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

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.

This lecture is here.

Recap

Updates

- **New code examples available, also in Python**.
- 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

Roadmap

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 interface[1] 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**.

[1] Drivers, 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

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.

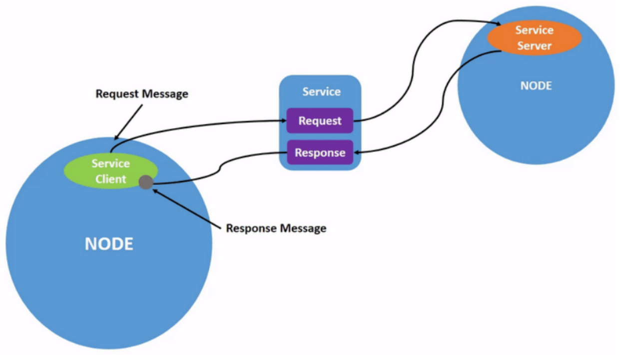

Figure 1: Two nodes acting as service *client* and *server*.

ROS 2 services

Communication overview

In actual ROS 2 applications:

1. The **client** sends a **request message** to the server.
2. The **server** receives the request and processes it.
3. Meanwhile, the **client** can either **block** waiting for the response or **synchronously poll** it.
4. When done, the **server** sends a **response message** to the client.
5. If waiting, the **client** awakes when receiving the reponse.

The main command is `ros2 service` with the following verbs:

- `list` Lists all active services.
- `type` Prints the service type.
- `find` Lists active services of the given type.
- `call` Calls the service with the request defined in the command line.

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

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`.

```
# REQUEST
int64 a
int64 b
---
# RESPONSE
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` package!

Roadmap

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**:

1. **Goal**: the full request of the operation to be executed.
2. **Feedback**: intermediate results and information about the ongoing processing.
3. **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

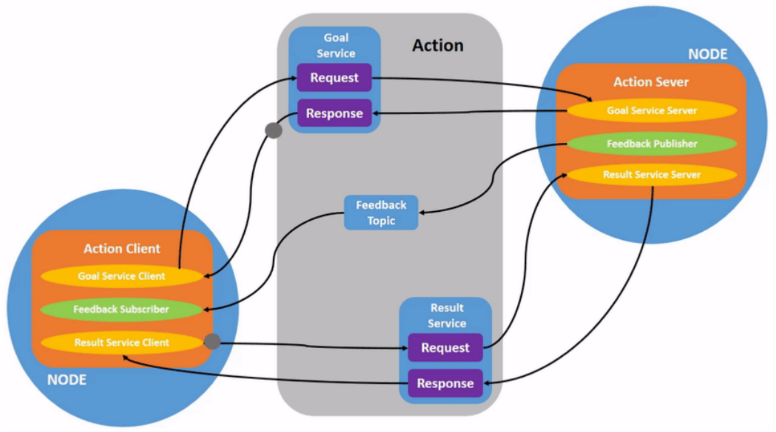

Figure 2: Example of an action *server* and *client*.

ROS 2 actions

The goal state machine

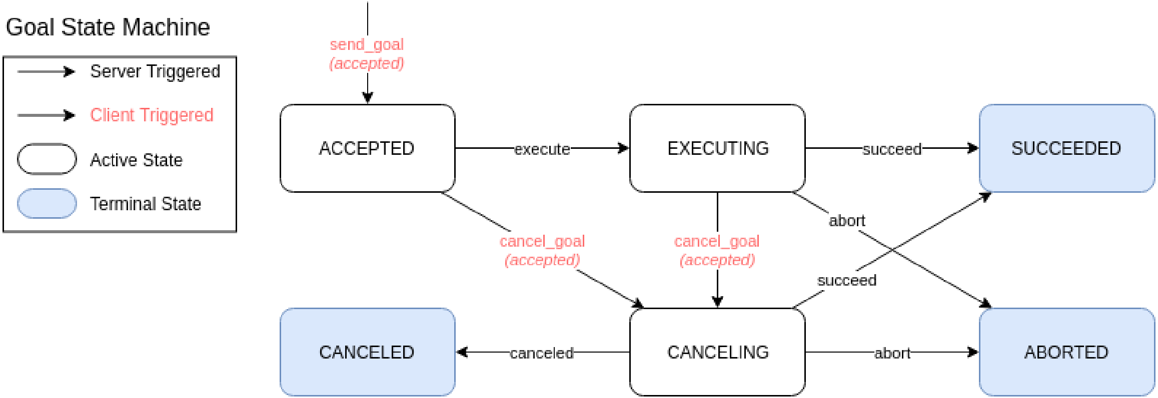

Figure 3: State machine[2] of an action goal, implemented and managed internally by ROS 2.

[2] Actions - ROS 2 Design

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.

1. The **client** sends a **goal service request** to the server.
2. The **server** may **accept** or **reject** the goal request.
3. 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.
4. 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**.
5. The **client** asks the server for the final result over the **result service**.

The main command is `ros2 action` with the following verbs:

- `list` Lists all active actions.
- `info` Prints information about an action.
- `send_goal` Sends a goal request to an action server, and prints the result; with `-f` prints also feedback messages.

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`.

```
# GOAL
int32 order
---
# RESULT
int32[] sequence
---
# FEEDBACK
int32[] partial_sequence
```

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

Now go have a look at the `ros2-examples/src/cpp/actions_example_cpp` package!

If you're curious, the `ros2-examples/src/cpp/advanced/complete_actions_cpp` package, which implements the complete goal state machine using a multithreaded executor.

Exercises

- Run the service client and server examples, and try to call the service from the command line.
- Create two new packages for server and client nodes, and for a custom service definition named `CapString.srv`; the server should take a string as input and return two strings: the same string fully capitalized, and the number of characters in the string.

Exercises

- Run the action client and server examples, and try to call the action from the command line.
- Modify the feedback message: instead of a partial sequence, it should publish the length of the sequence so far; this requires:
  1. modifying the action definition in `ros2_examples_interfaces`;
  2. modifying the server node to publish the length of the sequence instead of the partial sequence in the feedback message (hint: use methods of the `std::vector` class to get the length of the partial sequence in one go);
  3. modifying the feedback callback in the client to parse and print the length from the feedback message.