of Service** policies to organize transmissions (specifying things like **queue sizes**, **best-effort** or **reliable** transmissions...);

Data Distribution Service

- automatic discovery of **DDS participants** (over **multicast-IP/UDP**) and transmission of data (over **unicast-IP/UDP**) (Discovery Protocol).

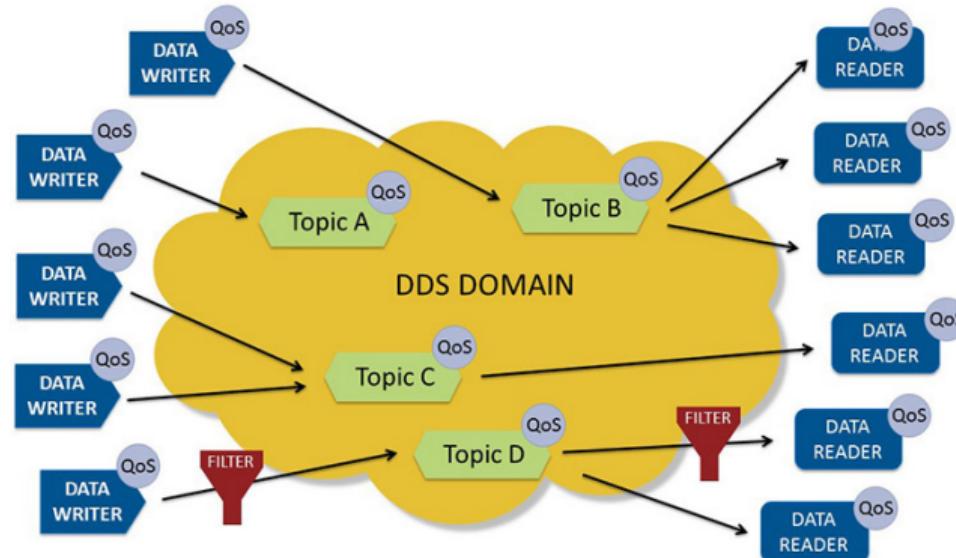


Figure 4: Scheme of a DDS-based network (data space).

DDS participants can either **publish to** or **subscribe to** a topic.

Definition of DDS topic

A DDS topic is **uniquely identified** by three attributes:

- a **name**, *i.e.*, a human-readable character string;
- an **interface**, *i.e.*, a custom packet format that specifies what data is carried over it (e.g. strings, numbers, arrays...);
- a **QoS policy** that specifies how transmissions should be performed.

Changing even only one of the above results in a completely different topic!

Roadmap

1 Introduction

2 Middleware in robotics

3 ROS 2 overview

4 Containers

5 Docker

6 Docker Compose

What is ROS 2?

ROS 2 is a **DDS-based, open-source** middleware for robotic applications. It allows developers to build and manage **distributed control architectures** made of many modules, usually referred to as **nodes**.



Figure 5: ROS 2 logo.

What is ROS 2?

ROS 2 currently supports **C++** and **Python** for application programming, and runs natively on **Ubuntu Linux 22.04**.

New versions are periodically released as **distributions**: the current LTS one is **Humble Hawksbill**; the development version is **Rolling Ridley** and can only be compiled from source. It is available as **binary deb packages** for x86 and ARMv8 architectures.

A **distribution** is a collection of software packages: **libraries**, **tools**, and **applications**.



Figure 5: ROS 2 logo.

Why ROS 2?

The ROS project started in 2007, to provide a middleware that could solve the **software integration** and **communication** problems in robotics. It has evolved much since then.

ROS 2 helps to design and build **distributed control architectures**, providing a common ground for the **integration** of different systems, sensors, actuators, and algorithms. It is a common framework for the development of **robotics software**.

Its adoption is still limited because of familiarity with the original ROS, but it is **growing**.



Figure 6: STM32 (bottom), Raspberry Pi (middle), and Nvidia Xavier AGX (top).

Main features

As a middleware, it offers many **services to roboticists**, including:

- **three communication paradigms**, easy to set up and based on the DDS: **messages**, **services** and **actions**;
- organization of software packages, allowing for **redistribution and code reuse**, thanks to the **colcon** package manager;
- module configuration tools: **node parameters** and **launch files**;
- integrated **logging subsystem** (involves both console and log files);
- CLI **introspection tools** for debugging and testing;

Main features

- may be integrated in some **simulators** (e.g., Gazebo) and **visualizers** (e.g., RViz).

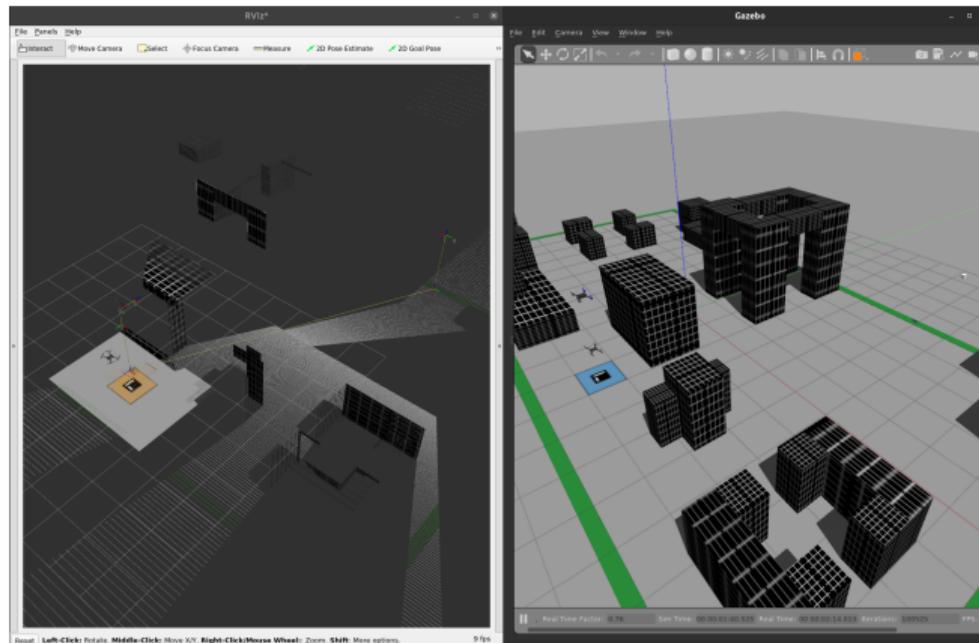


Figure 7: Simulated drone in Gazebo Classic and RViz.

Application Programming Interface

Deep dive into ROS 2 internals

A **ROS 2 installation**, from bottom to top, works as follows:

- ① **DDS**: the middleware that implements the **communication layer** (many different implementations are supported).
- ② **RMW**: ROS MiddleWare, is the **DDS abstraction layer**, which allows to use different DDS implementations without changing the application code.
- ③ **rcl**: ROS Client Support Library (C), implements basic entities: **nodes**, **publishers**, **subscribers**, **services**, **clients**, and **timers**.
- ④ **rclc/rclcpp/rclpy**: ROS Client Library (C/C++/Python), implements the same entities as rcl, plus extended functionalities like the **executor**: a job scheduler.

Then, there are **packages**: libraries, tools, and applications, both officially provided and community-contributed.

Application Programming Interface

How a ROS 2 application works

The most important entity is the **node**, whose functionalities are specified upon creation. A node must usually do **a single thing**, being the **unit** in a **distributed architecture**. With its class methods, it can:

- act as an **entry point** towards the DDS layer, to handle communications;
- embed **software modules** like data, algorithms, and threads, that implement the application logic;
- register **callbacks** to handle **events**, such as timers or incoming messages.

Thus, it is both an **operational unit** and a **communication endpoint**.

Nodes are usually handled by **executors**, which are responsible for scheduling and processing their workload.

Nodes are just objects in your application: they can embed any kind of software module, but they do not limit the design to their paradigms.

Executors

Handling events and callbacks

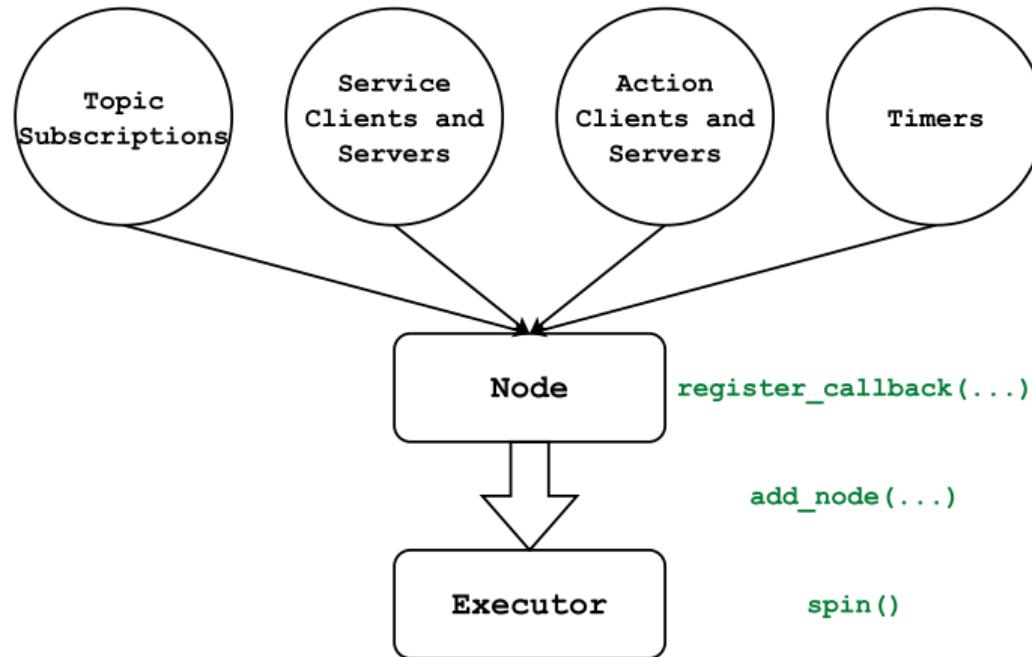


Figure 8: ROS 2 event-based programming paradigm.

Executors

Handling events and callbacks

- ① Middleware functionalities trigger **(a)synchronous events**.
- ② Events are handled by **background jobs**, coded in **callbacks** by the programmer.
- ③ Callbacks are **registered** into a **node** when its functionalities are specified (e.g., upon creation).
- ④ The workload that a node carries is scheduled and processed by an **executor**, single- or multi-threaded.

Executors implement a **round-robin, non-preemptive** policy that **always prioritizes timers**.

ROS 2 biggest flaws (as of today)

The main concerns arise when developing low-level stuff:

- **the DDS layer is almost completely abstracted**, so non-standard network configurations may get tricky;
- the internal job scheduling algorithm (namely the **executor**) is **not suited for hard real-time applications** because of its **non-preemptive** nature.

ROS 2 biggest flaws (as of today)

The main concerns arise when developing low-level stuff:

- **the DDS layer is almost completely abstracted**, so non-standard network configurations may get tricky;
- the internal job scheduling algorithm (namely the **executor**) is **not suited for hard real-time applications** because of its **non-preemptive** nature.

What to do when development gets to a really low level?

- Use something else.
- Hand off stuff to dedicated **microcontrollers**.
- Use **micro-ROS**: hard real-time ROS 2 on microcontrollers and different communication interfaces.

Roadmap

1 Introduction

2 Middleware in robotics

3 ROS 2 overview

4 Containers

5 Docker

6 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 **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.

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 9: 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.

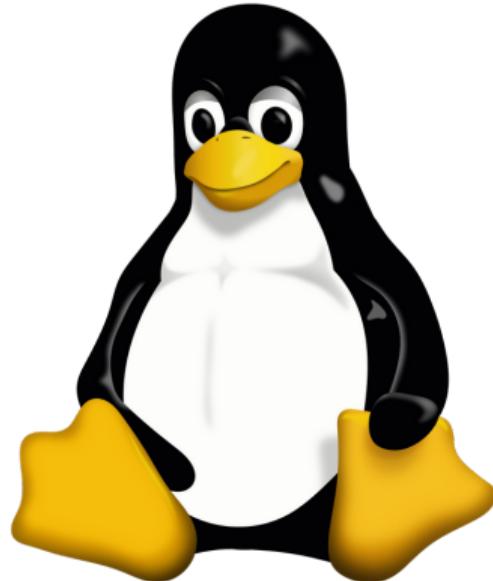


Figure 10: Tux.

Containers in the Linux kernel

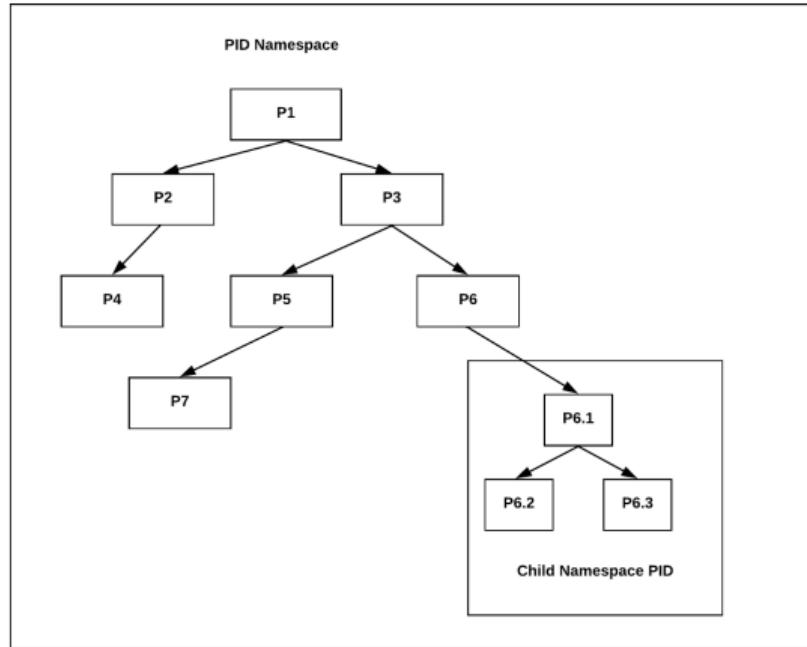


Figure 11: Nested PID namespaces.

Roadmap

1 Introduction

2 Middleware in robotics

3 ROS 2 overview

4 Containers

5 Docker

6 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.

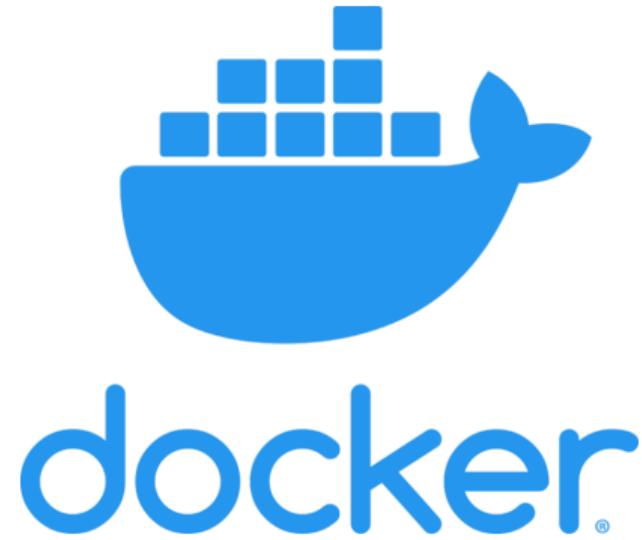


Figure 12: Docker logo.

Docker Engine

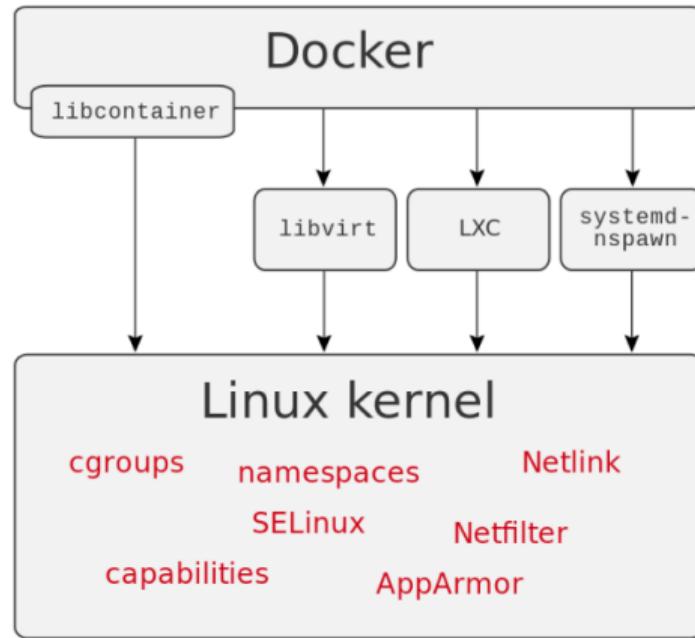


Figure 13: Docker Engine scheme.

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...).

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'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 **OverlayFS**.

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, slower** fashion.

Dockerfiles

```
1 ARG VERSION=22.04
2 FROM ubuntu:$VERSION # Note the tag!
3
4 ENV DEBIAN_FRONTEND=noninteractive
5
6 RUN apt-get update && \
7     apt-get install -y --no-install-recommends \
8     build-essential \
9     git && \
10    rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/* /apt/lists/*
11
12 ENV DEBIAN_FRONTEND=dialog
13 LABEL maintainer.name="Roberto Masocco"
14 CMD ["bash"]
```

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

Dockerfile commands

- **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** (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`);
- the **IPC namespace** should be shared to allow for **shared memory** and **IPC** to work properly (`--ipc host`);
- 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

1 Introduction

2 Middleware in robotics

3 ROS 2 overview

4 Containers

5 Docker

6 Docker Compose

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** contains development containers managed by Compose.

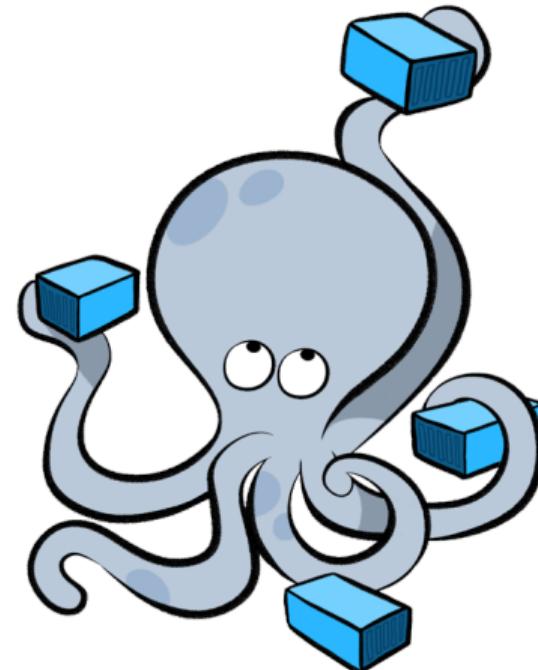


Figure 14: Docker Compose logo.

Compose files

```
1 services:
2   development:
3     build:
4       context: .
5       args:
6         TARGET: dev
7       image: devenv:latest
8       environment:
9         TERM: xterm-256color
10      network_mode: host
11      command: ["/bin/zsh"]
12      volumes:
13        - ~/.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`, and oriented only towards services specified in the local Compose file.

See the [Compose CLI reference](#) for more.