ENPM702

Introductory Robot Programming

L10: Robot Operating System (ROS) - Part II

v2.0

Lecturer: Z. Kootbally

Semester/Year: Summer/2025

MARYLAND APPLIED GRADUATE ENGINEERING

Table of Contents

Changelog

- **v1.0**: Original version.

Learning Objectives

By the end of this session, you will be able to:

- **Launch Files** - Covers creating launch files, passing arguments, conditional launching, and node groups.
- **Parameters** - Includes declaring, retrieving, and configuring parameters through various methods including callbacks.
- **Mobile Robot** - Focuses on Turtlebot control and proportional controller implementation.
- **Executors** - Covers single/multi-threaded executors and callback groups.
- **Custom Interfaces** - Describes specific packages for custom interfaces.

Launch Files

Launch files provide a streamlined mechanism for initiating multiple nodes simultaneously while enabling dynamic configuration.

- `cd` to the root of the workspace.
- `rosdep install -i --from-path src --rosdistro $ROS_DISTRO -y`
- `colcon build --symlink-install --packages-select launch_files_demo`
- Source the workspace.

Resources

- ROS 2 Documentation: Launch Files Tutorials
- ROS 2 Documentation: Creating Launch Files
- ROS 2 Documentation: Launch File Formats
- Robotics Backend: ROS 2 Launch File Examples
- GitHub: ROS 2 Launch Package
- ROS 2 Documentation: Using Substitutions in Launch Files

Purposes

1. **Node Management:** Launch, configure, and control individual ROS 2 nodes, including lifecycle and composable nodes, to define system behavior.
2. **Modularity and Reuse:** Structure complex systems by reusing launch files, grouping nodes, and managing namespace scopes for better organization and scalability.
3. **Configuration and Customization:** Allow users to tailor launch behavior using arguments, environment variables, and substitutions that adapt to various runtime contexts.
4. **Execution Control**: Control the timing and conditions under which nodes and actions are launched, including timers, conditionals, and event-driven execution.
5. **Custom Launch Logic:** Use Python-based logic and functions to perform dynamic setup, computations, or system introspection before launching actions.
6. **Logging and Diagnostics:** Monitor system behavior and assist debugging by printing messages, adjusting log levels, and tracking node status during launch.
7. **Interfacing with Non-ROS Systems:** Integrate external commands or scripts into your launch process to coordinate ROS 2 with other tools or system-level operations.

Anatomy

A typical Python launch file contains:

- **Import statements**: Required launch and ROS dependencies.
- **Launch description**: A function that returns the launch description containing all node configurations.
- **Node configuration**: Information about nodes, parameters, remappings, and more.

Location

Launch files are typically found in the **launch** directory within your package.

Edit **CMakeLists.txt** to install launch files.

Configurations

There are two main configurations for launch files (choose one of them).

demo1.launch.py

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

# This function must be defined
def generate_launch_description():
    ld = LaunchDescription()
    talker = Node(
        package="demo_nodes_cpp",
        executable="talker")
    listener = Node(
        package="demo_nodes_cpp",
        executable="listener")

    ld.add_action(talker)
    ld.add_action(listener)

    return ld
```

demo2.launch.py

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

# This function must be defined
def generate_launch_description():
    return LaunchDescription(
        [
            Node(
                package="demo_nodes_cpp",
                executable="talker",
            ),
            Node(
                package="demo_nodes_cpp",
                executable="listener",
            ),
        ]
    )
```

Demonstration

```
ros2 launch launch_files_demo demo1.launch.py
```

or

```
ros2 launch launch_files_demo demo2.launch.py
```

Advanced Features

Some advanced features provided by launch files include:

- Include launch files from other packages.
- Pass arguments to launch files.
- Conditional execution.
- Group nodes.
- Remap names.
- Pass parameter files.

Include Other Launch Files

Invoke another launch file from a different package.

```
ros2 launch launch_files_demo demo3.launch.py
```

Conditional Launching

Start a node conditionally from a launch argument.

```
ros2 launch launch_files_demo demo4.launch.py talker_arg:=true
```

You can check arguments that can be passed to a launch file with:

```
ros2 launch <package> <launch file> --show-args
```

Node Grouping

Start a group of nodes.

```
ros2 launch launch_files_demo demo5.launch.py
```

Node Grouping Conditional Launch

Start a group of nodes conditionally using a launch argument.

```
ros2 launch sensor_demo_pkg demo6.launch.py enable_nav_sensors:=true
```

Parameters

A parameter is a configurable value that can be used to customize the behavior of a node at runtime **without modifying the code**. Parameters allow nodes to store and retrieve data, such as tuning constants, file paths, or robot-specific settings.

- `cd` within your workspace directory.
- `colcon build --symlink-install --packages-select parameters_demo`
- Source the workspace.

Resources

- ROS 2 Parameters Overview
- ROS 2 Parameter Tutorials
- ROS 2 rclpy Parameters API
- ROS 2 CLI Parameter Commands
- Using Parameters with Launch Files
- Loading Parameters from YAML Files
- Parameter Groups

Parameters

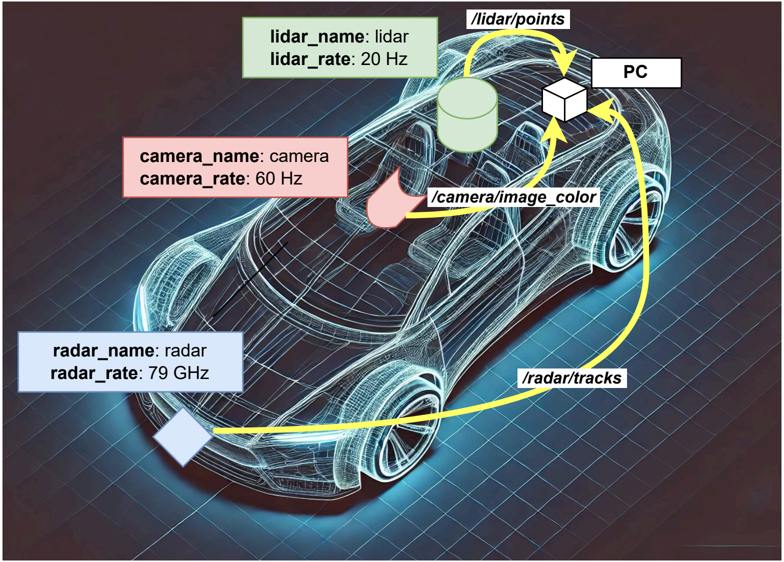
Parameters can be set or updated during runtime using the CLI or within the node itself.

Nodes can declare and retrieve parameters, making them useful for settings that do not change frequently.

Parameters can have the following types: `bool`, `bool[]`, `int64`, `int64[]`, `double`, `double[]`, `string`, `string[]`, and `byte[]`

Each parameter belongs to a specific node and cannot be accessed globally by other nodes.

Parameters



lidar_name: lidar
lidar_rate: 20 Hz
/lidar/points
PC
camera_name: camera
camera_rate: 60 Hz
/camera/image_color
radar_name: radar
radar_rate: 79 GHz
/radar/tracks

CLI for Parameters

- `ros2 param -h`

```
Commands:
  delete     Delete parameter
  describe   Show descriptive information about declared parameters
  dump       Dump the parameters of a node to a yaml file
  get        Get parameter
  list       Output a list of available parameters
  load       Load parameter file for a node
  set        Set parameter
```

- Start a node: `ros2 run demo_nodes_cpp talker`
- List all parameters: `ros2 param list /talker`
- Get information about a parameter:

`ros2 param get /talker use_sim_time`

Declaring Parameters

Parameters **must be explicitly declared** before they can be accessed within a node. If you try to access a parameter without declaring it first, ROS will throw an error.

```
this->get_parameter("camera_name");  // ERROR: Parameter 'camera_name' has not been
↪  declared
```

Declaring parameters in ROS 2 serves several important purposes:

1. **Enforces explicit configuration**: Explicit declaration reduces unexpected behavior and helps catch configuration mistakes early.
2. **Allows default values**: If a parameter is not provided externally, a default value ensures the node operates correctly.

```
this->declare_parameter("camera_name", "camera");
// Or with explicit type:
this->declare_parameter<std::string>("camera_name", "camera");
```

3. **Enables dynamic reconfiguration**: Declaring parameters makes them modifiable at runtime and allows dynamic tuning. Example of **modifying a parameter while the node is running**:

```
ros2 param set /camera_demo camera_name 'front_camera'
```

Declaring Parameters

There are multiple ways to declare parameters:

- Declare each parameter individually with `declare_parameter()`.

```cpp
this->declare_parameter<std::string>("camera_name", "camera");
this->declare_parameter<int>("camera_rate", 60);
```

- Declare each parameter individually with constraints and metadata with `declare_parameter()`. Constraints and metadata will be used with `ros2 param describe`

```cpp
// #include <rcl_interfaces/msg/parameter_descriptor.hpp>
// #include <rcl_interfaces/msg/integer_range.hpp>

rcl_interfaces::msg::ParameterDescriptor camera_name_desc;
camera_name_desc.description = "Name of the camera";
this->declare_parameter<std::string>("camera_name", "camera", camera_name_desc);

rcl_interfaces::msg::ParameterDescriptor camera_rate_desc;
camera_rate_desc.description = "Camera frame rate in Hz";
rcl_interfaces::msg::IntegerRange range;
range.from_value = 10;
range.to_value = 60;
range.step = 10;
camera_rate_desc.integer_range.push_back(range);
this->declare_parameter<int>("camera_rate", 60, camera_rate_desc);
```

Demonstration

- `ros2 run parameters_demo camera_demo`
- `ros2 param list /camera_demo`

```
camera_name
camera_rate
...
```

- `ros2 param get /camera_demo camera_name`

```
String value is: camera
```

- `ros2 param describe /camera_demo camera_rate`

```
Parameter name: camera_rate
  Type: integer
  Description: Camera frame rate in Hz
  Constraints:
    Min value: 10
    Max value: 80
    Step: 10
```

Retrieving Parameters

After declaring a parameter, you might want to retrieve its value for several reasons:

- **Initialization**: Parameters are often declared with default values, but you may need to retrieve the actual value to initialize parts of your node or its functionalities based on this configuration.
- **Dynamic reconfiguration**: Parameters can be changed at runtime. Retrieving the parameter allows your node to react to changes in configuration and adjust its behavior dynamically.
- **Operational tuning**: Parameters are frequently used for tuning algorithms or adjusting settings for sensors and actuators. Getting the parameter value allows your node to adapt its operation according to these settings.

Example

```cpp
// Get param camera_name and store it for later use
camera_name_ = this->get_parameter("camera_name").as_string();

// Get param camera_rate and store it for later use
camera_rate_ = this->get_parameter("camera_rate").as_int();

// Alternative approach using get_parameter_value()
camera_name_ = this->get_parameter("camera_name").get_parameter_value().get<std::string>();
camera_rate_ = this->get_parameter("camera_rate").get_parameter_value().get<int>();
```

Using Parameters

Implement parameters as functional class attributes.

Example

- The p **camera_name** parameter provides meaningful context in logs:

```
RCLCPP_INFO(this->get_logger(),
  "%sPublished frame from:%s %s%s%s",
  parameters_demo_nodes::Color::PURPLE, parameters_demo_nodes::Color::RESET,
  parameters_demo_nodes::Color::RED, camera_name_.c_str(), parameters_demo_nodes::Color::RESET);
```

- The p **camera_rate** parameter directly controls publishing frequency:

```
auto timer_period = std::chrono::duration<double>(1.0 / camera_rate_);
```

Setting Parameters

Setting parameters involves **modifying the value of a declared parameter either prior to or during the execution of the node.** This allows dynamic node configuration without requiring code modifications or recompilation.

There are several ways to set parameters:

1. Configure individual parameters using the CLI (slide 24).
2. Define individual parameters within a launch file (slide 25).
3. Use a parameter file (YAML file):
   - Use CLI to configure parameters using a parameter file (slide 27).
   - Use a launch file to configure parameters using a parameter file (slide 28).
4. Set parameters programmatically (slide 31).
5. Set parameters with `ros2 param set` (slide 30).
6. Set parameters using launch file arguments (slide 34).

Configure Individual Parameters (CLI)

- Use `--ros-args` to pass arguments to a node on the command line.
- Use `-p <parameter>:=<value>` to set a value for a parameter.

Demonstration

```
ros2 run parameters_demo camera_demo
```

```
[INFO] [1755214205.549434905] [camera_demo]: Published frame from: camera
```

```
ros2 run parameters_demo camera_demo --ros-args -p camera_name:='front_camera'
```

```
[INFO] [1755214205.549434905] [camera_demo]: Published frame from: front_camera
```

Exercise #1

Assign different values to the p **camera_rate** parameter: 5, 15, 70, and 90.

Configure Individual Parameters (Launch File)

Set parameter values in a launch file.

Example

```
camera_node = Node(
    package='parameters_demo',
    executable='camera_demo',
    parameters=[
        {'camera_name': 'front_camera'},
        {'camera_rate': 30}
    ],
    output='screen',
    emulate_tty=True
)
```

Demonstration

```
ros2 launch parameters_demo demo1.launch.py
```

Use a Parameter File (YAML)

A **parameter file** in ROS 2 is a YAML configuration file that stores parameters for one or more nodes.

```yaml
camera_demo:  # Name of the node
  ros__parameters:
    camera_name: 'front_camera'
    camera_rate: 30

lidar_demo:  # Name of the node
  ros__parameters:
    lidar_name: 'top_lidar'
    lidar_rate: 20

radar_demo:  # Name of the node
  ros__parameters:
    radar_name: 'front_radar'
    radar_rate: 60

processing_demo:  # Name of the node
  ros__parameters:
    processing_mode: 'all'
    processing_rate: 10
```

- YAML files are usually placed in the **config** directory (best practice).
- Ensure you edit **CMakeLists.txt** to install the **config** folder.

Store Parameters in a Parameter File (YAML)

- Use the parameter file on the command line by providing the path to the file (relative or absolute).
  - Use `--ros-args` to pass arguments to a node on the command line.
  - Use `--params-file <path>` to pass the parameter file to the node (absolute or relative path).

Demonstration

```
ros2 run parameters_demo camera_demo --ros-args --params-file <file path>
```

Store Parameters in a Parameter File (YAML)

- Pass a parameter file to a launch file.
  1. Retrieve the path to the parameter file:

```
parameters_demo_file = PathJoinSubstitution(
    [FindPackageShare("parameters_demo"), "config", "parameters_demo.yaml"]
)
```

  2. Pass the parameter file to the node:

```
camera_node = Node(
    package="parameters_demo",
    executable="camera_demo",
    parameters=[parameters_demo_file],
    output="screen",
    emulate_tty=True,
)
```

Demonstration

```
ros2 launch parameters_demo demo2.launch.py
```

Programmatic Parameter Configuration

Parameters can be defined and modified directly within your program. This enables dynamic adjustments in which different values are assigned to the same parameter based on specific logic.

Demonstration

```
// Include required header:
// #include <rclcpp/parameter.hpp>

this->set_parameters({rclcpp::Parameter("camera_rate", 40)});

// Alternative with explicit type specification:
this->set_parameters({
    rclcpp::Parameter("camera_rate", rclcpp::ParameterValue(40))
});
```

CLI Parameter Configuration

Parameters can be updated while your node is running with `ros2 param set`

Demonstration

1. Start all nodes.
   `ros2 launch parameters_demo demo2.launch.py`
2. Display all parameters for the node n **camera_demo**
   `ros2 param list /camera_demo`
3. Get the value for the parameter p **camera_name**
   `ros2 param get /camera_demo camera_name`
   ```
   String value is: front_camera
   ```
4. Change the value of p **camera_name**
   `ros2 param set /camera_demo camera_name 'front_facing_camera'`
   ```
   Set parameter successful
   ```
   `ros2 param get /camera_demo camera_name`
   ```
   String value is: front_camera
   ```

The value of p camera_name has been updated, but the C++ node's attribute still reflects the previous value. After a parameter is read during initialization, the node does not observe subsequent updates unless it is explicitly notified.

Add an on-set-parameters callback so the node is notified immediately when the parameter is modified.

The method `rclcpp::Node::add_on_set_parameters_callback` registers a callable that is invoked whenever a parameter change is requested through the ROS 2 parameter API (for example, using `ros2 param set`). The callback receives the proposed `std::vector<rclcpp::Parameter>` and must return `rclcpp::SetParametersResult` to accept or reject the update.

Parameter Change Callback

- Register a callback function for parameter changes:

```
// In constructor or initialization
param_callback_handle_ =
↪ this->add_on_set_parameters_callback(std::bind(&CameraDemoNode::parameter_update_callback,
↪ this, std::placeholders::_1));
```

- Define the callback: see `parameter_update_callback()`

Demonstration

1. `ros2 launch parameters_demo demo2.launch.py`
2. `ros2 param set camera_demo camera_name 'front_facing_camera'`

Parameter Change Callback

A timer is initialized when the node is created and executes its associated callback once the node begins running. To modify the timer's frequency, it must be canceled and then recreated using the updated frequency.

1. Cancel the timer:

```
data_camera_timer_.reset();
```

2. Recreate the timer with the new rate:

```
data_camera_timer_ = this->create_wall_timer(
    std::chrono::duration_cast<std::chrono::nanoseconds>(timer_period),
    std::bind(&CameraDemoNode::data_camera_pub_callback, this));
```

Demonstration

1. `ros2 run parameters_demo camera_demo`
2. `ros2 topic hz /camera/image_color`
3. `ros2 param set camera_demo camera_rate 10` → New frequency applied.

Parameters as Launch File Arguments

Launch file arguments can be used as parameters.

Example: lidar_demo.cpp

- Declare and store the parameter: p **lidar_model**

Example: demo3.launch.py

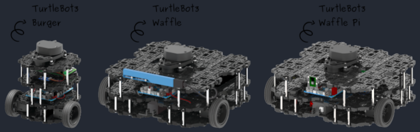
- Declare a launch configuration variable.
- Declare a launch argument with default value.
- Pass the launch configuration variable to the node:

```
lidar_node = Node(
    package="parameters_demo",
    executable="lidar_demo",
    parameters=[parameters_demo_file,{"lidar_model": lidar_model}],
    output="screen",
    emulate_tty=True,
)
```

- Add the launch argument: `ld.add_action(lidar_model_arg)`

Mobile Robot

The **Turtlebot** is a differential wheeled robot, i.e., its movement is based on two separately driven wheels placed on either side of the robot body.



Key Differences

- **Processing Power:** Waffle > Burger = Waffle Pi
- **Camera Quality:** Waffle (depth camera) > Waffle Pi (RGB camera) > Burger (none)
- **Platform Size:** Waffle = Waffle Pi > Burger
- **Price:** Burger < Waffle Pi < Waffle
- **Battery Life:** Waffle = Waffle Pi > Burger

Build the package:

```
colcon build --packages-select bot_controller_demo --symlink-install
```

Topics

- **Control:** To control the robot, publish to t **/cmd_vel**.

> - **Gazebo Integration:** Requires m `geometry_msgs/msg/TwistStamped` messages on t `/cmd_vel` topic.
> - **RViz Visualization:** Accepts m `geometry_msgs/msg/Twist` messages on t `/cmd_vel` topic.

  - `ros2 interface show geometry_msgs/msg/Twist` or `ros2 interface show geometry_msgs/msg/TwistStamped`
    - Positive `linear.x` values move the robot forward, while negative values move it backward.
    - Positive `angular.z` values rotate the robot counterclockwise, while negative values rotate it clockwise.
- **Pose:** To obtain the robot's current pose (position and orientation), subscribe to the t **/odometry** topic.
  - `ros2 interface show nav_msgs/msg/Odometry`

Visualization

- **RViz2**: A 3D visualization tool for displaying robot data, sensor information, and debugging ROS topics. **It does not simulate physics or robot behavior**.
- Used for monitoring robot state, planning paths, visualizing sensor data (laser scans, point clouds), and debugging system behavior in real-time.
- `ros2 launch turtlebot3_fake_node turtlebot3_fake_node.launch.py`

Simulation

- **Gazebo**: A physics-based robot simulator that provides realistic sensor simulation, dynamics, and environmental interactions.
- When using Gazebo, set p **use_sim_time** to true to synchronize all ROS nodes with the simulation clock rather than system time, ensuring proper timing coordination.
- Gazebo publishes simulated sensor data, handles robot physics, and can simulate complex scenarios for testing before deploying on real hardware.
- `ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py`
- `ros2 launch turtlebot3_fake_node rviz2.launch.py` (optional)

Manual Control (Teleoperation)

- Launch the robot simulation and/or visualization environment.
- Run the keyboard teleop node to control the robot manually:
- `ros2 run teleop_twist_keyboard teleop_twist_keyboard`
- Use keyboard keys to move the robot around

Autonomous Control (Programmatic)

- `ros2 launch turtlebot3_gazebo empty_world.launch.py`
- `ros2 launch bot_controller_demo random_controller.launch.py controller_type:=gazebo`

or

- `ros2 launch turtlebot3_fake_node turtlebot3_fake_node.launch.py`
- `ros2 launch bot_controller_demo random_controller.launch.py controller_type:=rviz`

Proportional Controller

A **proportional controller** is a feedback control mechanism where the control output is directly proportional to the current error. The error is the difference between the desired value (`setpoint`) and the measured value (`process variable`). Mathematically, the control signal $u(t)$ is given by:

$$u(t) = K_p \cdot e(t)$$

where:

- $u(t)$ is the control output at time t.
- K_p is the proportional gain, a constant that determines the strength of the controller's response to the error.
- $e(t)$ is the error at time t (desired value - measured value).

Mobile Robot Application

For a mobile robot aiming for a goal (x_g, y_g) and currently at (x_r, y_r), we can define errors in the x and y directions:

- $e_x = x_g - x_r$
- $e_y = y_g - y_r$

The robot's linear and/or angular velocities can be controlled proportionally to these errors. For instance:

- **Linear Velocity (v):** $v = K_{pv} \cdot \sqrt{e_x^2 + e_y^2}$
  where K_{pv} is the proportional gain for linear velocity.
- **Angular Velocity (ω):** Let θ_g be the angle to the goal from the robot's current orientation θ_r. The error in angle is $e_\theta = \theta_g - \theta_r$. Then, the angular velocity can be controlled as:

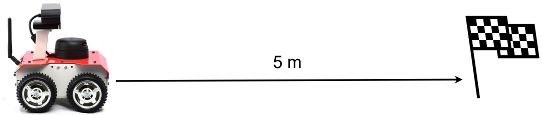
$$\omega = K_{p\omega} \cdot e_\theta$$

where $K_{p\omega}$ is the proportional gain for angular velocity.

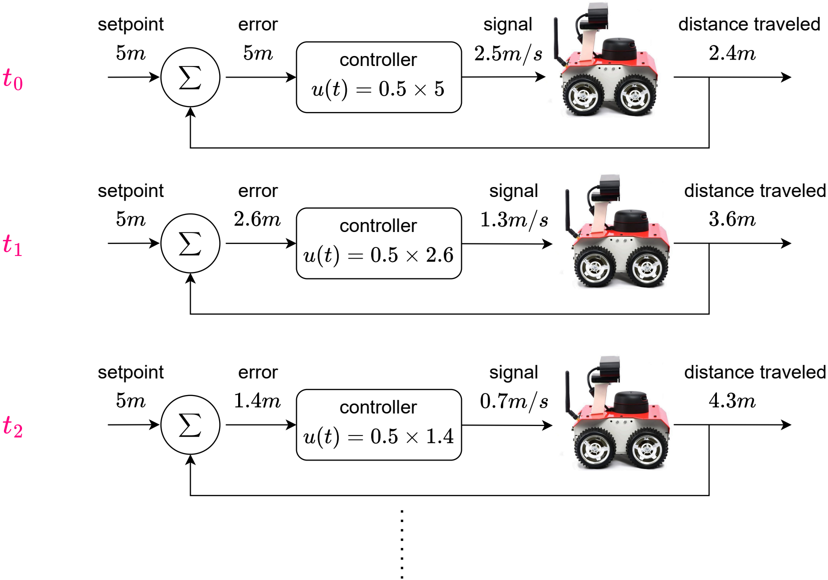
The controller continuously calculates the errors and adjusts the robot's velocities. As the robot gets closer to the goal, the errors (e_x, e_y, and potentially e_θ) decrease, causing the velocities to decrease as well. Ideally, when the robot reaches the goal, the errors become zero, and the robot stops (or its velocity approaches zero).

Example

Drive a distance of 5 m in a straight line.



We will assume a value for the proportional gain, e.g., $K_{pv} = 0.5s^{-1}$. The units ensure that when multiplied by distance in meters, we get velocity in m/s.


t_0
setpoint
$5m$
Σ
error
$5m$
controller
$u(t) = 0.5 \times 5$
signal
$2.5m/s$
distance traveled
$2.4m$
t_1
setpoint
$5m$
Σ
error
$2.6m$
controller
$u(t) = 0.5 \times 2.6$
signal
$1.3m/s$
distance traveled
$3.6m$
t_2
setpoint
$5m$
Σ
error
$1.4m$
controller
$u(t) = 0.5 \times 1.4$
signal
$0.7m/s$
distance traveled
$4.3m$

Demonstration

- `ros2 launch turtlebot3_gazebo empty_world.launch.py`
- `ros2 launch bot_controller_demo proportional_controller.launch.py controller_type:=gazebo`

or

- `ros2 launch turtlebot3_fake_node turtlebot3_fake_node.launch.py`
- `ros2 launch bot_controller_demo proportional_controller.launch.py controller_type:=rviz`

Scaling Beyond Single Tasks

The proportional controller demo illustrated a **single-purpose node** - one callback managing robot movement. But real robotic systems require:

Multiple Simultaneous Tasks

- Process sensor data (cameras, lidar, IMU).
- Update control commands at different rates.
- Monitor system health and safety.
- Handle user commands.
- Log data for analysis.

Coordination Challenges

- How to handle multiple callbacks?
- What if one callback blocks others?
- Can we process sensors in parallel?
- How to prioritize critical tasks?
- When to use threads vs. sequential processing?

Solution: ROS 2 Executors

Executors

Executors manage how and when your callbacks run, enabling complex multi-task robotic systems.

- Executors simplify the task of handling threads by providing an abstraction layer, allowing operation with either a single thread (e.g., single-threaded executor) or multiple threads (e.g., multi-threaded executor).
- Executors can manage the callbacks of one or more nodes at the same time.

Thread

In computer science, a **thread of execution** (or thread) is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.

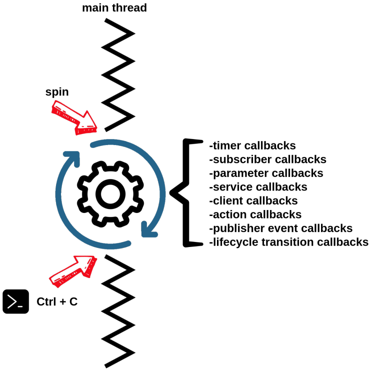
Build the package:

```
colcon build --packages-select executors_demo --symlink-install
```

Single-Threaded Executors

A **single-threaded executor** ensures that all these callbacks are executed sequentially in a single thread.

- One callback at a time **in the order they are scheduled** and **without concurrency**.
- Suitable for applications with low computational demands or when deterministic execution is required.



Example

Imagine a robot subscribing to sensor data and publishing commands. With a single-threaded executor:

1. It receives a sensor message and runs the callback to process it.
2. Only after the first step is done does it move to the next task, like running a timer callback to publish a command.

- This approach keeps things simple and predictable, avoiding issues like race conditions that can pop up when multiple threads access shared resources.
- However, it is not great for performance if you have a lot of tasks that could run independently, for that, you might look at a multi-threaded executor instead.

rclcpp::spin() vs SingleThreadedExecutor

They are essentially the same! `rclcpp::spin(node)` is a convenience wrapper.

Using `rclcpp::spin()`

```
int main(int argc, char** argv) {
    rclcpp::init(argc, argv);

    auto node = std::make_shared<MyNode>();

    // Simple spinning
    rclcpp::spin(node);

    rclcpp::shutdown();
}
```

What happens internally:

- Creates a `SingleThreadedExecutor`
- Adds your node to it
- Calls `executor.spin()`
- Blocks until shutdown

Using Executor Explicitly

```
int main(int argc, char** argv) {
    rclcpp::init(argc, argv);

    auto node = std::make_shared<MyNode>();

    // Explicit executor
    rclcpp::executors::SingleThreadedExecutor
    ↪  executor;
    executor.add_node(node);
    executor.spin();

    rclcpp::shutdown();
}
```

More control over:

- Multiple nodes in one executor
- Switching executor types easily
- Custom spin behaviors

When to use explicit executors?

Multiple Nodes

```
executor.add_node(camera_node);
executor.add_node(lidar_node);
executor.spin();
```

Easy Type Switching

```
// Just change the type!
rclcpp::executors::MultiThreadedExecutor executor;
```

Multi-Threaded Executors

A **multi-threaded executor**, is a mechanism for managing and executing callbacks across multiple threads, allowing for concurrent processing of tasks.

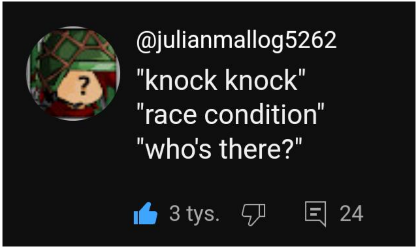
- **Thread pool**: A multi-threaded executor creates a pool of threads (you can often specify how many). Each thread can independently process callbacks from nodes added to the executor.
- **Callback scheduling**: When events occur, like a message arriving on a topic, a timer firing, or a service request, the executor assigns pending callbacks to available threads. If multiple callbacks are ready at once, they can run concurrently across different threads.
- **Spinning**: Calling `executor.spin()` starts an event loop that continuously checks for and dispatches work to the thread pool.

Benefits

- **Performance:** Ideal for applications with many independent tasks (e.g., processing data from multiple sensors). Concurrent execution can reduce latency and improve throughput.
- **Scalability:** Handles multiple nodes or high-frequency callbacks better than a single-threaded executor, which can bottleneck under heavy load.
- **Responsiveness:** Critical tasks (like responding to an emergency stop signal) might not get delayed by slower, less urgent ones.

Challenges

- **Race Conditions:** If callbacks access shared resources (e.g., a class attribute), you will need synchronization mechanisms like locks to prevent data corruption. Single-threaded executors avoid this issue entirely.



- **Overhead:** Managing multiple threads introduces some complexity and CPU overhead. If your application is lightweight, the extra threads might not be worth it.

Callback Groups

A **callback group** is a container within a node that holds callbacks (e.g., for subscriptions, timers, or services). Each group defines how its callbacks are handled in terms of execution and threading.

- By default, all callbacks belong to the node's implicit callback group. You can create explicit callback groups to customize execution behavior.
- **Two types exist**: `MutuallyExclusive` (only one callback executes at a time) and `Reentrant` (multiple callbacks can execute in parallel).
- Useful for managing concurrency, preventing race conditions, prioritizing certain callbacks, and isolating time-critical operations from blocking ones.
- The executor type (**single-threaded** vs. **multi-threaded**) determines whether callback groups can actually leverage concurrency.

Mutually Exclusive (Mutex) Callback Group

Callbacks within a **mutually exclusive callback group** cannot run at the same time, even in a multi-threaded executor. They are executed sequentially, one after another.

Use case: When callbacks share resources (e.g., modifying the same variable) and you want to avoid race conditions without explicit locks.

Reentrant Callback Group

Callbacks within a **reentrant callback group** can run concurrently with each other (and with callbacks in other reentrant groups), assuming the executor supports multiple threads.

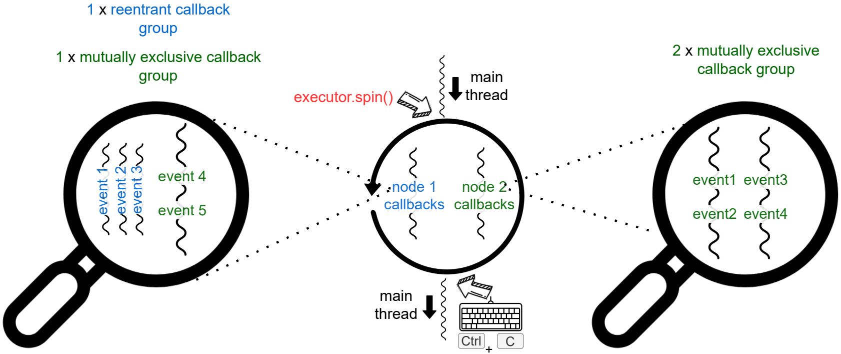
Use case: Independent tasks that don't interfere with each other, maximizing concurrency.

> If you use a `MultiThreadedExecutor` without explicitly defining callback groups, the default behavior is equivalent to all callbacks being in a **single, reentrant callback group**. ROS 2 assumes that callbacks are independent unless told otherwise.

Example

Consider two nodes added to a multi-threaded executor:

- `ros2 run executors_demo mutex_reentrant`
  - One reentrant callback group containing 3 callbacks.
  - One mutually exclusive callback group containing 2 callbacks.
- `ros2 run executors_demo two_mutex`
  - Two mutually exclusive callback groups, each containing 2 callbacks.



Reentrant callback groups allow multiple instances of the same callback to run simultaneously on different threads.

Demonstration

- Include a delay of 5 seconds in `timer1_callback()` (`mutex_reentrant.cpp`)
  - **Second 1:** Timer1 starts (Thread ID: 12475764978651514590) and begins sleeping for 5 seconds.
  - **Second 2:** Timer1 starts AGAIN (Thread ID: 9514691491644469051) on a different thread while the first instance is still sleeping.
  - **Second 3:** Timer1 starts AGAIN (Thread ID: 8441981844220120691) on yet another thread
  - And so on...

Reentrant = Multiple Concurrent Executions

- The executor doesn't wait for Timer1 to finish before starting it again.
- Each timer firing gets its own thread and can run in parallel.
- You can have multiple instances of the same callback running simultaneously.

Interfaces

An interface defines the data structure used for communication between nodes. Interfaces specify what kind of data is exchanged but not how it is transmitted.

CLI

- `ros2 interface -h`
- `ros2 interface list`

Resources

- ROS2 Interfaces

Interface Types

ROS 2 provides three types of interfaces:

- **Message** (.msg) – Used for topics (one-way communication).
- **Service** (.srv) – Used for request/response communication (synchronous).
- **Action** (.action) – Used for goal-oriented tasks with feedback (asynchronous).

Key Benefits

- **Type Safety:** Compile-time checking prevents data type mismatches.
- **Language Agnostic:** Same interface definition works across C++, Python, etc.
- **Standardization:** Consistent communication contracts across nodes.

Scenario: Robot Status Publishing

A mobile robot publishes its operational status to monitor its health and performance. The robot continuously sends sensor readings and operational mode.

Topic Communication

The system uses simple topic-based communication:

- **Publisher:** n **robot_monitor** publishes status data.
- **Topic:** t **/robot_status** carries the status messages.
- **Subscriber:** n **fleet_monitor** receives and processes status.

Interface Package

Custom interfaces are created in a dedicated package with no implementation code. This package should have the suffix **_msgs** or **_interfaces**.

The package ***robot_custom_interfaces*** was created with:

```
ros2 pkg create robot_custom_interfaces --dependencies std_msgs
```

```
robot_custom_interfaces
├── msg
│   └── RobotStatus.msg
├── CMakeLists.txt
└── package.xml
```

For this topic message example, we only need the **msg** folder. The **src** and **include** folders are not needed.

Complex Message: RobotStatus.msg

This message demonstrates key ROS 2 message features including constants, header, string, and arrays.

```
# Robot operational mode constants
uint8 MODE_IDLE=0
uint8 MODE_MOVING=1
uint8 MODE_WORKING=2
uint8 MODE_CHARGING=3
uint8 MODE_ERROR=4

# Standard ROS header with timestamp and frame
std_msgs/Header header

# Robot identification and current mode
string robot_name
uint8 mode                    # Use mode constants above

# Sensor readings (temperature, humidity, etc.)
float32[] sensor_readings
```

Message Structure Analysis

- `Constants`: Predefined values (`MODE_*`) improve code readability and prevent magic numbers.
- `header`: Standard ROS header providing timestamp and coordinate frame reference.
- `string`: Variable-length text field for robot identification.
- `float32[]`: Dynamic array of sensor readings (temperatures, voltages, distances).

Usage Examples

- `sensor_readings[0]`: Battery voltage (V)
- `sensor_readings[1]`: Internal temperature (°C)
- `sensor_readings[2]`: Distance to nearest obstacle (m)

package.xml

```xml
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

CMakeLists.txt

```cmake
find_package(rosidl_default_generators REQUIRED)

# Generate custom interfaces
rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/RobotStatus.msg"
  DEPENDENCIES std_msgs
)

ament_export_dependencies(rosidl_default_runtime)
```

1. Build the package:
   ```
   colcon build --packages-select robot_custom_interfaces
   ```
2. Source the workspace.
3. Check the message interface:
   ```
   ros2 interface show robot_custom_interfaces/msg/RobotStatus
   ```
4. Check the **install** folder to see that **RobotStatus.msg** was converted to C++ and Python code.

Using the Interface

Any package that uses `robot_custom_interfaces/msg/RobotStatus` must include ***robot_custom_interfaces*** as a dependency.

The package ***robot_interfaces_demo*** was created with:

```
ros2 pkg create robot_interfaces_demo --dependencies std_msgs \
    robot_custom_interfaces rclcpp
```

1. Build the package:
   ```
   colcon build --packages-up-to robot_interfaces_demo
   ```
2. Source the workspace.
3. Start the robot monitor (publisher and subscriber):
   ```
   ros2 run robot_interfaces_demo robot_status_demo
   ```

Next Class

- Lecture 11: ROS (Part 3)
- Quiz #6