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 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.

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.

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 service, and updates are **broadcasted** to every other node.

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**[1].

[1] Only from Galactic.

Parameter Types

From the `rcl_interfaces/msg/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.

Coding with Parameters

Parameters 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.
- Parameter values can be retrieved **atomically** by calling an API, but accessing the middleware's internals to do this might be **slow**: it is best to define **local variables** that track the value of each parameter by being updated each time the parameter is.

Example: Parametric Publisher

Now go have a look at the ros2-examples/src/parameters_example 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 modules in such files:

- console and text files **logs**;
- command line **arguments**;
- node **parameters**;
- **remappings**.

Targeted modules may even not be ROS 2 nodes.

Launch files may be **included**, so that large architectures can be started with single commands.

Launch Files

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

# The following function MUST be specified

"""Builds a launch description."""
def generate_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

Coding Launch Files

Launch Files Best Practices

- Their extension is usually `.launch.py`.
- They are usually placed in a directory named `launch` that is installed in the workspace path during build.
- 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`.

A comprehensive description of all the features of launch files can be found in launch_files.md.

Example: Bringup Package

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