Robot Operating System 2

Lecture 1: Middleware Fundamentals

Roberto Masocco

roberto.masocco@uniroma2.it

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

April 27, 2022

Roadmap

Roadmap

What is middleware?

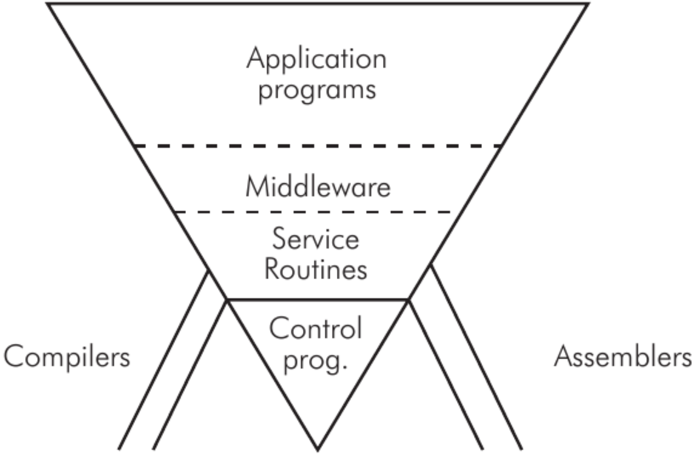

Figure 1: Software organization in a generic computer system

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** (not only microcontrollers!);
- **software** organization and maintenance;
- **communication** (involves both hardware and software!);
- debugging and **testing**.

Middlewares can help to tackle and solve each one!

Data Distribution Service

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;
- **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**).

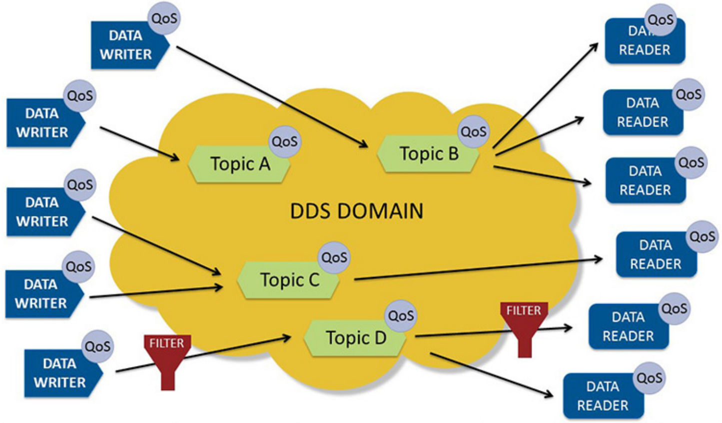

Figure 2: Scheme of a DDS-based network

Data Distribution Service

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

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 3: ROS 2 logo

What is ROS 2?

ROS 2 currently supports **C++** and **Python** for application programming, and runs natively on **Ubuntu Linux 20.04**.

New versions are periodically released as **distributions**: the current LTS one is **Foxy Fitzroy** and the latest one today is **Galactic Geochelone**; the development version is **Rolling Ridley** and can only be compiled from source.

Figure 3: ROS 2 logo

Why ROS 2?

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

Figure 4: 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

- integration with **simulators** (e.g. Gazebo) and **visualizers** (e.g. RViz).

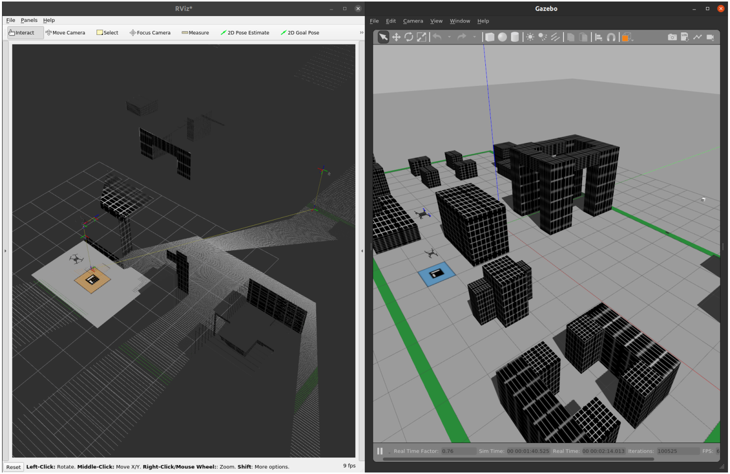

Figure 5: Simulated drone in Gazebo and RViz (credit: Lorenzo Bianchi)

Flaws

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**.

Flaws

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**.

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

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

Job Executors: Events and Callbacks

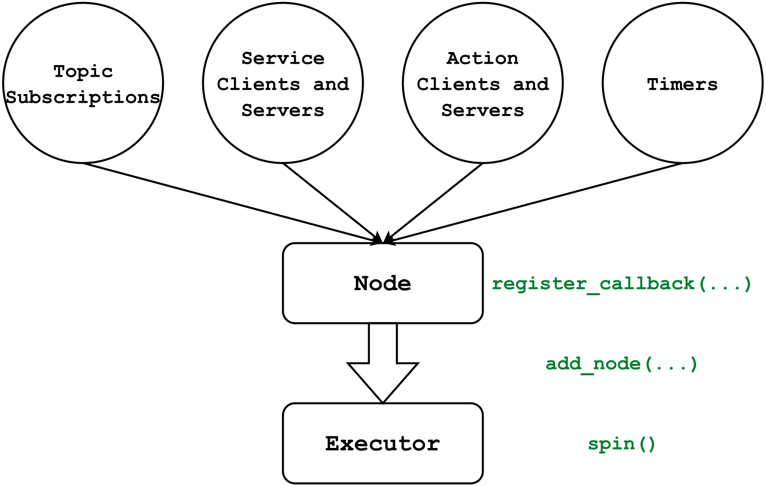

Figure 6: ROS 2 event-based programming paradigm

Job Executors: Events and Callbacks

1. Middleware functionalities trigger **(a)synchronous events**.
2. Events are handled by **background jobs**, coded in **callbacks**.
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** scheduling policy that **always prioritizes timers**.

Executors are currently being redesigned, importing changes from the priority-based **rclc Executor** of **micro-ROS**.

Roadmap

ROS 2 Messages

A message is a **single DDS data packet** sent over a **topic**, from **publisher nodes** to **subscriber nodes**, with a specific **QoS policy**.

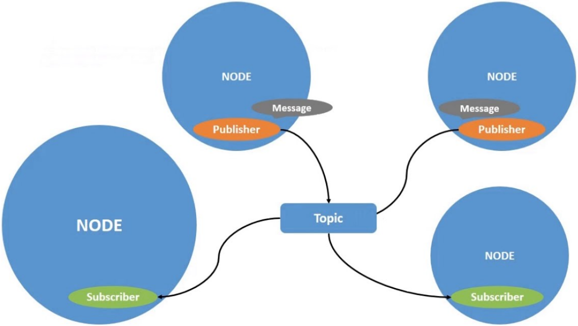

Figure 7: Example of a topic with multiple publisher and subscriber nodes

Interface Files - Messages

Interface files format is specified by the DDS, with data types resolved to machine types according to the platform being used[1].
Message file names end with `.msg`.
Things start very simply...

```
int64 data
```

Listing 1: Definition of the `std_msgs/msg/Int64` message

```
string data
```

Listing 2: Definition of the `std_msgs/msg/String` message

[1] About ROS Interfaces (ROS 2 Galactic docs)

Interface Files - Messages

… then escalate quickly!

```
std_msgs/Header header

uint32 height
uint32 width

string encoding

uint8 is_bigendian
uint32 step
uint8[] data
```

Listing 3: Definition of the `sensor_msgs/msg/Image` composite message

Interface Files - Messages

Special values (i.e. *constants*) may be specified.

```
int64 MYNUM=1 # Must be of compatible type

int64 number
```

Listing 4: Definition of an example message with a constant value

They are not bound to any field and will appear as special selectable values in the generated C++/Python libraries.
ROS 2 adds its own guidelines[2], and installed interfaces can be inspected with the `ros2 interface show` command.

[2] ros2-examples/interfaces.md

Message Topics - Quality of Service

A **QoS policy**[3] for publishers/subscribers has the following attributes:

- **History** (*keep last N* or *all*)
- **Depth** (queue size *N*)
- **Reliability** (*best-effort* or *reliable*, default: reliable)
- **Durability** (publishers resend all messages to "late-joiners")
- Deadline
- **Lifespan** (message expiration date)
- Liveliness
- Lease Duration

Default **profiles** are available (e.g. **Sensor data**, **Service**...).

[3] About QoS Settings (ROS 2 Galactic docs)

C++ Fundamentals

Dust off your C programming skills, then add:

Object-Oriented Programming

```cpp
class MyClass : public ParentClass
{
public:
  MyClass();
  // ...
protected:
  // ...
private:
  // ...
};
```

Listing 5: Example of definition of a C++ class

Namespaces

Subdivision of the global namespace to avoid naming collisions between multiple libraries, resolved with the `::` operator.

```
MyLib::foo();
MyClass::foo();
// Completely different names for the compiler!
```

Listing 6: Example of namespaces usage

Names may become very long, so usually they are hidden with `typedef`.

C++ Fundamentals

Templates

Classes or functions whose implementation depends on some data type. When instantiated or called with a specific type, the corresponding code is generated by the compiler.

```
std::vector<int> int_vector;
std::vector<double> double_vector;
```

Listing 7: Example of objects of the template class `std::vector`

These too make names very long, so are usually typedef'd.

Shared Pointers

A kind of **smart pointer** (there are also `unique` and `weak`) that also holds an **usage counter**, incremented by every function or object that is handling the pointer. **When the `shared_ptr` is destroyed, if the counter is zero the pointed object is also destroyed and its memory deallocated.**

```
{
  // A new scope starts here
  std::shared_ptr<rclcpp::Node> node =
    std::make_shared<rclcpp::Node>("my_node");
}
// Here the node pointer has been destroyed!
```

Listing 8: Example of shared pointer creation

Obviously `std::shared_ptr` is a template class. ROS 2 heavily relies on them, and the `SharedPtr` alias is frequently defined.

Example: Topic Pub/Sub

Example packages and additional materials are available on GitHub.

[IntelligentSystemsLabUTV/ros2-examples]

Now go have a look at the [ros2-examples/src/topic_pubsub] package!