UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

ROS 2 Development

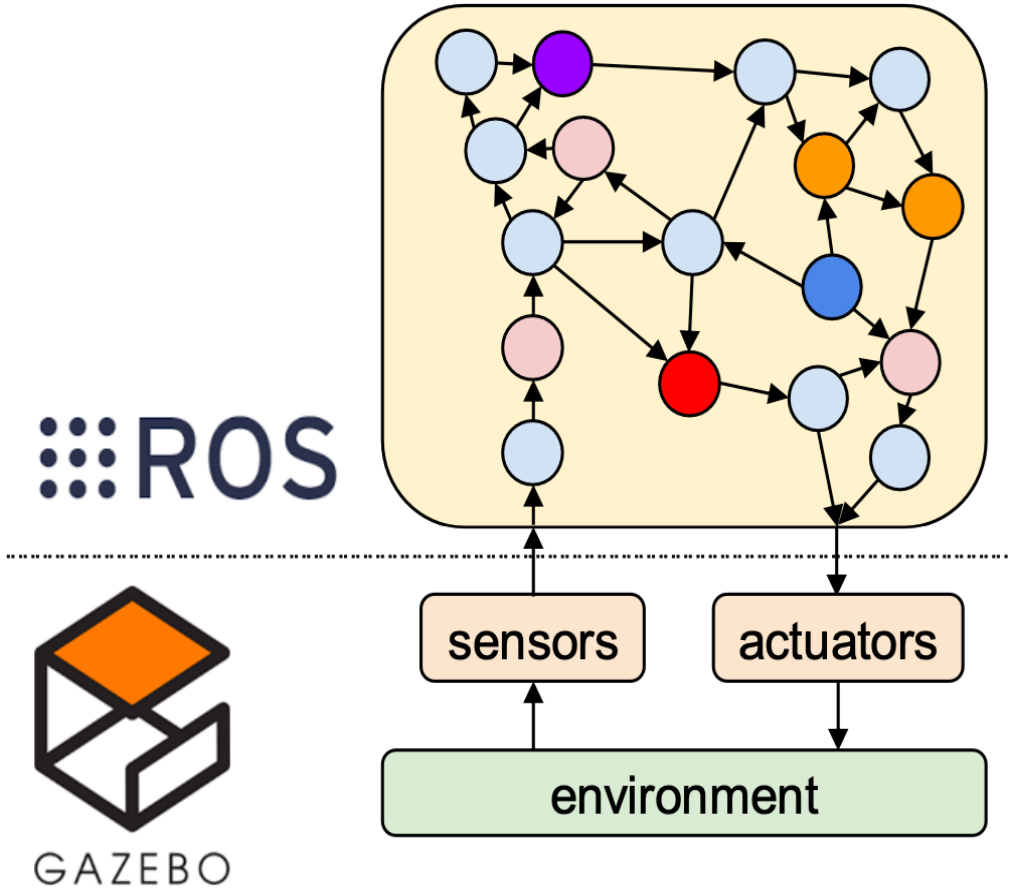
Edoardo Lamon, Luigi Palopoli, Enrico Saccon

Software Development for Collaborative Robots

Academic Year 2025/26

ROS 2 Structure

- A ROS2 project is divided in **workspaces** and **packages**
- A workspace is a collection of other workspaces or packages
- A package is a collection of files and resources related
- A node is an executable that run and communicates with the ROS2 framework
- Packages may contain multiple nodes



Creating the First Workspace

- Prerequisites-install colcon: sudo apt install python3-colcon-common-extensions
- Let's adhere to the standard and create the workspace in ~/ros_ws and add the directory src inside the workspace:
  mkdir -p ~/ros2_ws/src
  cd ~/ros2_ws
- Clone the examples: git clone https://github.com/ros2/examples src/examples -b humble
- Source the ROS 2 libraries (underlay): source /opt/ros/humble/setup.bash
- Run $ colcon build --symlink-install

```
ros2_ws
   ├── build
   ├── install
   ├── log
   └── src
```

Test the compiled examples

- Inside the workspace, source it (overlay):
  source install/setup.bash

- Run a subscriber node from the examples:
  ros2 run examples_rclcpp_minimal_subscriber subscriber_member_function

- In another terminal, source the workspace and run a publisher node:
  ros2 run examples_rclcpp_minimal_publisher publisher_member_function

ROS 2 Nodes

Publishers and Subscribers

Create the First Package

- Create a new package with the following command:

```
$ ros2 pkg create custom_pkg --build-type ament_cmake
```

- --node-name <name> allows to specify also the name of the file containing a node you will create

  e.g. ros2 pkg create custom_pkg --build-type ament_cmake --node-name custom_node

Writing the Publisher Node

ROS1-way

```
int main (int argc, char **argv) {

 rclcpp::init(argc, argv);

 auto node=std::make_shared<rclcpp::Node>(“
  my_publisher”);

 rclcpp::spin(node);
 rclcpp::shutdown();

 return 0;
}
```

ROS2-way

```
class MyPublisher : public rclcpp::Node {
 public:
  MyPublisher() : Node(“my_publisher”) {}
};

int main (int argc, char **argv) {
 rclcpp::init(argc, argv);
 auto node=std::make_shared<MyPublisher>();
 rclcpp::spin(node);
 rclcpp::shutdown();
 return 0;
}
```

Writing the Publisher Node

```
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

using namespace std::chrono_literals;


class MyPublisher : public rclcpp::Node {
public:
  MyPublisher() : Node("my_publisher") {...}
private:
  void timer_callback(){..}
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
}
```

Writing the Publisher Node

```
class MyPublisher : public rclcpp::Node {

public:
  MyPublisher() : Node("my_publisher") {...}

private:
  void timer_callback() {
    auto message=std_msgs::msg::String();
    message.data="Hello World!";
    publisher_->publish(message);
  }
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
}
```

Writing the Publisher Node

```
class MyPublisher : public rclcpp::Node {
public:
  MyPublisher() : Node("my_publisher") {
    publisher_ = this->create_publisher<std_msgs::msg::String>(TOPIC_NAME, 10);
    timer_ = this->create_wall_timer(500ms, std::bind(&MyPublisher::timer_callback, this));
  }
private:
  void timer_callback(){
    auto message=std_msgs::msg::String();
    message.data="Hello World!";
    publisher_->publish(message);
  }
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
}
```

Naming your topic

There are rules to look out when assigning a name to an interface:

- It must not be empty.
- Must only contain alphanumeric symbols and underscores '_' or the forward slash '/'; the latter two should never be repeated and never be at the end of the name.
- Can start with the prefix "~/", the private namespace substitution character (you don't really need it).
- Can use curly brackets '{}' for name substitutions (e.g., "{node}/chatter").

Some topic names are by convention bounded to specific message types and usages. This is not an enforced rule, but to avoid confusion it's good rule to respect it (e.g., 'initialpose', 'joint_states', 'rosout', 'tf' and 'tf_static', etc.)

Compiling the Publisher Node

- Add dependencies to package.xml

```
<depend>std_msgs</depend>
<depend>rclcpp</depend>
```

- Add dependencies to CMakeLists.txt

```
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)

add_executable(mytalker src/publisher_node.cpp)
ament_target_dependencies(mytalker rclcpp std_msgs)

install(TARGETS
     mytalker
     DESTINATION lib/${PROJECT_NAME})
```

Compiling the Publisher Node

- It’s good practice to run rosdep in the root of your workspace (ros2_ws) to check for missing dependencies before building:

```
sudo apt install python3-rosdep2
rosdep update
rosdep install -i --from-path src --rosdistro humble -y
```

- In the workspace directory, compile with:

```
colcon build
```

- Source the new files so that the client library can find the new nodes:

```
source install/setup.bash
```

Run the Publisher Node

- Run the publisher node:

  $ ros2 run custom_pkg mytalker

- Check the name:

  ros2 node list

- Listen to the messages sent on the topic:

  $ ros2 topic list

  $ ros2 topic echo /topic

  $ ros2 topic hz /my_topic

How to instantiate a ROS 2 node #1

```cpp
#include "rclcpp/rclcpp.hpp"
int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);
    auto node = std::make_shared<rclcpp::Node>("
    best_node_name");
    rclcpp::spin(node);
    rclcpp::shutdown();
    return 0;
}
```

- The node is created as a **shared pointer** to a rclcpp::Node class instantiated with your favourite name.
- This pointer can now be used to read parameters, create publishers, subscribers, and all the other fancy features.
- This solution can prove to be problematic and **should be avoided**!

How to instantiate a ROS 2 node #2

```cpp
#include "rclcpp/rclcpp.hpp"

class OurClass : public rclcpp::Node {
  public:
    OurClass() : Node("best_node_name") {
      // Some constructor
    }
}

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);
    auto node = std::make_shared<OurClass>();
    rclcpp::spin(node->get_node_base_interface());
    rclcpp::shutdown();
    return 0;
}
```

- **Separation**: The custom OurClass class, which inherits from the Node class, is easy to isolate it into a library that is independent from the execution of a process:
  - The node class contains the *functionality*.
  - The main only bootstraps ROS and runs the Node.

- **Modularity** and **reusability**:
  - The Node class can be reused in different executables or launch configurations.
  - If you change the node's logic, you don't touch main.cpp.
  - You might spin multiple nodes in one process (using executors).

- **Testing**:
  - You can unit-test the Node class directly (without spinning ROS)

How to instantiate a ROS 2 node #3

```cpp
#include "rclcpp/rclcpp.hpp"

namespace our_namespace
{

class OurClass : public rclcpp::Node {
  public:
    OurClass(const rclcpp::NodeOptions & options) : Node("
    best_node_name", options) {
      // Some constructor
    }
}

}
#include "rclcpp_components/register_node_macro.hpp"
RCLCPP_COMPONENTS_REGISTER_NODE(our_namespace::OurClass)
```

- **Components** are the recommended way to create nodes in ROS 2;

- A component can be:
  1. manually integrated into your program at compile time (in the CMakeList.txt);

```
add_library(our_component SHARED
  src/our_file.cpp)

# For Windows compatibility
target_compile_definitions(our_component
  PRIVATE "COMPOSITION_BUILDING_DLL")

ament_target_dependencies(our_lib
  "rclcpp"
  "rclcpp_components")

rclcpp_components_register_nodes(our_component "
    our_namespace::OurClass")
```

  2. loaded into an **executor** at runtime
     - using command line commands;
     - through a launchfile.

How to instantiate a ROS 2 node #3

By making the process layout a deploy-time decision the user can choose between:

- running multiple nodes in **separate processes** with the benefits of **process/fault isolation** as well as **easier debugging** of individual nodes;
- running multiple nodes in a **single process** with the **lower overhead** and optionally **more efficient communication** (see Intra Process Communication).

Exercise

- Re-write and compile the publisher node using separate files for class header, class implementation and main (my_publisher.hpp, my_publisher.cpp, my_publisher_node.cpp) .

- Re-write and compile the publisher node as component.

Useful Links

- CMake build system:
  https://docs.ros.org/en/humble/How-To-Guides/Ament-CMake-Documentation.html
  https://colcon.readthedocs.io/en/released/index.html
- Code style:
  https://docs.ros.org/en/humble/The-ROS2-Project/Contributing/Code-Style-Language-Versions.html
- Documentation, testing:
  https://docs.ros.org/en/humble/The-ROS2-Project/Contributing/Developer-Guide.html#documentation

Writing The Listener

```
class MySubscriber: public rclcpp::Node {
 public:
   MySubscriber() : Node("my_subscriber") {}
};

int main (int argc, char **argv) {
 rclcpp::init(argc, argv);
 auto node=std::make_shared<MySubscriber>();
 rclcpp::spin(node);
 rclcpp::shutdown();
 return 0;
}
```

Writing The Listener

```
class MySubscriber : public rclcpp::Node
{
  public:
    MySubscriber(): Node("my_subscriber")
    {
      subscription_ = this->create_subscription<std_msgs::msg::String>(
      "my_topic", 10, std::bind(&MySubscriber::topic_callback, this, _1));
    }

  private:
    void topic_callback(const std_msgs::msg::String & msg) const
    {
      RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg.data.c_str());
    }
    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
};
```

Compiling the Subscriber

- No new dependencies to package.xml and CMakeLists.txt needed (`rclcpp, std_msgs`);
- Add the executable and target for the subscriber node in the CMakeLists.txt:

```
add_executable(listener src/subscriber_member_function.cpp)
ament_target_dependencies(listener rclcpp std_msgs)

install(TARGETS
  mytalker
  mylistener
  DESTINATION lib/${PROJECT_NAME})
```

Exercise

- Re-write and compile the subscriber node using separate files for class header, class implementation and main (my_subscriber.hpp, my_subscriber.cpp, my_ subscriber_node.cpp) .

ROS 2 Nodes

Logging messages

Logging in ROS 2

- In a generic C++ program, it is common to use the "standard output stream" with the object std::cout.
- In ROS, a set of utilities were created for the same goal, but with added functionality.
- The **logging subsystem** in ROS 2 aims to deliver logging messages to a variety of targets, including:
  - To the console (if one is attached), for real-time visualisation;
  - To log files on disk (if local storage is available);
  - To the /rosout topic on the ROS 2 network.

Logging in ROS 2

The most common way of printing a message looks like:

```
RCLCPP_INFO ( node - > get_logger () , " Hello ROS ! " ) ;
```

This is composed of three parts:

- A macro (RCLCPP_INFO in this case);
- A logger object;
- The message that you would like to print.

The output is:

```
[ INFO ] [{ timestamp }] [{ logger name }]: " Hello ROS ! "
```

Choosing the Severity Level

Log messages have a **severity level** associated with them: DEBUG, INFO, WARN, ERROR or FATAL, in ascending order.

```
RCLCPP_DEBUG(node->get_logger(), "This is a detailed info for debugging");
RCLCPP_INFO(node->get_logger(), "This is an general information message");
RCLCPP_WARN(node->get_logger(), "This is a warning message");
RCLCPP_ERROR(node->get_logger(), "This is an serious issues message");
RCLCPP_FATAL(node->get_logger(), “The system is unusable message");
```

Choosing the Severity Level

Level	Purpose	Example (Manipulation)	Color
DEBUG	Detailed internal info, useful for debugging specific components. *Disabled by default.*	List of all detected objects on the table.	Green
INFO	Normal operation info — helps understand robot behavior.	Command received or object being grasped.	White
WARN	Unexpected events that may reduce performance; system still runs.	Using default parameter because custom value not found.	Yellow
ERROR	Something failed — robot cannot complete its task, needs recovery.	Object missing → abort task.	Red
FATAL	Critical failure — robot inoperable, no recovery possible.	Lost connection to motor controller.	Dark red

Changing the Severity Level

- A logger will only process log messages with severity at or higher than a specified level chosen for the logger.
- The default logging level in ROS 2 is **INFO**;
- You can change the logging level
  - from the command line:
    ros2 run my_package my_node_executable --ros-args --log-level DEBUG
  - in the code:
    node->get_logger().set_level(rclcpp::Logger::Level::Debug);
  - as an environmental variable:
    export RCUTILS_LOGGING_MIN_SEVERITY=DEBUG

Logging Macros in ROS 2

ROS 2 provides different logging options.

Logging style:
- RCLCPP_{DEBUG,INFO,WARN,ERROR,FATAL} - output the given printf-style message every time this line is hit
- RCLCPP_{DEBUG,INFO,WARN,ERROR,FATAL}_STREAM - output the given C++ stream-style message every time this line is hit

When logging:
- RCLCPP_{DEBUG,INFO,WARN,ERROR,FATAL}_ONCE - output only the first time this line is hit
- RCLCPP_{DEBUG,INFO,WARN,ERROR,FATAL}_EXPRESSION - output only if the given expression is true
- RCLCPP_{DEBUG,INFO,WARN,ERROR,FATAL}_FUNCTION - output only if the given function returns true
- RCLCPP_{DEBUG,INFO,WARN,ERROR,FATAL}_SKIPFIRST - output all but the first time this line is hit
- RCLCPP_{DEBUG,INFO,WARN,ERROR,FATAL}_THROTTLE - output no more than the given rate in integer milliseconds

Also combinations of the macros are possible. The full list is available here.

ROS 2 Nodes

Callbacks and Executors

The callback

- It is that part of the code that can be executed only by an executor, or by an invocation of one of the spinning functions (*spin, spin once,* and *spin until future complete*).
- You define what the callbacks do, but you can not strictly enforce the moment in which they will be executed.
- Examples:
  - Subscription callbacks (called whenever a message arrives);
  - Timer callbacks (called at a certain frequency);
  - Service calls, including parameters callbacks;
  - Received client responses.

The executors

- An **executor** is an event loop.
  - It checks all the nodes you've given it for *work to do* (incoming messages, timer events, service requests, etc.)
  - It then runs the corresponding callbacks according to its scheduling rules.
- For example, the code: `rclcpp::spin(node);` is just a shorthand for creating a **SingleThreadedExecutor**, adding the node to it, and spinning.
- When executors are **important**:
  - Single-threaded is safe but may be too slow; multi-threaded gives concurrency but needs careful coding;
  - You can run multiple nodes in the same process by adding them to one executor;
  - In real-time robotics, choosing the right executor affects timing guarantees.

The executors

- **SingleThreadedExecutor:**
  - Default if you just call rclcpp::spin(node).
  - Processes one callback at a time, in a single thread.
  - Simple, avoids race conditions, but may block if a callback is slow.

```
rclcpp::executors::SingleThreadedExecutor exec;
exec.add_node(node1);
exec.add_node(node2);
exec.spin();
```

- **StaticSingleThreadedExecutor** (introduced for deterministic, real-time-ish scenarios)
  - Similar to SingleThreadedExecutor, but builds a static schedule of what to check, avoiding some overhead.

The executors

- **MultiThreadedExecutor**
  - Can run multiple callbacks in parallel, using a thread pool.
  - Useful when callbacks are slow or blocking (e.g. heavy computations, waiting for I/O).
  - Careful: you need thread-safe code (shared variables → mutexes).

```
rclcpp::executors::MultiThreadedExecutor exec(rclcpp::ExecutorOptions(), 4); // 4 threads
exec.add_node(node1);
exec.add_node(node2);
exec.spin();
```

The executors

Example with two nodes.

```cpp
#include "rclcpp/rclcpp.hpp"
#include "talker.hpp"
#include "listener.hpp"

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

  auto talker   = std::make_shared<Talker>();
  auto listener = std::make_shared<Listener>();

  // Option A: single-threaded (one callback at a time)
  // rclcpp::executors::SingleThreadedExecutor exec;

  // Option B: multi-threaded (callbacks may run concurrently)
  rclcpp::executors::MultiThreadedExecutor exec;

  exec.add_node(talker);
  exec.add_node(listener);
  exec.spin();

  rclcpp::shutdown();
  return 0;
}
```

ROS 2 Nodes

Custom Interfaces

Standard and Custom ROS 2 Interfaces

✅ **Use *standard* message types (from common packages) when possible**

- **Easier integration** — works seamlessly with other nodes and teams.
- **Fewer dependencies** — no need to ship custom `.msg` files.
- **Faster development** — no extra compilation or maintenance.

❌ **Avoid misusing standard messages**

Don't force data into an existing type *just because it fits syntactically*. Use a **custom message** if:

- The semantics don't match (e.g., voltages ≠ `geometry_msgs/Quaternion`).
- You can't fill all required fields — receivers can't tell if defaults are meaningful.

Custom ROS 2 Interfaces

To create custom messages/services/actions follow these guidelines to keep your system clean, reusable, and compatible:

1. Use dedicated *interface packages*

- Define message/service/action files **in a separate package**.
- That package should contain **only interface definitions** — no source code.
- Prevents circular dependencies.
- Easier distribution (users only need `.msg` files to communicate).

2. Naming convention

- End interface package names with **`_msgs`** or **`_interfaces`**:
  Example: `my_robot_msgs`, `navigation_interfaces`

3. Represent optional values with vectors

- Clear semantics for “optional” fields.
- Use an **empty vector** to mean “value not provided”:
  Example: float64[<=1] optional_value_name

Custom Messages and Services

- The .msg files are required to be placed in directories called msg inside your package: mkdir msg srv

- In msg (or srv), create a custom definition and fill the data types:
  touch MyMsg.msg (or MySrv.srv)

e.g.
geometry_msgs/Point center
float64 radius

```
int64 a
int64 b
int64 c
---
int64 sum
```

Compile the Custom Message

```
find_package(geometry_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)


rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/MyMsg.msg“
  “srv/MySrv.srv"
DEPENDENCIES geometry_msgs # Add packages that above messages depend on, in this case geometry_msgs for My_msg.msg)
ament_export_dependencies(rosidl_default_runtime)
```

Compile the Custom Message

```
# My_msg dependencies
<depend>geometry_msgs</depend>

# Compile-time and Run-time dependencies to generate My_msg
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

Build and Verify

- Compile only a package:
  colcon build --packages-select <pkg_name>

- Souce the overlay to see the new message:
  source install/setup.bash

- Verify that your interface creation worked:
  ros2 interface show <pkg_name>/msg/MyMsg
  ros2 interface show <pkg_name>/srv/MySrv

Using Custom Msgs/Srvs in your Package

- If the msg is defined in an **external package** (e.g. custom_msgs), to link it to your node:
  - In CMakeLists.txt we need to add it as a dependency:

```
find_package(custom_msgs REQUIRED)


add_executable(my_publisher_node src/my_publisher_node.cpp)
ament_target_dependencies(my_publisher_node custom_msgs <other dependencies>)
```

  - Also package.xml should be informed

```
<depend>custom_msgs</depend>
```

  - Include the headers which have the form

```
#include "<package_name>/msg/<interface_file_name>.hpp"
#include "custom_msgs/msg/my_msg.hpp"
```

Using Custom Msgs/Srvs in your Package

- If the msg is defined in the **same package** (e.g. custom_pkg), to link it to your node:
  - In CMakeLists.txt:

```
add_executable(my_publisher_node src/my_publisher_node.cpp)
rosidl_get_typesupport_target(cpp_typesupport_target ${PROJECT_NAME}
rosidl_typesupport_cpp)
target_link_libraries(my_publisher_node "${cpp_typesupport_target}")
ament_target_dependencies(my_publisher_node <other dependencies>)
```

  - No additions to the package.xml are required because the generated files are in the same pkg!
  - Include the headers which have the form

```
#include "<package_name>/msg/<interface_file_name>.hpp"
#include "custom_pkg/msg/my_msg.hpp"
```

Exercise

- Use the new custom message in the Publisher node by adding a new Publisher broadcasting on a new topic with message type MyMsg.msg.

ROS 2 Nodes

Parameters

Setting the Parameters

```
class MyPublisher : public rclcpp::Node {
public:
  MyPublisher() : Node("mypublisher"){
    publisher_ = this->create_publisher<std_msgs::msg::String>(TOPIC_NAME, 10);
    timer_ = this->create_wall_timer(500ms, std::bind(&MyPublisher::timer_callback, this));
  }
private:
  void timer_callback(){
    auto message=std_msgs::msg::String();
    message.data="Hello World!";
    publisher_->publish(message);
  }
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
}
```

Setting the Parameters

```
class MyPublisher : public rclcpp::Node {
public:
  MyPublisher() : Node(“mypublisher”), topic_("default_topic") {
    this->declare_parameter("topic", topic_);
    publisher_ = this->create_publisher<std_msgs::msg::String>(topic_, 10);
    timer_ = this->create_wall_timer(500ms, std::bind(&MyPublisher::timer_callback, this));
  }
private:
  void timer_callback(){
    auto message=std_msgs::msg::String();
    message.data=“Hello World!”;
    publisher_->publish(message);
  }
  std::string topic_;
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
}
```

Setting the Parameters

```
void MyPublisher::timer_callback(){
  auto message=std_msgs::msg::String();
  message.data="Hello World!";
  publisher_->publish(message);

  std::string newTopic = this->get_parameter("topic").as_string();
  if (newTopic != this->topic_) {
    this->topic_ = newTopic;
    this->publisher_.reset();
    this->publisher_ = this->create_publisher<std_msgs::msg::String>(this->topic_, 10);
  }
}
```

Setting the Parameters

- Get the parameter:

  $ ros2 param get /mypublisher topic

- Listen to verify that the data is arriving correctly:

  $ ros2 topic echo default_topic

- Change the parameter:

  $ ros2 param set /mypublisher topic important_topic

Why Parameters?

- Parameters are "*settings*" for the node
- Allow to dynamically change the configuration of a node
  - Used in launch files
  - Depend on the **context** in which the node is executed
- Sometimes you don't know the actual value of a variable at *compile time*
- They are **associated** to the node
  - ROS1 used the Parameter Server
  - ROS2 access to a local dictionary, eliminating the overhead of network access *

* it's possible to use topics and service to subscribe to other nodes' parameters and monitor their changes, but should be avoided.

How to Access Parameters

- Declare a parameter:

```
auto value = this->declare_parameter("topic_name", "default_value");
```

- Each parameter can be declared **only once**
- Initialization of parameter:
  - default_value
  - Override (declare_parameter function returns the value of the parameter updated with the overrides):
    - Launch file
    - YAML file (saved with dump command in *config* directory):

      ros2 run <pkg> <node> --ros-args --params-file <path to YAML file>

    - Arguments provided through command line:

      ros2 run <pkg> <node> --ros-args -p <param_name>:=<param_value>

How to Access Parameters

- Declare a parameter:

```
auto value = this->declare_parameter("topic_name", "default_value");
```

- Retrieve the parameter value:

```
auto value = this->get_parameter("topic _name ");
```

- Change the parameter:

```
this->set_parameter("topic _name", "new_value");
```

Parameter Types and Description

- Parameters have types:
  - double
  - integer
  - string
  - list → std::vector
- Can have a description:
  - read_only: if true, the parameter value **cannot** be changed at run-time
  - dynamic_typing: if true, it's possible to **change** the parameter type
  - FloatingPointRange/IntegerRange: limit the **range** of the parameter's value
  - additional_constraints: additional constraints

```
auto param_desc = rcl_interfaces::msg::ParameterDescriptor{};
param_desc.description = "This parameter is mine!";

this->declare_parameter("my_parameter", "world", param_desc);
```

Parameter Types and Description

- Parameters have types:
  - double
  - integer
  - string
  - list → std::vector
- Can have a description:
  - read_only: if true, the parameter value **cannot** be changed at run-time
  - dynamic_typing: if true, it's possible to **change** the parameter type
  - FloatingPointRange/IntegerRange: limit the **range** of the parameter's value
  - additional_constraints: e.g. your parameter is a string that can assume only specific values, these could be listed here in a comma separated list.

If your parameter controls important functions and will be updated at run-time, then remember to set the range!

```
auto param_desc = rcl_interfaces::msg::ParameterDescriptor{};
param_desc.description = "This parameter is mine!";

this->declare_parameter("my_parameter", "world", param_desc);
```

Handling Default Parameters in ROS 2

PROBLEM: If you always assign *default values* to parameters, you can't tell whether a parameter was **overridden by the user** or is still using the **default**.

EXAMPLE: Typos or wrong namespaces in YAML files cause silent failures. The node starts, but behaves incorrectly.

SOLUTIONS:

Approach	Behavior	Pros / Cons
1️⃣ **No defaults**	No override → exception at runtime.	✅ Forces correct parameter definition. ❌ Node may crash if parameters missing.
2️⃣ **Warn on missing override**	Use default but log a **warning**.	✅ Prevents crashes. ✅ Makes issues visible. 💡 Can be done by extending `declare_parameter()`.

Dynamic Parameters Updates

- Dynamic parameters updates
  - Repeatedly call `get_parameter()`
  - Subscribe to parameter changes
    - Self
    - Other's

- Having too many parameters might create confusion, so don't abuse them!
  - Use namespaces or parameter groups to group them;
  - Try to generate mechanisms for **automatic parameter tuning.**

Dynamic Parameter Updates

Use ParameterEventHandler → subscribes to parameter changes:

```
// Create a parameter subscriber to detect changes in the parameters
this->param_subscriber_ = std::make_shared<rclcpp::ParameterEventHandler>(this);

// Define the callback to be called when the parameter changes
auto cb = [this](const rclcpp::Parameter & p){
  this->timer_.reset();
  this->timer_ = this->create_wall_timer(std::chrono::milliseconds(p.as_int()), std::bind(&MyPublisher::timer_callback, this));
};
// Specify to which parameter changes to subscribe and which callback to call
this->param_cb_handle_ = this->param_subscriber_->add_parameter_callback("rate", cb);
...
private:
  std::shared_ptr<rclcpp::ParameterEventHandler> param_subscriber_;
  std::shared_ptr<rclcpp::ParameterCallbackHandle> param_cb_handle_;
```

Parameters Without Declaration

- ROS1 allowed for parameters **without declaration**
- Not mandatory in ROS2, but *strongly discouraged* (aka, declare them)
  - Static typing
  - Additional constraints
  - Control over multiple declarations
  - Less prone to errors