

# Roboticist 101

## Software and middleware for robotics

Roberto Masocco  
[roberto.masocco@uniroma2.it](mailto:roberto.masocco@uniroma2.it)

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

May 7, 2025



**TOR VERGATA**  
UNIVERSITÀ DEGLI STUDI DI ROMA

1 Introduction

2 Middleware in robotics

3 ROS 2 overview

# Roadmap

1 Introduction

2 Middleware in robotics

3 ROS 2 overview

# Information

- **Calendar:** from May 7 to June 12, 2025, see Teams for schedule.
- **Topics:**
  - ① Middleware for robotics and more.
  - ② Development tools for robotics (e.g., Docker).
  - ③ The MARTe2 framework for real-time control architectures in tokamaks (recorded).
- **Materials:**
  - ▶ Code repository: [github.com/ros2-examples](https://github.com/ros2-examples) (jazzy branch).
  - ▶ Lectures: [github.com/robmasocco](https://github.com/robmasocco), Teams directory (this lecture is [here](#)).
  - ▶ Textbook (if you really really need it): [my PhD thesis](#).
- **Useful background:**
  - ▶ Basics of C and Python programming, everyday Linux commands, and a Unix-like system.
  - ▶ Basics of Git workflow (check out [this tutorial](#) by Atlassian).

# Information

- **Calendar:** from May 7 to June 12, 2025, see Teams for schedule.
- **Topics:**
  - ① Middleware for robotics and more.
  - ② Development tools for robotics (e.g., Docker).
  - ③ The MARTe2 framework for real-time control architectures in tokamaks (recorded).
- **Materials:**
  - ▶ Code repository: [github.com/ros2-examples](https://github.com/ros2-examples) (jazzy branch).
  - ▶ Lectures: [github.com/robmasocco](https://github.com/robmasocco), Teams directory (this lecture is [here](#)).
  - ▶ Textbook (if you really really need it): [my PhD thesis](#).
- **Useful background:**
  - ▶ Basics of C and Python programming, everyday Linux commands, and a Unix-like system.
  - ▶ Basics of Git workflow (check out [this tutorial](#) by Atlassian).
- **Exam:** ... ask Prof. Carnevale ☺ some projects may be proposed later on.

# Program

Live lectures:

- 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 Inside the roboticist's toolbox - Linux kernel, Docker, and more

[Recorded lectures](#) (materials still on Teams, else ask me):

- ROS 2 - Sensor sampling and image processing
- Localization and mapping - From EKF to SLAM
- MARTe2 - A real-time control framework for nuclear fusion

# Survey

Please answer without fear!

Who knows what about:

# Survey

Please answer without fear!

Who knows what about:

- Version control and Git?

Please answer without fear!

Who knows what about:

- Version control and Git?
- Python?

# Survey

Please answer without fear!

Who knows what about:

- Version control and Git?
- Python?
- C/C++, from coding to linking?

# Survey

Please answer without fear!

Who knows what about:

- Version control and Git?
- Python?
- C/C++, from coding to linking?
- Object-oriented programming?

Please answer without fear!

Who knows what about:

- Version control and Git?
- Python?
- C/C++, from coding to linking?
- Object-oriented programming?
- Networking and the ISO/OSI model?

# Survey

Please answer without fear!

Who knows what about:

- Version control and Git?
- Python?
- C/C++, from coding to linking?
- Object-oriented programming?
- Networking and the ISO/OSI model?
- Extended Kalman Filter and variants?

# The roboticist

A new path for control engineers

A **control engineer** is a specialist in the **design** of controllers to drive **dynamic 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.

**A control engineer can be very effective as a roboticist.**

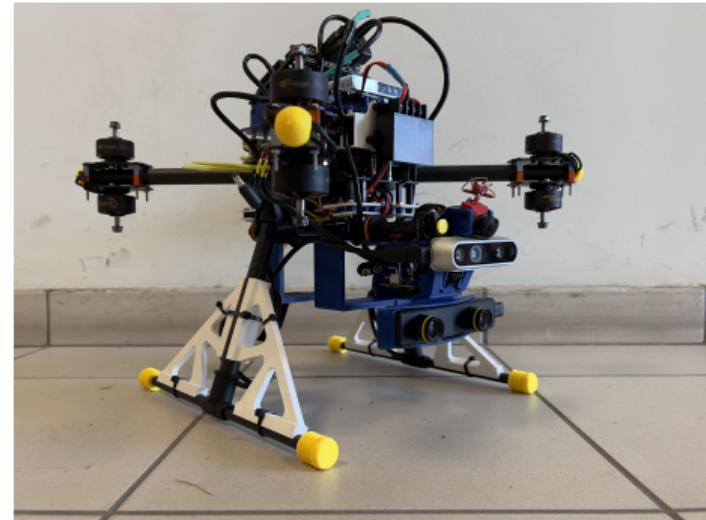


Figure 1: Stanis autonomous drone prototype.

# 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

# What is middleware?

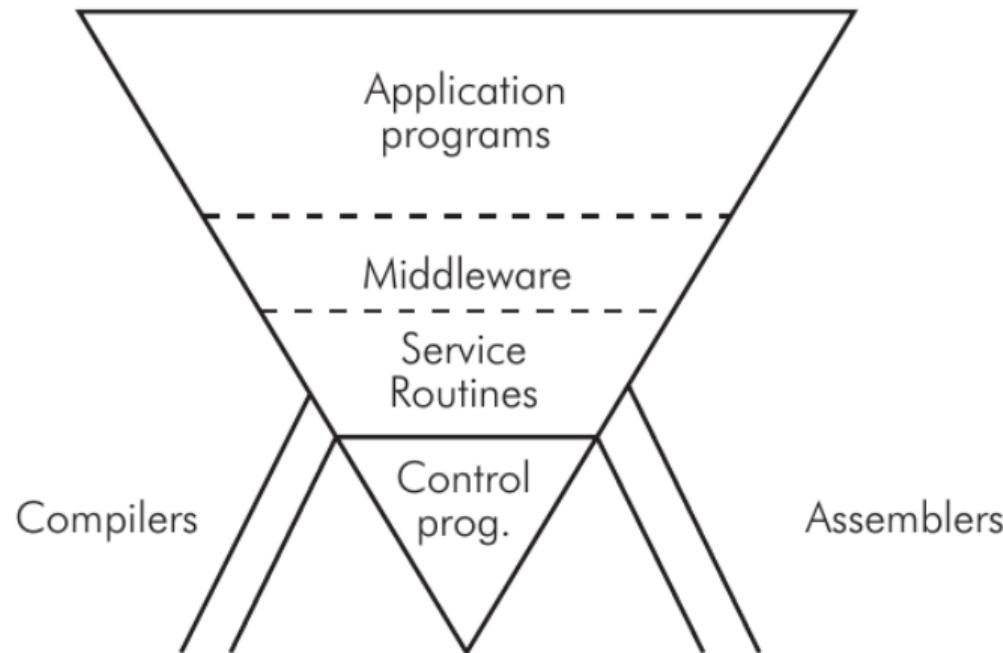


Figure 3: Software organization in a general-purpose computer system.

# 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.

Middleware is 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**.

Middleware can help to tackle and solve each one!

## Definition of DDS

The DDS is a **data-centric communication protocol** used for **distributed** software application communications, describing APIs and communication semantics between data **providers** and **consumers**.

## Definition of DDS implementation

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

It is defined by an [open standard](#) maintained by the **Object Management Group**.

# Data Distribution Service

DDSs are currently used in automotive, aerospace, military, robotics...

The open standard defines:

- **serialization** and **deserialization** of data packets over supported mediums;
- **security protocols** and cryptographic operations;
- enforcing of **Quality of Service** policies to organize transmissions (specifying things like **queue sizes**, **best-effort** or **reliable** transmissions...);
- automatic discovery of **DDS participants** (Discovery Protocol over **multicast-IP/UDP**) and transmission of data (over **unicast-IP/UDP**).

Any vendor may add their own extensions to their implementations.

# Data Distribution Service

An application using the DDS can create one or more **participants**.

They represent the access point to the DDS network, and embed a **QoS policy** specifying:

- the **domain** they belong to (numerical ID);
- the **network interfaces, protocols**, and **settings** to use.

Participants can create **entities**:

- **Publishers**, as publication entities, embedding **DataWriters**;
- **Subscribers**, as subscription entities, embedding **DataReaders**;
- **Topics**, as configuration entities with a prescribed **interface** (packet format).

Each may enforce a **QoS policy** specifying its **desired behavior**.

**Communication happens when entities with compatible policies match.**

# Data Distribution Service

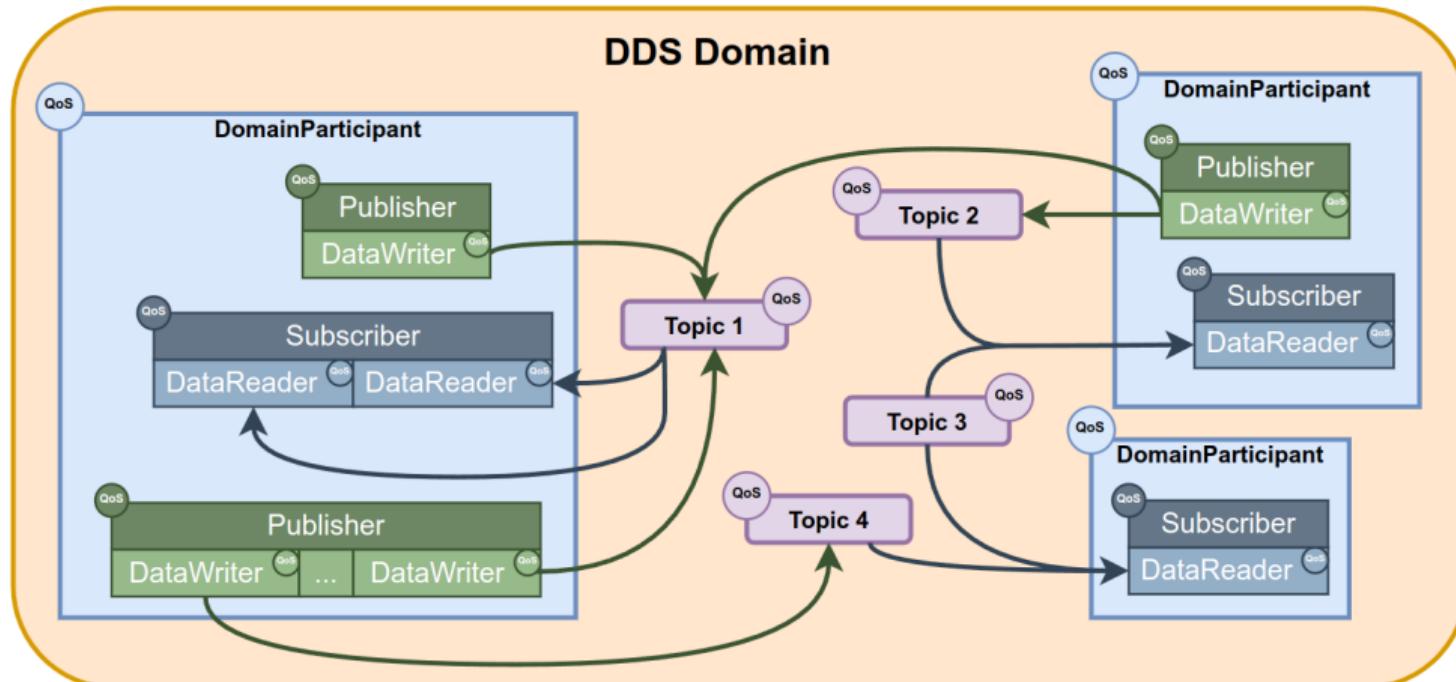


Figure 4: Scheme of a DDS-based network (data space).

# Data Distribution Service

## Peeking into the low level

Communications are implemented below the DDS layer, by the **Real-Time Publish Subscribe (RTPS)** protocol.

It defines:

- a **Discovery Protocol** to automatically find participants over a network in a same domain;
- a **Wire Protocol** to serialize and deserialize data packets;
- supported mediums and network protocols;
- data exchange semantics: publishing **changes** of a **history**.

All the entities appear to the application programmer as **objects** in the code.

# Roadmap

1 Introduction

2 Middleware in robotics

3 ROS 2 overview

# What is ROS 2?

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

Its default DDS implementation is currently **Fast DDS** from **eProsima**.



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 24.04**.

New versions are periodically released as **distributions**: the current LTS one is **Jazzy Jalisco**; 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 2005, to provide a middleware that could solve the **software integration** and **communication** problems in robotics. It has evolved much since then.

**ROS is not an operating system in the traditional sense of process management and scheduling; rather, it provides a structured communications layer above the host operating systems of a heterogenous compute cluster.**

Quigley, M. et al., **ROS: an open-source Robot**

**Operating System**, ICRA workshop on open source software, 2005



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

# Why ROS 2?

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 (for now!): **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).

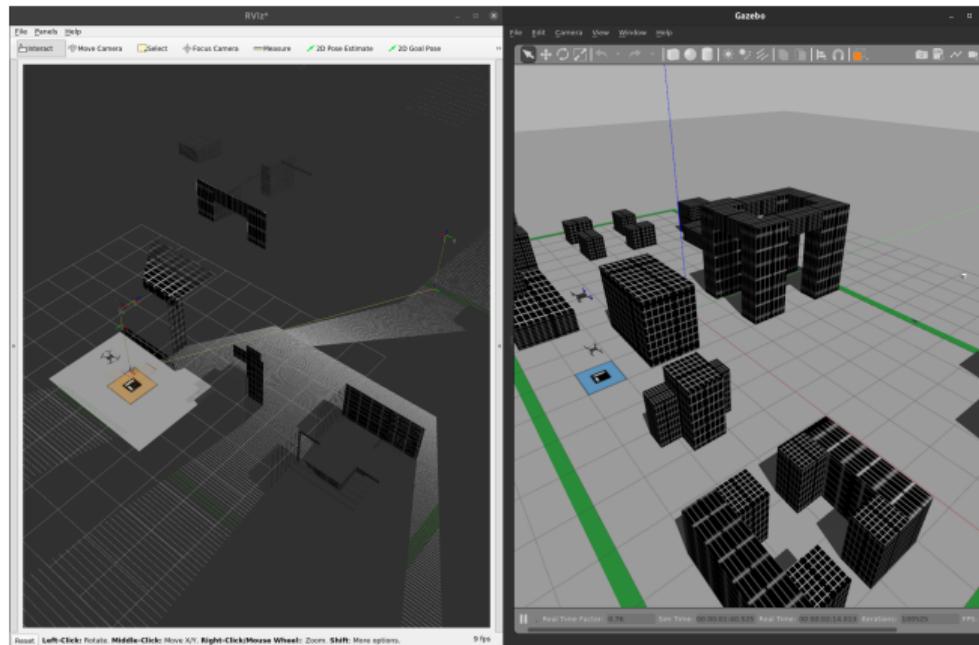


Figure 7: Simulated drone in Gazebo Classic and RViz.

# The power of open source

The ROS project aims at establishing a common ground for the **development** of robotics just like what happened in the **software industry** in the last 30 years.

By **reusing code** and implementing **open hardware**, **building** a functioning robot may be only a matter of installing software, then writing four kinds of **text files**:

- **system files**, for onboard computers;
- **launch files**, to configure software startup and operation;
- **parameter files**, to set configuration parameters for software modules;
- **robot description files**, to tell the software how the robot is built;

even without a single line of code!

Open source enables quick **diffusion** and **improvement**, even on the **security** side, although new threats are rising: see the recent [xz backdoor](#) case.

# 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) (for now!).
- 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. The entire ROS 2 codebase is on [GitHub](#).

Keep this in mind during debugging, or while looking for information on an API!

# 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 RMW 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 **ROS-workload**.

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

# Executors

## Handling events and callbacks

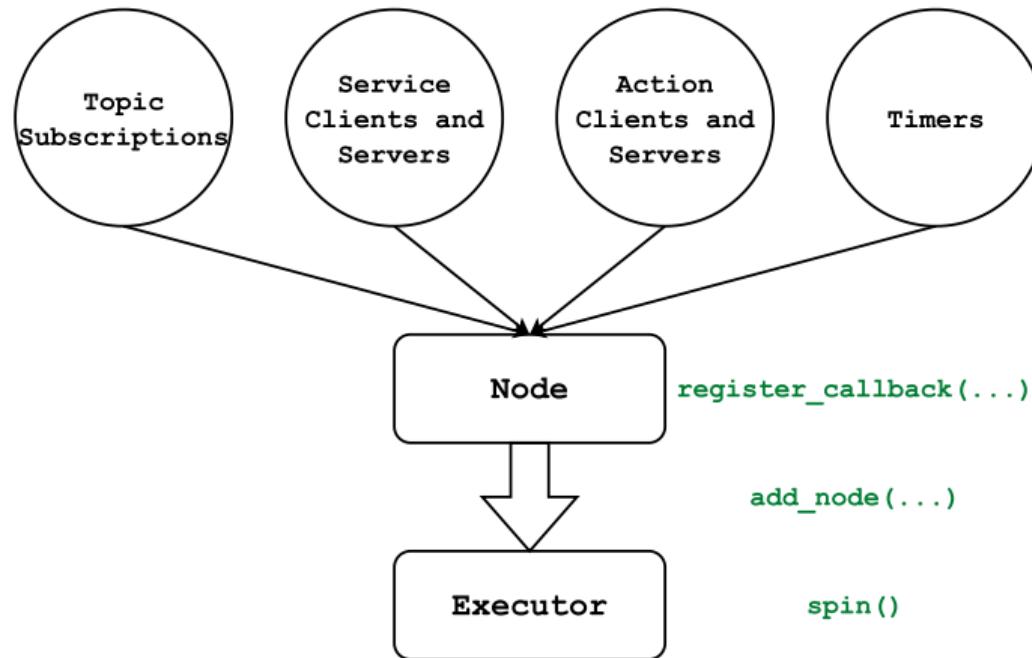


Figure 8: ROS 2 event-based programming paradigm.

# 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**.

# Flaws

The devil is in the details

## ROS 2 main design flaws as of today

The main concerns arise when developing low-level stuff:

- **the DDS/RMW layer is almost completely abstracted**, so non-standard network or DDS 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.

# Flaws

The devil is in the details

## ROS 2 main design flaws as of today

The main concerns arise when developing low-level stuff:

- **the DDS/RMW layer is almost completely abstracted**, so non-standard network or DDS 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?

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

# Flaws

The DDS layer abstraction

ROS 2 nodes can either **publish to** or **subscribe to a topic**.

## Definition of ROS 2 topic

A ROS 2 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 **single QoS policy** that specifies how transmissions should be performed.

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

Compare this to the DDS way of setting up a communication channel between two applications...

# Flaws

## The DDS in a multi-agent environment

Over the years, ROS 2 helped identify some **flaws** in the **DDS standard**:

- communication with **large data** (e.g., images, pointclouds...) over a **lossy network** becomes inefficient or impossible, due to **retransmission policies**;
- **discovery** of **many endpoints**, especially over a **lossy network**, may become slow or **clog** the network completely, due to the amount of generated **traffic**.

Just think about a **swarm** of drones, each one cameras and the like, trying to communicate over a **wireless network**...

# Flaws

Abandoning the DDS?

ROS 2 steering committees are considering the **abandonment** of the DDS middleware in favor of **new solutions**. The main requirements are:

- **scalability**, especially with entity queries;
- **low latency**, on constrained hardware and low-power networks;
- **configurability**;
- **optimization of data transfers**.

The newest LTS version, **Jazzy Jalisco**, is the first one to ship with a RMW based on the **Zenoh** middleware.



Figure 9: Zenoh logo.

**Zenoh** is a new middleware for distributed communication meant as a **successor of the DDS**.

It is designed to handle **three main types of data**:

- **data in motion** that is being transmitted over the network;
- **data at rest** that is stored in a device and can be retrieved at any time;
- **data for computations** as in remote procedure calls.

It relies on a **router daemon** (`zenohd`).



Figure 10: Zenoh logo.

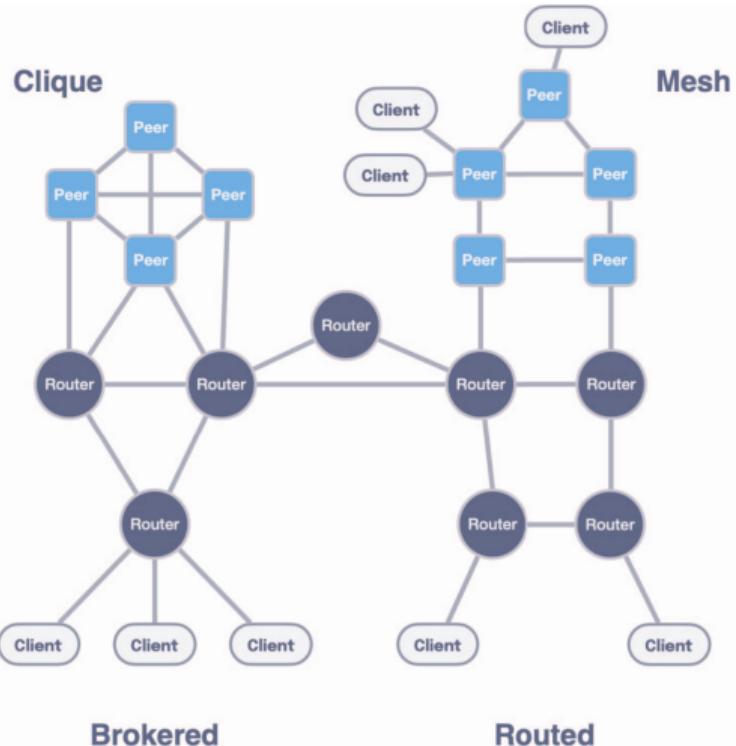


Figure 11: Possible network configurations in Zenoh.