ROS 2

Node configuration

Roberto Masocco
roberto.masocco@uniroma2.it

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

May 31, 2023

TOR VERGATA
UNIVERSITY OF ROME
School of Engineering

Recap

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

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

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

New code examples are available.

This lecture is here.

Roadmap

1. Namespaces
2. Node parameters
3. Launch files

Roadmap

1. Namespaces
2. Node parameters
3. Launch files

Roadmap

1. Namespaces
2. Node parameters
3. Launch files

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 --ros2-args -p
param_name:=param_value ...
```

It is also possible to specify a **configuration file**:

```
ros2 run PACKAGE_NAME EXECUTABLE_NAME --ros2-args --params-file
PARAMETER_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`! 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**.

- 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**.
- Parameter values can be retrieved **atomically** by calling an API, but accessing the middleware's internals to do this 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.
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

1. Namespaces
2. Node parameters
3. Launch files

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

```python
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.

Coding launch files

Hints 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!