ROS 2

Node configuration

Roberto Masocco
roberto.masocco@uniroma2.it

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

May 21, 2024

TOR VERGATA
UNIVERSITY OF ROME
School of Engineering

Recap

ROS 2 nodes can communicate using three different **paradigms**:

- **Topics**: asynchronous, unidirectional communication.
- **Services**: stateless, bidirectional communication.
- **Actions**: stateful, bidirectional communication.

All rely on **messages**, which must be **defined**, and on **QoS policies**.

Upcoming lectures will focus on **real applications** of these tools.
Check Teams channel for **schedule updates**!

New code examples are available.

This lecture is here.

Roadmap

Roadmap

Why namespaces?

Example: Two driver nodes

- **Two sensor driver nodes** must publish on a same `sensor` topic.
- By default, the two nodes have the **same name** since it is hardcoded.
- The two driver nodes are **not connected to the same sensor**, but to two distinct ones of the same kind.

Why namespaces?

Example: Two robots

- Two distinct robots are **deployed in the same network**.
- The two robots have **the same hardware**, and must perform **the same tasks**.
- Thus, the two robots have **the same software architecture**.

Why namespaces?

Just to name a few of the **collisions** that may arise in the previous scenarios:

- **node names** must be unique to each node, otherwise the **node graph** will be **ambiguous**, and **undefined behavior** may occur;
- **topic names** may not be unique, but they must allow to **distinguish different entities**;
- **service names** as above;
- **action names** as above.

We need a way to **group entity names**. That's what **namespaces** are for.

Defining ROS 2 namespaces

The hierarchical structure

Remember the ~/ in topic names? That sequence identifies the **namespace** of the entity. **Entity names** follow the same **hierarchical structure** of **UNIX file system paths**:

`/<namespace>/<node_name>/<entity_name>` where:

- `<namespace>` is the **namespace** of the entity, and can be made of **multiple nested levels**;
- `<node_name>` is the **name of the node** that **owns** the entity;
- `<entity_name>` is the **name of the entity**: topic, service, action, parameter...

Thus, while coding, prepending ~/ in front of the entity name will **automatically prepend the namespace and owner node name** upon name resolution.

Defining ROS 2 namespaces
Coding tips

While coding, prepending `~/` in front of the entity name will **automatically prepend the namespace and owner node name** upon name resolution. This way, you do not have to worry about namespaces while **coding** your nodes, only when **launching** them.
For example, assuming we are working on a node named `my_node`:

```
this->create_publisher<String>("~/my_topic");
```

creates a publisher whose topic name may be resolved to either:

- `"/my_node/my_topic"` if the node is launched without a namespace;
- `"/my_namespace/my_node/my_topic"` if the node is launched with the `my_namespace` namespace.

The same holds for **services** and **actions**. For **nodes** and **parameters** this is **automatically done** by the middleware and you must only specify the name of the entity upon creation.

Remapping ROS 2 entity names

CLI commands

Names can be **specified** or **overridden** when starting applications with `ros2 run`:

```
ros2 run PACKAGE_NAME EXECUTABLE_NAME --ros-args -r <old_name>:=<new_name>
```

where `<old_name>` can be one of:

- `__ns` to remap the **namespace**;
- `__node` to remap the **node name**;
- the actual name of a **topic**, **service** or **action** that you want to remap.

Multiple remappings can be specified by **repeating** the `-r` option.

Roadmap

Why parameters?

Example: The camera driver node

Suppose you have to integrate an **RGB camera** into your architecture, by writing a ROS 2 node that acts as a **driver**:

- the node uses the necesary libraries to interact with the camera hardware;
- RGB frames are constantly published on some topic;
- during constant operation, you would like to **change some values** to tune image quality, *e.g.*, exposure.

You could **encode** such parameters in your program, or pass them as **command-line arguments**, but this is just the beginning...

Why parameters?

Example: The controller node

Suppose you implemented some **discrete-time control law** in a ROS 2 node:

- subscribers constantly sample sensor measurements, and callbacks embed the control algorithm;
- the control law depends on some **parameters**;
- you would like to change the parameters without having to **recompile** your software each time;
- you would like to have **other modules** to change such parameters automatically if need be, and to **automatically react** to such changes.

... Middleware support is evidently needed.

Node parameters

A ROS 2 node can have one or more **parameters**: values that can be specified at **startup**, changed at **runtime**, and used in the implementation.

The parameter system is **decentralized** and **built on messages and services**: each node has its own parameters and related services, and updates are **broadcasted** to every other node over the `/parameter_events` topic (try to inspect it! Remember to log it too during experiments!).

Parameters can be **listed**, **queried**, **described** and **set**, using either **CLI tools** or **service calls**; **YAML** configuration files may be **loaded** or **dumped**.

It is possible to specify what to do when a parameter update is requested by defining a **callback**.

A parameter may be **read-only** and its type may be **dynamic**.

Parameter types

From the `rcl_interfaces/ParameterType` message file:

- `BOOL`
- `INTEGER`
- `DOUBLE`
- `STRING`
- `BYTE_ARRAY`
- `BOOL_ARRAY`
- `INTEGER_ARRAY`
- `DOUBLE_ARRAY`
- `STRING_ARRAY`

Parameters CLI commands

- `ros2 param list NODE_NAME`
  Lists available parameters of a node.
- `ros2 param describe NODE_NAME PARAMETER_NAME`
  Shows information about a parameter.
- `ros2 param get NODE_NAME PARAMETER_NAME`
  Returns the value of a parameter.
- `ros2 param set NODE_NAME PARAMETER_NAME VALUE`
  Sets a given value for a parameter.
- `ros2 param dump NODE_NAME`
  Dumps the current parameter configuration in a YAML file.
- `ros2 param load NODE_NAME PARAMETER_FILE`
  Loads parameters from a YAML file.

Parameters CLI commands

When starting a node with `ros2 run`, it is possible to specify parameters as **command-line arguments**:

```
ros2 run PACKAGE_NAME EXECUTABLE_NAME --ros-args -p param_name:=param_value
```

Multiple parameter values can be specified by **repeating** the `-p` option.
It is also possible to specify a **configuration file**:

```
ros2 run PACKAGE_NAME EXECUTABLE_NAME --ros-args --params-file CONFIG_FILE
```

Parameters services

When a node is launched, it **automatically** instantiates the following **services**:

- `~/get_parameters`
- `~/set_parameters`
- `~/list_parameters`
- `~/describe_parameters`
- `~/get_parameter_types`
- `~/set_parameters_atomically`

Try to inspect them with `ros2 service type`, or call them from the command line! They all take **groups of parameters** on which to operate, and can be called either from a **CLI tool** or from a **client node**: that is the way to go if you want to **automate parameter updates**.

Coding with parameters

Tips and best practices

- Parameters are referred to by their **name**.
- Before being used, a parameter must be **declared** to the middleware: this is usually done in the constructor of a node specifying their **name** and **default value** using the `declare_parameter` API.
- Parameter values can be retrieved **atomically** by calling the `get_parameter` API, which returns an `rclcpp::Parameter` object that must be **casted** to the appropriate type using the `as_*` methods (`as_int()`, `as_bool()`, ecc.).
- Accessing the middleware's internals to retrieve parameters might be **slow**: define **class member variables** that track the value of each parameter by being updated each time the parameter is.

Coding with parameters

Suggested TODO list

1. Define **class member variables** to track the value of each parameter.
2. Define a **callback** to be called when a parameter is updated (there's an API for this).
3. **Declare** the parameters in the **constructor** of the node; do this **first**, so that the parameters are available as soon as the node is started.
4. **Register** the callback to the middleware.
5. **Use** the parameter values in the implementation.

There are **two APIs**, corresponding to **two ways** to declare parameters:

1. the **lazy way**: `declare_parameter(...)` specifying **only the name and the default value** of the parameter;
2. the **complete way**: `declare_parameter(...)` specifying **all the information** about the parameter, including its **type**, **description**, **read-only** flag, **max** and **min** values, **constraints** and more, using a `rcl_interfaces::msg::ParameterDescriptor` object.

The complete way is **recommended**, but induces a lot of **boilerplate code**!
Check out our `params_manager` library!

Example: Parametric publisher

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

Roadmap

Scripting ROS 2 architectures

A ROS 2-based control architecture for a robot can easily get to have 20 nodes or more.

It then becomes critical to be able to **automate startup and configuration** of all the modules, or some subsets, also for testing.

That is what the ROS 2 **Launch System** is for.

Launch files

Launch files are **Python scripts** that specify how ROS 2 modules must be **located**, **configured** and **started**. Their format is such that the Launch System can parse and integrate them when invoked.

Many things can be configured about ROS 2 modules in such files:

- console and text files **logs**;
- command line **arguments**;
- node **parameters**;
- **remappings** of namespaces, node names, topics, services and actions.

It is also possible to start **custom executables**, define **environment variables**, and more.

Launch files may be **included**, so that **large architectures** can be started with **one command**.

Launch files

```
from launch import LaunchDescription
from launch_ros.actions import Node

# The following function MUST be specified

def generate_launch_description():
    """Builds a launch description."""
    ld = LaunchDescription()
    node = Node(
      package='PACKAGE_NAME',
      executable='EXECUTABLE_NAME')
    ld.add_action(node)
    return ld
```

Listing 1: Minimal example of a launch file that starts a ROS 2 node.

Launch System CLI commands

The main command to **execute the `LaunchDescription`** specified in a launch file is

```
ros2 launch PACKAGE_NAME LAUNCH_FILE
```

This will start a **Python interpreter** in your current shell, that will **parse** the launch file and **execute** the `LaunchDescription`. You will **interface with the underlying processes** through this interpreter instance. You can also specify **launch arguments** that you will be able to access from within the launch file:

```
ros2 launch PACKAGE_NAME LAUNCH_FILE ARG1:=VALUE1 ARG2:=VALUE2 ...
```

Have a look at the options of this command!

Coding launch files

Tips and best practices

- Their **extension** is usually `.launch.py`.
- They are usually placed in a package subdirectory named `launch/` that is **installed** in the workspace path during build, via appropriate directives in either `CMakeLists.txt` or `setup.py` files.
- A module can have its own launch files but those for the entire **architecture** must form an appropriate **package**, whose name is usually `PROJECT_bringup`.

Find a comprehensive description of all the features of launch files in [`launch_files.md`].

Example: Bringup package

Now go have a look at the `ros2-examples/src/ros2_examples_bringup` package!

Roadmap

Composable nodes

Isolation vs integration

A ROS 2-based control architecture for a robot can easily get to have 20 nodes or more.

If each node runs **in its own process**, its **errors or crashes** may not affect the behavior of the other nodes, but a lot of system resources are wasted for this (memory, context switches...).

On the other hand, if all nodes run **in the same process**, they can share resources and memory and ease the burden on the OS, but a **critical error** in one node may **affect the others**.

You may even **load the entire control architecture with a single launch file**!

Composable nodes

The building blocks of a control architecture

When the software of a ROS 2 node is mature enough, it can be compiled into a **component** (or **composable node**, formerly *nodelet*) (C++-only feature!).

Components are **shared libraries**, dynamically loaded into a single **container process** that hosts an **executor** and a **context** (`pluginlib` library, built around the `dlopen` system call).

Inside the container, the single executor schedules **all jobs**, and nodes use **shared memory** to communicate, bypassing the DDS and the RMW.

Nodes can be **loaded** (constructed) and **unloaded** (destructed) at runtime, without restarting the container.

The container process contains a `rclcpp_components::ComponentContainer` **node**, that exposes the `load_node`, `list_nodes`, and `unload_node` **services**.

Composable nodes

CLI commands

- `ros2 run rclcpp_components component_container`
  Starts a simple component container.
- `ros2 component list [CONTAINER_NAME]`
  Lists active components [within the given container].
- `ros2 component load CONTAINER_NAME PACKAGE_NAME PLUGIN_NAME`
  Loads a component into a container.
- `ros2 component unload CONTAINER_NAME COMPONENT_ID`
  Unloads a component from a container.
- `ros2 component types`
  Output a list of components registered in the ament index.
- `ros2 component standalone PACKAGE_NAME PLUGIN_NAME`
  Creates a new container and loads the given component into it.

To compile and use a component, you must:

1. add the `rclcpp_components` **dependency**;
2. add **macros** to your node class definition code to register the component class;
3. build a **shared library target**, comprising only the node class, within your `CMakeLists.txt` file;
4. **register the component** with ament within your `CMakeLists.txt` file;
5. launch it using the `ComposableNodeContainer` **launch action** and the `ComposableNode` **launch description**.

Find more in `components.md`.

Example: Composable pub/sub

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

Exercises

- Pick an example package of your choice, and make a launch file for it.
- Re-run the components example, then start a container from the command line and load the components in there.
- Pick an example package of your choice, and make components from it.