ROS 2

Advanced communication II

Roberto Masocco

roberto.masocco@uniroma2.it

University of Rome Tor Vergata

Department of Civil Engineering and Computer Science Engineering

May 16, 2024

TOR VERGATA
UNIVERSITY OF ROME
School of Engineering

Recap

Messages are the most basic, **one-way** communication paradigm.

QoS policies and other topic settings affect communication behavior among entities.

Services are a simple implementation of a **two-way**, **client-server** communication paradigm.

This lecture is here.

Recap

Updates

- Follow-up on **message topics code examples**:
  - `resetting_sub` example.

Roadmap

1. Interface packages

2. Actions

Roadmap

1. Interface packages
2. Actions

Roadmap

1. Interface packages

2. 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 the server crashes before sending the response, the client's behavior is undefined, especially when it is blocking** (no **state machine**! Say hello to **deadlocks**, crashes...).
- Once a service is called, **the processing of 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**...

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, as well as many **API inconsistencies**.
It is found in the `rclcpp_action` and `rclpy_action` libraries.

They are **extensively used to implement robot navigation and movement**, and many more complex real-world operations, providing a **software abstraction** for them.

ROS 2 actions

Full client-server paradigm

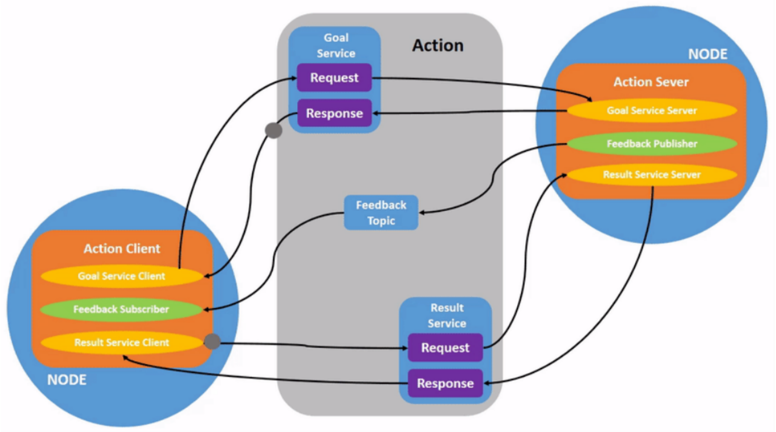

Figure 1: Example of an action *server* and *client*.

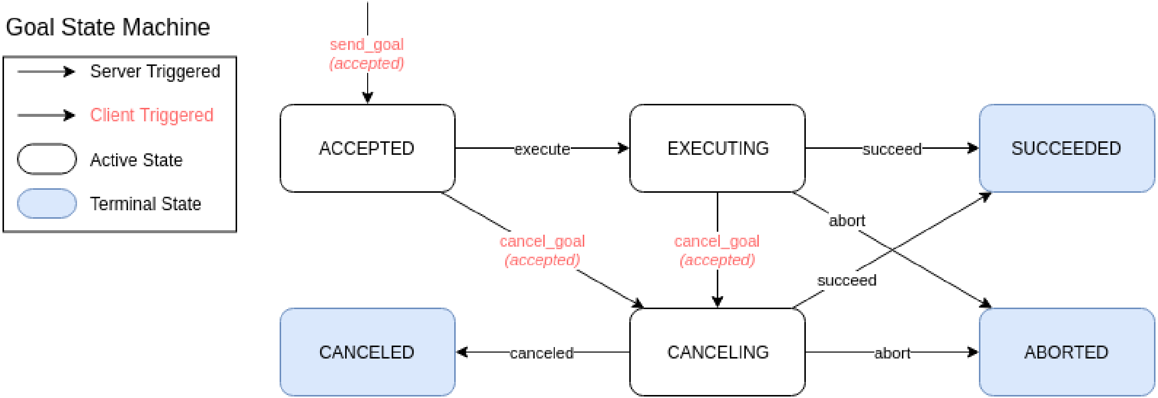

Figure 2: State machine[1] of an action goal, implemented and managed internally by ROS 2.

[1] 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.

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 done in **separate threads**).
Cancellation requests are handled with **callbacks** and can be **polled** during the computation.
When done, the goal must be marked as **succeeded** or **aborted**.

Clients

Similarly to services, much is done with `future` **objects**, but **callbacks may 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**. ☺

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 1: 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 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.