Robot Operating System 2

Lecture 2: Advanced Communication

Roberto Masocco
roberto.masocco@uniroma2.it

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

April 28, 2022

TOR VERGATA
UNIVERSITY OF ROME
School of Engineering

Recap

ROS 2 is a **DDS**-based, open-source middleware for robotics software development.
Messages are the most basic communication paradigm, entirely built on DDS layer communication APIs.

Follow-up

- ROS 2 **installation**;
- colcon **workspace structure**;
- interface libraries **namespaces**.

Roadmap

Roadmap

1. Services
2. Actions

ROS 2 Services

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

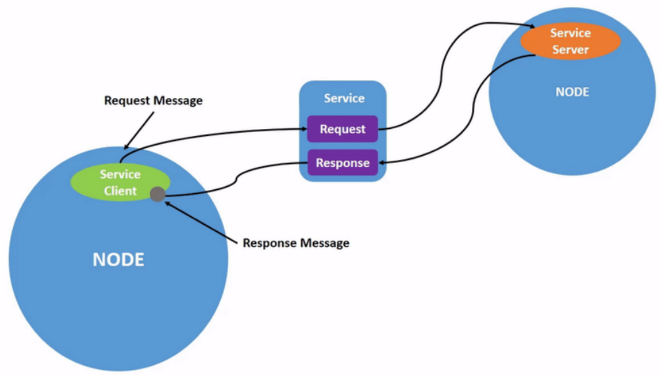

Figure 1: Two nodes acting as service *client* and *server*

ROS 2 Services

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.

Coding Servers and Clients

Servers

Similarly to topic subscriptions, requests are processed in appropriate **callbacks**, 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] (related to *asynchronous I/O*, no further details required).

[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`.

```
# 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/simple_service package!

Roadmap

1. Services
2. Actions

Services Limitations

The third paradigm exists because services implementation relies 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).
- 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** (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 practically-interesting stuff such as **movement**, **navigation**...

ROS 2 Actions

Built on services and message topics, they **decouple computations from middleware tasks**, thanks to the concepts of **goal**, **feedback** and **result**. Their implementation is still a bit cumbersome, but are **extensively used for robot navigation and movement**.

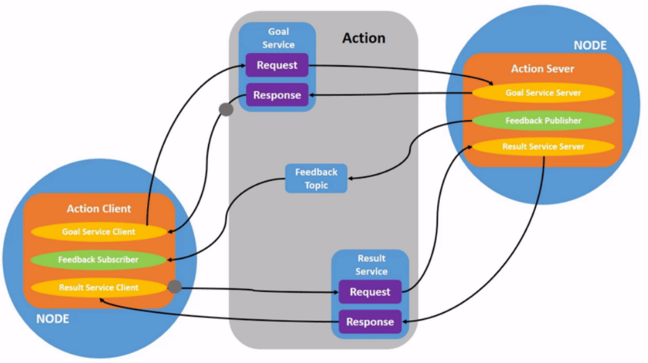

Figure 2: Example of an *action server* and *client*

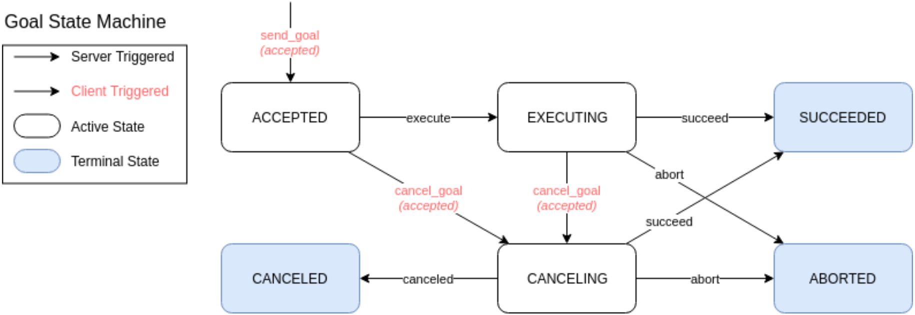

Figure 3: State machine[1] of an action goal, managed by the middleware

[1] Actions - ROS 2 Design

ROS 2 Actions

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**.

Coding Action 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**, which triggers another callback to contact the client's result service.

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`.
There are no example packages for actions as of today.

```
# GOAL
int32 order
---
# RESULT
int32[] sequence
---
# FEEDBACK
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/actions_example package!