ROS 2

Advanced communication I

Roberto Masocco
roberto.masocco@uniroma2.it

University of Rome Tor Vergata
Department of Civil Engineering and Computer Science Engineering

May 14, 2024

TOR VERGATA
UNIVERSITY OF ROME
School of Engineering

Recap

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

Messages are the most basic, **one-way** communication paradigm.
On the **DDS RMW**, ROS 2 topics directly resolve to **DDS topics**.

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

This lecture is here.

Recap

News

- Updated **lectures program**.
- Follow-up on **message topics code examples**:
  - ros2-examples/src/cpp/topic_pubsub_cpp
  - CLI inspection tools.

Recap

Updated lectures program

1. Roboticist 101 - Software and middleware for robotics
2. ROS 2 - Workflow and basic communication
3. **ROS 2 - Advanced communication I**
4. **ROS 2 - Advanced communication II**
5. ROS 2 - Node configuration
6. ROS 2 - Sensor sampling and image processing
7. Localization and mapping - From EKF to SLAM
8. Inside the roboticist's toolbox - Linux kernel, Docker, and more
9. microROS - Bridging the gap
10. MARTe2 - A real-time control framework for nuclear fusion

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 applications 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 userspace **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 applications 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

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.

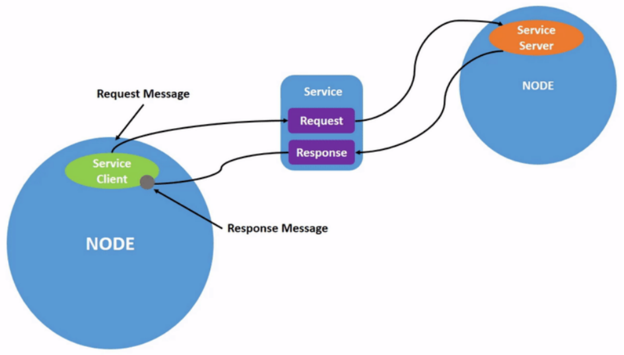

Figure 1: Two nodes acting as service *client* and *server*.

In actual ROS 2 applications:

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 response.

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

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/cpp/simple_service_cpp` package!

Exercises

- Add a timer to the subscriber in the `topic_pubsub_cpp` package to periodically toggle the ROS 2 subscriber on and off; how would you do that?
  (Solution: `resetting_sub` example.)
- 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.