

ROS 2

Advanced communication

Roberto Masocco

`roberto.masocco@uniroma2.it`

University of Rome Tor Vergata

Department of Civil Engineering and Computer Science Engineering

May 24, 2023



TOR VERGATA
UNIVERSITY OF ROME

School of Engineering

Recap

ROS 2 software is organized in **packages**, built by **colcon** calling either **CMake** or **setuptools**.

Messages are the most basic communication paradigm, entirely built on DDS layer communication APIs.

Messages formats are defined in **interface files** which usually constitute entire packages.

Updates

- **New code examples available.**
- Revised **lectures program**.
- Follow-up on **message topics code examples**:
 - ▶ Subscriber.
 - ▶ CLI inspection tools.
 - ▶ Interface packages and the [custom_topic_cpp](#) example.
 - ▶ [resetting_sub](#) example.

Roadmap

1 Asynchronous I/O

2 Services

3 Actions

Roadmap

1 Asynchronous I/O

2 Services

3 Actions

What is I/O?

Informal definition

In an **operating system**, a **task** (specifically, a **thread**) can perform operations pertaining to these two broad families:

- execute **computations** (e.g., $1 + 1 = 2$), using regular **CPU** instructions;
- access **system resources** (both **hardware** and **software**) through calls to the **kernel** (i.e., **system calls**), **exchanging data** in both directions.

When these resources are not part of the OS, but rather the OS enables tasks to interface¹ with them, we talk about **I/O** (*Input/Output*).

OS schedulers typically distinguish between **CPU-bound** and **I/O-bound** tasks, because of their different **execution patterns**.

¹Drivers, protocols, software stacks...

Blocking I/O

What the OS likes the most

The most common execution pattern for a task that performs an I/O system call goes like this:

- ① prepare the **input data** for the system call;
- ② call an **API** that performs the system call;
- ③ the OS **blocks the task**, which is **waiting** for the operations to complete;
- ④ the OS **returns control** to the task when the system call is completed;
- ⑤ **output data**, returned by the kernel, can be accessed by the task.

This is **blocking I/O**, because the task is **blocked** while waiting for the system call to complete.

Examples of **blocking calls**: read, write to **file descriptors**.

Non-blocking I/O

What userspace application like the most

If the kernel supports this feature, a task can perform a **non-blocking system call**:

- 1 prepare the **input data** for the system call;
- 2 call an **API** that performs the system call;
- 3 the OS **returns control** to the task **immediately**, without blocking it;
- 4 the task can **poll** the system call **status** to check if it is completed;
- 5 when the system call is completed, the task can **access the output data**;
- 6 optionally, a **callback** routine can be registered to be executed right when the system call is completed.

This is **non-blocking I/O** (or *asynchronous I/O*, or *overlapped I/O*), because the task is **not blocked** while waiting for the system call to complete, and things can happen in between.

Examples of **non-blocking calls**: read, write to **sockets** configured appropriately.

Non-blocking I/O

What userspace application like the most

Usually, the operation status can be inspected through some kind of **handle object** returned by the API.

Some **programming languages** implement **future objects**: datatypes that hold the result of an asynchronous operation, which can be inspected to check if the operation is completed, and to retrieve the result once it is; **they are said to hold a value only when the operation is completed.**

ROS 2 makes a heavy use of **callbacks** and **future objects** to handle **asynchronous I/O**.

Roadmap

1 Asynchronous I/O

2 Services

3 Actions

ROS 2 services

Basic client-server paradigm

ROS 2 extends the basic DDS messages adding two more **communication paradigms**: the first is the **service**. It allows nodes to establish quick and simple **client-server** communications.

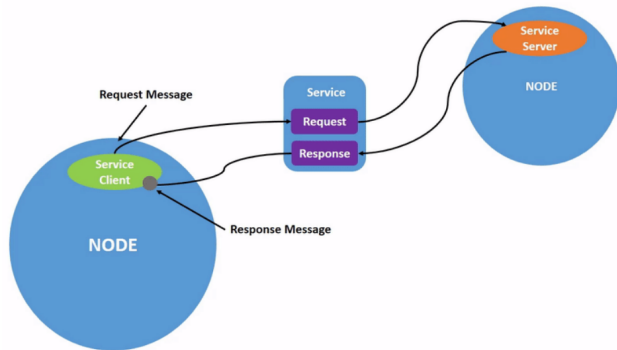


Figure 1: Two nodes acting as service *client* and *server*.

In actual ROS 2 applications:

- ➊ The **client** sends a **request message** to the server.
- ➋ The **server** receives the request and processes it.
- ➌ Meanwhile, the **client** can either **block** waiting for the response or **synchronously poll** it.
- ➍ When done, the **server** sends a **response message** to the client.
- ➎ If waiting, the **client** awakes when receiving the response.

ROS 2 services

Coding hints for servers and clients

Servers

Similarly to topic subscriptions, requests are processed in appropriate **callbacks**, taking **two arguments**, in which responses are also populated. The server object is as well only needed to instantiate the service.

Clients

As per the previous dynamics, one has to **code each step** of the client side into their application using appropriate **ROS 2 APIs**. **The client object is used to send requests**, while **responses are handled as future objects^a**.

^a[std::future - C++ Reference](#)

Interface files

Services

The entire system is built on messages, so **combine two of them** in a single interface file, separated by ---.

Service file names end with `.srv`.

```
1 # REQUEST
2 int64 a
3 int64 b
4 ---
5 # RESPONSE
6 int64 sum
```

Listing 1: Definition of the `example_interfaces/srv/AddTwoInts` service.

Example

Simple service

Now go have a look at the [ros2-examples/src/cpp/simple_service_cpp](https://github.com/ros2/examples/tree/main/src/cpp/simple_service_cpp) package!

Roadmap

1 Asynchronous I/O

2 Services

3 Actions

Limitations of services

The third paradigm exists because services rely on the following **restrictive assumptions**.

Services implementation assumptions

- Since the client may block for the entire duration of the request processing, **server computations should be short and always produce some result** (e.g., even an error must be a result, but **we** have to encode it).
- Service calls are finished only when the response has been received, *i.e.*, **if either the client or the server crash, the behaviour of the other one is undefined** (no **state machine**! Say hello to **deadlocks**, crashes...).
- Once a service is called, **the request may never be interrupted**.

These make operations that **must be requested** and **take a long time** (for CPUs!) completely unfeasible.

Think of real stuff such as **movement**, **navigation**...

ROS 2 actions

Full client-server paradigm

Built on services and message topics, they **decouple computations from middleware APIs**, thanks to three concepts that embody the **three stages of the communication**:

- ➊ **Goal**: the full request of the operation to be executed.
- ➋ **Feedback**: intermediate results and information about the ongoing processing.
- ➌ **Result**: the final result of the requested operation.

Their implementation is still a bit cumbersome because of the **many different data types** (classes) involved, and is found in the [rclcpp_action](#) and [rclpy_action](#) libraries.

They are **extensively used for robot navigation and movement**.

ROS 2 actions

Full client-server paradigm

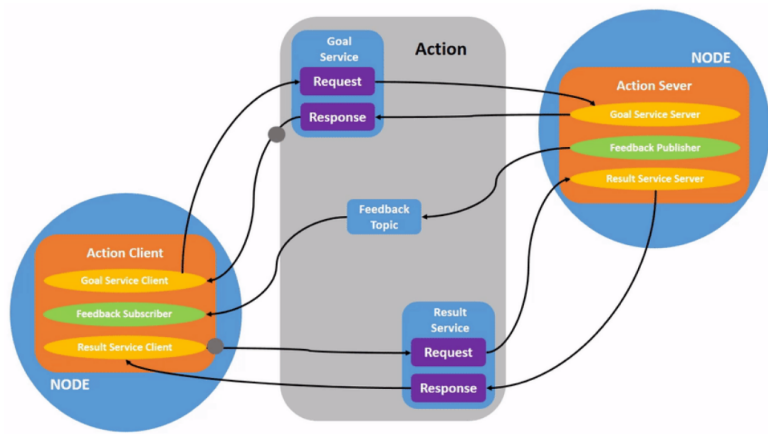


Figure 2: Example of an action server and *client*.

ROS 2 actions

The goal state machine

Goal State Machine

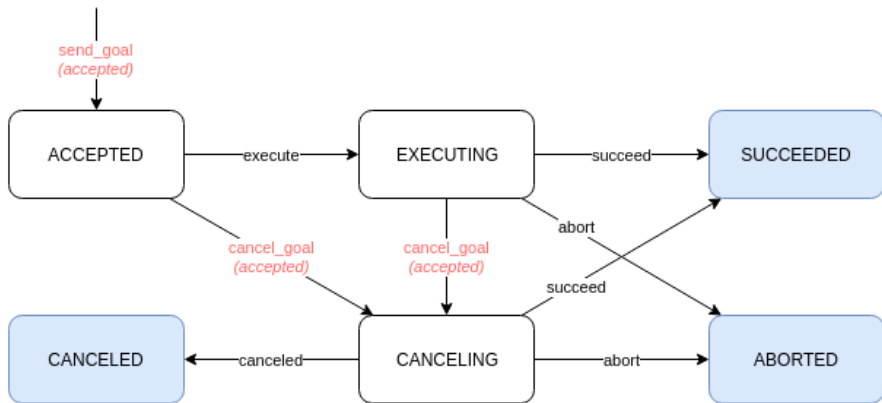
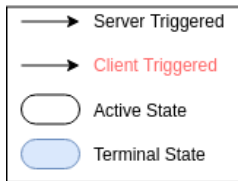


Figure 3: State machine² of an action goal, implemented and managed internally by ROS 2.

²[Actions - ROS 2 Design](#)

ROS 2 actions

Communication overview

In actual ROS 2 applications, the **client** requests the completion of some **goal** to the **server**. The middleware only offers APIs to **notify the state of the goal** between the two.

- ➊ The **client** sends a **goal service request** to the server.
- ➋ The **server** may **accept** or **reject** the goal request.
- ➌ Server computations are usually started when the goal is **executed**: the middleware only keeps track the state of the goal, its updates and the rest are up to the developer.
- ➍ The **client may cancel** the goal request; the **server may abort** the goal request; intermediate results and information, if any, are published by the server on the **feedback topic**.
- ➎ The **client** asks the server for the final result over the **result service**.

ROS 2 actions

Coding hints for servers and clients

Servers

Goal requests are handled with **callbacks**, while computations can be handled freely (usually in **separate threads**). When done, the goal must be marked as **succeeded** or **aborted**.

Clients

Similarly to services, much is done with **future objects**, but **callbacks** must be defined to handle **goal**, **result** and **cancellation responses**, and **feedbacks**.

Handling all possible scenarios for a goal results in the **longest and most complicated code that a ROS 2 application may ever require.** 😊

Interface files

Actions

Combine **three messages** in a single interface file, separated by ---.
Action file names end with `.action`.

```
1 # GOAL
2 int32 order
3 ---
4 # RESULT
5 int32[] sequence
6 ---
7 # FEEDBACK
8 int32[] partial_sequence
```

Listing 2: Definition of the `ros2_examples_interfaces/action/Fibonacci` action.

Example

Fibonacci computer

Now go have a look at the [ros2-examples/src/cpp/actions_example_cpp](#) package!