

---

# **ENPM702**

## INTRODUCTORY ROBOT PROGRAMMING

### **L9: Robot Operating System (ROS) - Part I**

v1.0

**Lecturer:** Z. Kootbally

**Semester/Year:** Summer/2025



MARYLAND APPLIED  
GRADUATE ENGINEERING

# Table of Contents

- ◎ Learning Objectives
- ◎ Prerequisites
- ◎ Prerequisites
- ◎ What is ROS?
  - ◎ Debian Packages vs. Source Installation
- ◎ Where is ROS 2 Used?
  - ◎ ROS vs. Alternative Frameworks
- ◎ ROS 2 Architecture
  - ◎ Core Components
  - ◎ ROS 2 Daemon
  - ◎ Discovery
  - ◎ Distributed vs. Monolithic Architecture
  - ◎ Communications
    - ◎ Publish/Subscribe Model
    - ◎ Analogy
- ◎ The colcon Build Tool
- ◎ ROS 2 Workspaces
  - ◎ Workspace Structure
  - ◎ Workspace Sourcing and Overlays
- ◎ ROS 2 Packages
  - ◎ Package Creation and Structure
  - ◎ Package Manifest
  - ◎ Build Configuration
- ◎ Writing Your First Node
  - ◎ Simple Functional Node
  - ◎ Object-Oriented Node Design
  - ◎ Advanced Build Options
  - ◎ Node Lifecycle and Best Practices
- ◎ Node Spinning
- ◎ ROS 2 Messages
  - ◎ Running Nodes
  - ◎ Standard Messages (std\_msgs)
  - ◎ Message Introspection
  - ◎ Using Messages in C++
  - ◎ Message Dependencies
  - ◎ Publishing Test Messages
- ◎ Publishers
  - ◎ Creating a Publisher
  - ◎ Initialize the Publisher
  - ◎ Publish Messages
- ◎ Subscribers
  - ◎ Creating a Subscriber Node
  - ◎ Initialize the Subscriber
  - ◎ Process Messages
- ◎ Communication Scenarios
  - ◎ Scenario Analysis
- ◎ Next Class

## ☰ Changelog

---

- v1.0: Original version.

## Learning Objectives

---

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

- **Understand ROS 2 Architecture:** Explain the distributed system design and core components (nodes, topics, messages).
- **Set Up ROS 2 Development Environment:** Create and configure workspaces, understand package structure and dependencies.
- **Build ROS 2 Packages:** Use colcon build system, manage dependencies, and understand workspace overlays.
- **Create ROS 2 Nodes:** Implement both functional and object-oriented node designs using `rclcpp`.
- **Work with Messages:** Use standard message types, understand message introspection, and handle message dependencies.
- **Implement Publishers:** Create publishers to send messages to topics with appropriate QoS settings.
- **Implement Subscribers:** Create subscribers with callbacks to receive and process messages from topics.
- **Apply Node Spinning:** Understand different spinning methods and keep nodes alive for message processing.
- **Analyze Communication Patterns:** Debug publisher-subscriber systems and understand message flow scenarios.

# Prerequisites

Add the following function to your shell configuration file:

For .zshrc users:

```
function ros702 {  
    #ws="/home/zeid/Documents/ros702_ws"  
    source /opt/ros/jazzy/setup.zsh  
    #source "${ws}/install/setup.zsh"  
    #eval "$(register-python-argcomplete ros2)"  
    #eval "$(register-python-argcomplete colcon)"  
    #cd ${ws}  
}  
ros702
```

For .bashrc users:

```
function ros702 {  
    #ws="/home/zeid/Documents/ros702_ws"  
    source /opt/ros/jazzy/setup.bash  
    #source "${ws}/install/setup.bash"  
    #source  
    #→ /usr/share/colcon_argcomplete/hook/colcon-argcomplete.bash  
    #cd ${ws}  
}  
ros702
```

## ☰ After Adding the Function

- Reload your shell: `source ~/.bashrc` or `source ~/.zshrc`
- Use the function: `ros702`
- This will setup your ROS 2 environment and navigate to your workspace.

# What is ROS?

The Robot Operating System (ROS) is an open-source framework designed for developing, building, and deploying robotic applications.

- Prototypes originated from Stanford AI research and officially released by Willow Garage in 2007.
- ROS 2 was redesigned from the ground up and released in 2017.
- Currently maintained by [Open Robotics](#)



Despite its name, ROS is not a standalone operating system; rather, it is a flexible set of software tools, libraries, and conventions that help developers create complex and modular robot software.



Watch the video at <https://ros.org/>

## Resources

---

- Official website: <http://wiki.ros.org/>
- ROS 2 Documentation: <https://docs.ros.org/en/humble/>
- Tutorials: <http://wiki.ros.org/ROS/Tutorials>
- Package index: <https://index.ros.org>
- Community forum: <https://discourse.ros.org/>
- ROS Enhancement Proposals: <https://www.ros.org/reps/>
- GitHub organizations: [ros2](#), [ros-planning](#), [ros-perception](#)

## Debian Packages

- Located in:  /opt/ros/<distribution>/share
- Installed via:  
 `sudo apt install ros-<distribution>-*`
- Use for stable and production systems.
- Automatic dependency management.

## Source Installation

- Located in: Your workspace  src directory
- Built with:  `colcon build`
- Use for development and customization.
- Easy to modify and debug.



Can override Debian packages.



Never modify packages installed in  /opt/ros/, instead copy packages to your workspace and modify the copies.

# Where is ROS 2 Used?

## Transportation:

- Autonomous vehicles ([CARLA](#), [Autoware](#))
- Delivery drones ([Amazon Prime Air](#), [UPS Flight Forward](#))
- Maritime autonomous systems ([Kongsberg Maritime](#))
- Railway automation ([Siemens Mobility](#))

## Manufacturing:

- Industrial robotic arms ([KUKA](#), [ABB Robotics](#))
- Quality inspection ([Cognex](#), [Keyence](#))
- Collaborative robots ([Universal Robots](#), [Rethink Robotics](#))
- Warehouse automation ([Amazon Robotics](#), [Berkshire Grey](#))

## Resources

---

- [ROS Robotics Companies](#)

## Specialized Domains:

- Medical robots ([Intuitive Surgical](#), [Stryker Mako](#))
- Space exploration ([NASA JPL](#), [NASA Astrobee](#))
- Agricultural automation ([Blue River Technology](#), [John Deere](#))
- Search and rescue ([Boston Dynamics Spot](#), [ANYbotics](#))
- Research and education ([TurtleBot](#), [Husarion](#))

## Emerging Areas:

- Home service robots ([iRobot](#), [Savioke](#))
- Entertainment ([Anki](#), [SoftBank Pepper](#))
- Security ([Knightscope](#), [Cobalt Robotics](#))
- Environmental monitoring ([Ocean Infinity](#), [Clearpath Robotics](#))

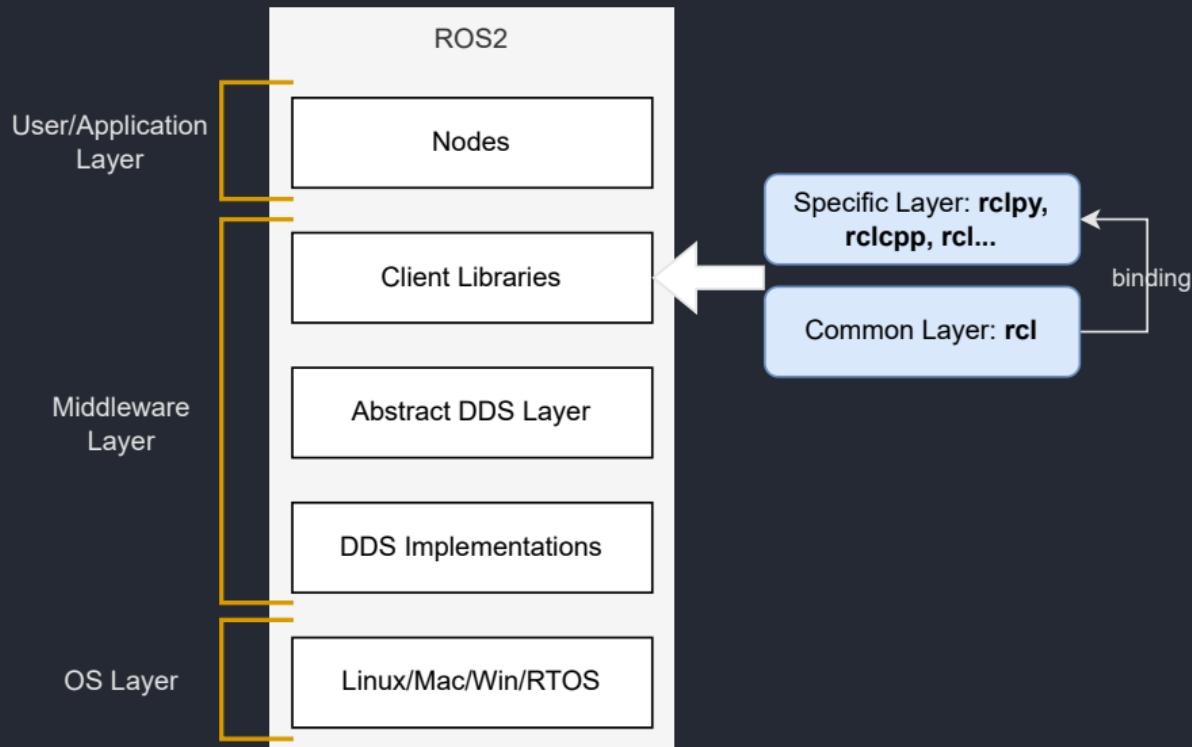
## Where is ROS 2 Used? ► ROS vs. Alternative Frameworks

Framework/Solution	Key Features	Differences
<u>Microsoft Robotics Developer Studio (MRDS)</u>	Windows-based, visual programming, simulation support	Primarily for Windows
<u>Open Robot Control Software (ORoCoS)</u>	Real-time control, deterministic execution	Focuses on industrial control; ROS 2 offers more flexibility and integration ease
<u>Mission Oriented Operating Suite (MOOS)</u>	Marine robotics, real-time autonomy	Specialized for marine applications, whereas ROS 2 is more general-purpose
<u>Yet Another Robot Platform (YARP)</u>	Inter-process communication, modularity	Emphasizes communication, whereas ROS 2 provides a more standardized middleware layer with DDS
Custom-built solutions	Optimized for specific applications	Lack modularity, community support, and extensibility that ROS 2 provides

Choose ROS 2 when you need:

- Rapid prototyping and development.
- Large ecosystem of existing solutions.
- Community support and collaboration.
- Standard interfaces and protocols.

# ROS 2 Architecture



## ☰ Core Components

---

- Nodes: Individual processes performing specific computations or tasks
- Topics: Named channels for asynchronous data streaming (Publisher/Subscriber model)
- Services: Synchronous request-response communication between nodes
- Actions: Long-running, interruptible tasks with feedback
- Interfaces: Data structures for messages, services, and actions
- Launch files: Automation scripts for starting multiple nodes with configuration

## ☰ Quality of Service (QoS)

---

- **Reliability**: Best effort vs. reliable delivery
- **Durability**: Transient vs. volatile data storage
- **History**: Keep last N messages vs. keep all
- **Deadline**: Maximum expected time between messages

### What is the ROS 2 Daemon?



A background process that caches ROS 2 discovery information to optimize CLI performance.

### ☰ Key Functions

---

- **Discovery Cache:** Maintains a live cache of nodes, topics, services, and actions.
- **Performance Boost:** CLI commands return instantly instead of waiting for discovery.
- **Network Efficiency:** Reduces repeated DDS discovery traffic.
- **Automatic Management:** Starts automatically when needed.

### ☰ Common Commands

---

---

- **ros2 daemon status** - Check if daemon is running.
- **ros2 daemon stop** - Stop the daemon (commands will be slower).
- **ros2 daemon start** - Manually start (usually automatic).

### When to Restart the Daemon



- `ros2 topic list` shows outdated information
- After changing `ROS_DOMAIN_ID`
- Network configuration changes



The daemon only affects CLI tool performance. Nodes communicate directly without it.

## ☰ Node Discovery Mechanism

---

- **ros2 daemon** – Background process that caches information about active ROS 2 nodes, topics, services, and actions within a network.
  - Provides fast lookup of network topology.
  - Reduces discovery latency for new nodes.
  - Can be disabled for minimal overhead:  **ros2 daemon stop**
- **DDS(Data Distribution Service)** – ROS 2's communication middleware providing automatic discovery
  - Implementations: Fast DDS (default), Cyclone DDS, RTI Connext DDS
  - See [REP-2000](#) for DDS vendor support by distribution
  - Check your DDS implementation:  **ros2 doctor --report**
  - Switch DDS:  **export RMW\_IMPLEMENTATION=rmw\_cyclonedx\_cpp**



DDS discovery uses UDP multicast by default. In network environments where multicast is blocked, you may need to configure static discovery or use different network settings.

## ☰ Traditional Robotics: Monolithic Architecture

---

### ■ Single Program Approach:

- One large C++ program handles everything.
- All sensors, actuators, and algorithms in one codebase.
- Tight coupling between components.
- Single point of failure.



### Problems with Monolithic Design:

- Change sensor? Recompile everything.
- Add feature? Risk breaking existing code.
- Debug issue? Entire system complexity.
- Team development? Code conflicts.

### Monolithic Robot Program

**main.cpp**

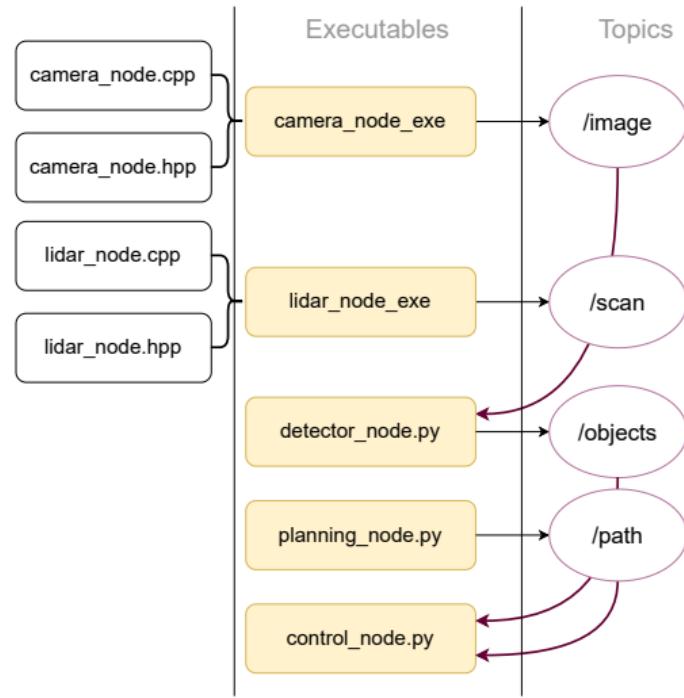
Camera Driver  
Lidar Driver  
Motor Control  
Path Planning  
Object Detection  
Sensor Fusion  
State Machine  
Logger  
...everything else

## ☰ ROS Architecture: Distributed System

- Multiple Independent Programs:
  - Each component is a separate executable.
  - Different nodes can use different languages.
  - Loose coupling via message passing.
  - Start/stop components independently.
  - Fault isolation and recovery.

### Benefits of Distributed Design:

- Modular: Swap components easily.
- Scalable: Add nodes without changes.
- Robust: One crash doesn't kill system.
- Collaborative: Teams work in parallel.



## Real-World Example: Autonomous Vehicle

Instead of one massive program, an autonomous vehicle using ROS 2 runs many specialized programs:

### Perception Nodes

- `camera_driver (C+)`
- `lidar_driver (C+)`
- `radar_processor (Python)`
- `object_detector (Python/TF)`
- `lane_detector (C+/OpenCV)`
- `traffic_sign_reader (Python)`

### Planning Nodes

- `global_planner (C+)`
- `local_planner (C+)`
- `behavior_planner (Python)`
- `trajectory_optimizer (C+)`
- `collision_checker (C+)`

### Control Nodes

- `vehicle_controller (C+)`
- `motor_driver (C+)`
- `brake_controller (C+)`
- `steering_controller (C+)`
- `diagnostics_monitor (Python)`

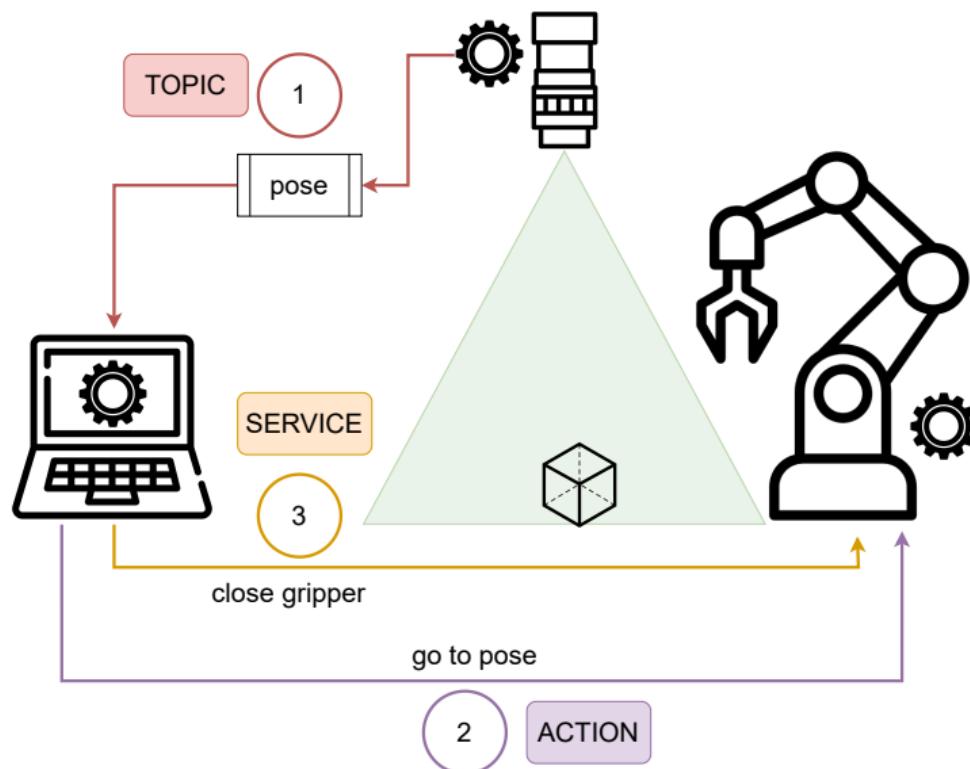
### Key Insight



Each node is a **separate executable program** that can be:

- Written in different languages (C<sup>+</sup>, Python, Rust, etc.)
- Developed by different teams or even different companies.
- Started, stopped, or replaced without affecting other nodes.
- Running on different computers in the same network.

## ☰ ROS 2 Communication Patterns





### When to Use Each Pattern

- **Topics:** Continuous data streams (sensor readings, robot state).
- **Services:** Request specific information or trigger actions (get current pose, save map).
- **Actions:** Long-running tasks with progress updates (navigate to goal, pick object).

### ☰ Publish/Subscribe Model

---

The publish/subscribe model allows nodes to communicate asynchronously via topics. A publisher sends messages to a specific topic, and any subscribers listening to that topic receive the messages.

### ☰ Key Benefits

---

- **Loose coupling:** Nodes don't need to know each other's existence
- **Scalability:** Easy to add new publishers or subscribers
- **Fault tolerance:** Nodes can start/stop independently
- **Flexible patterns:** Supports 1-to-many, many-to-1, many-to-many
- **Quality of Service:** Configurable reliability and performance guarantees

### ☰ Real-world Example

---

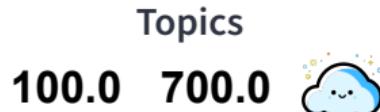
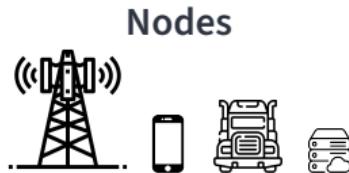
- Camera node publishes images to `t /camera/image_raw`.
- Object detection node subscribes to process images.
- Visualization node also subscribes to display images.
- Navigation node subscribes for obstacle detection.
- All work independently and can be started in any order.

### ☰ Analogy: Radio Towers

---

An analogy from Edouard Renard demonstrates how nodes, topics, and messages work together.

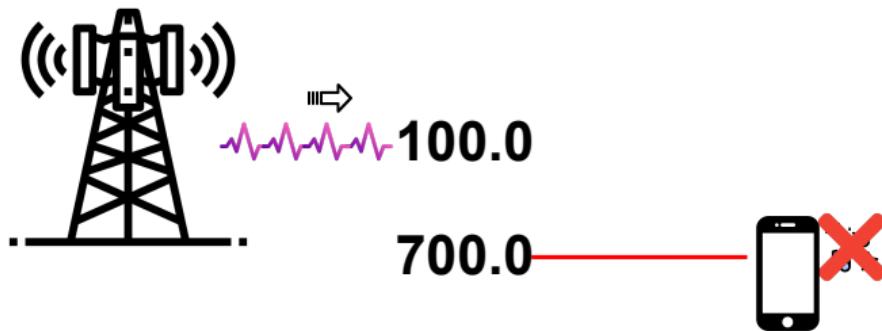
---





### Rule 1: Topic Name Matching

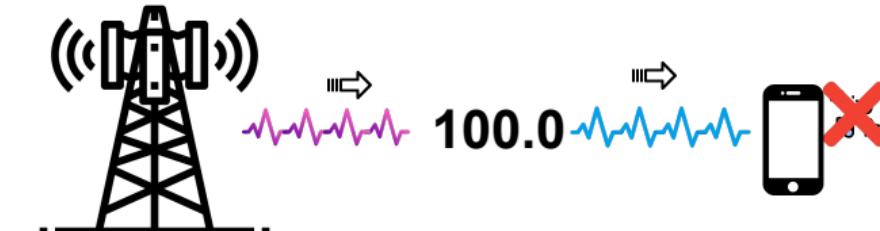
A subscriber must subscribe to the same topic that the publisher is using.



### Rule 2: Message Type Compatibility

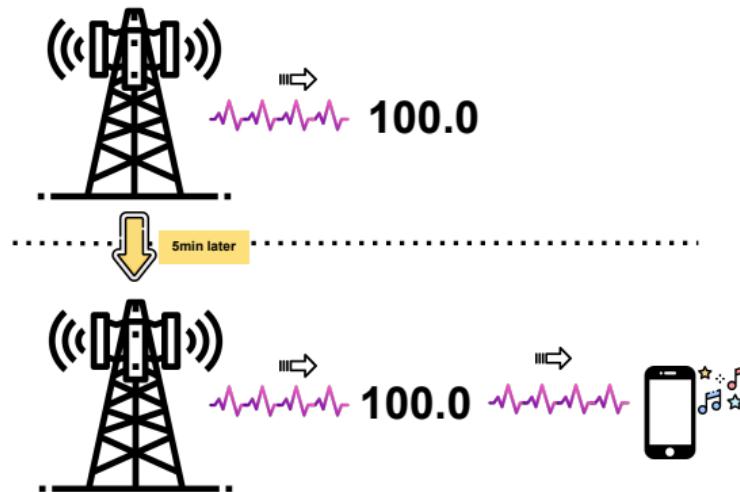


A subscriber must use the same message type as the publisher to receive the messages.



**Rule 3: Publishers Don't Wait**

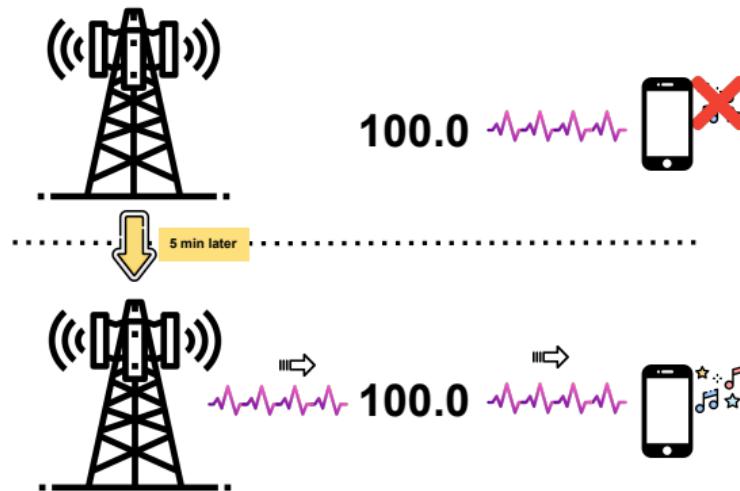
Publishers send messages regardless of whether subscribers are listening.





### Rule 4: Subscribers Don't Require Publishers

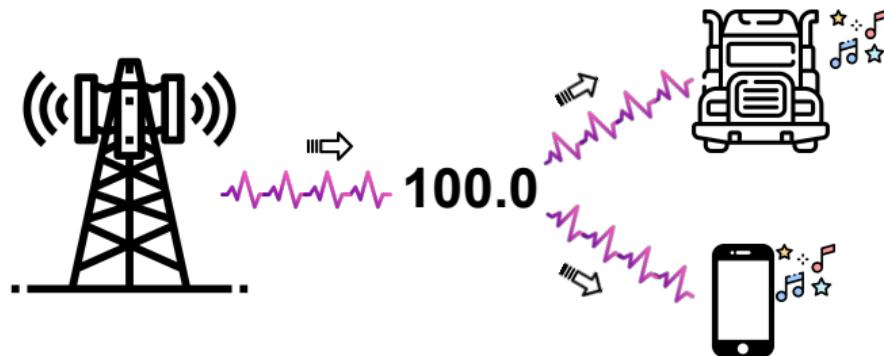
Subscribers can listen to topics even when no publisher is active.





### Pattern 1: One-to-Many Communication

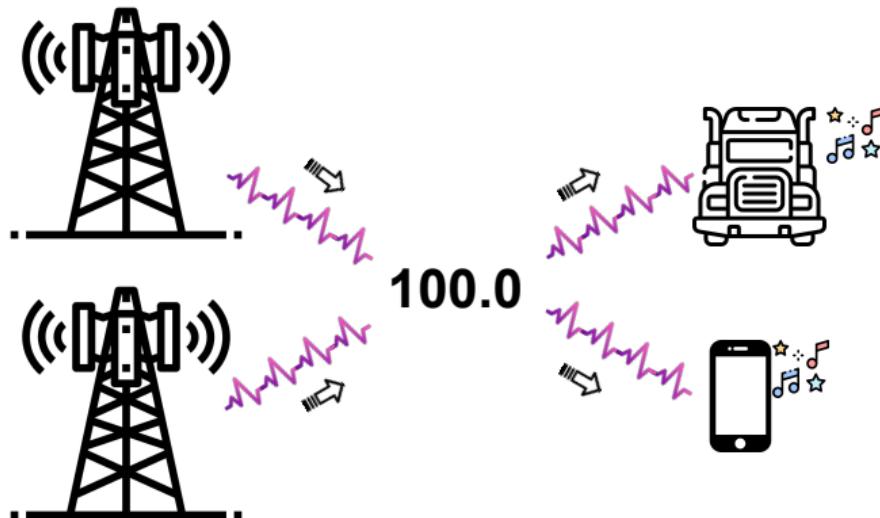
A single publisher can send data to multiple subscribers.



### Pattern 2: Many-to-One Communication

i

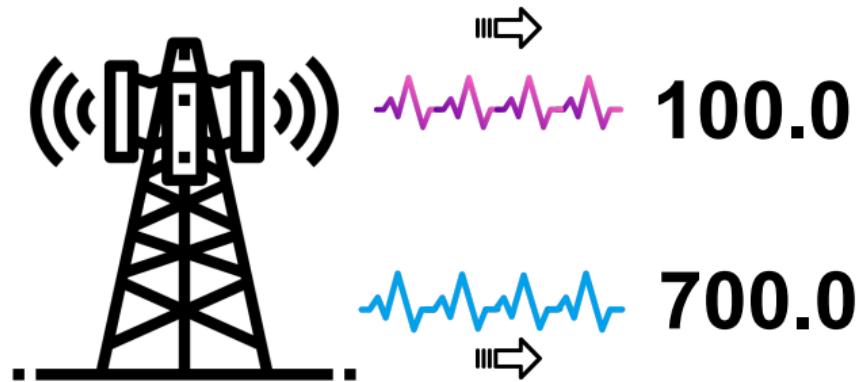
Multiple publishers can send data to a single topic (requires careful management).





### Pattern 3: Multi-Topic Publishers

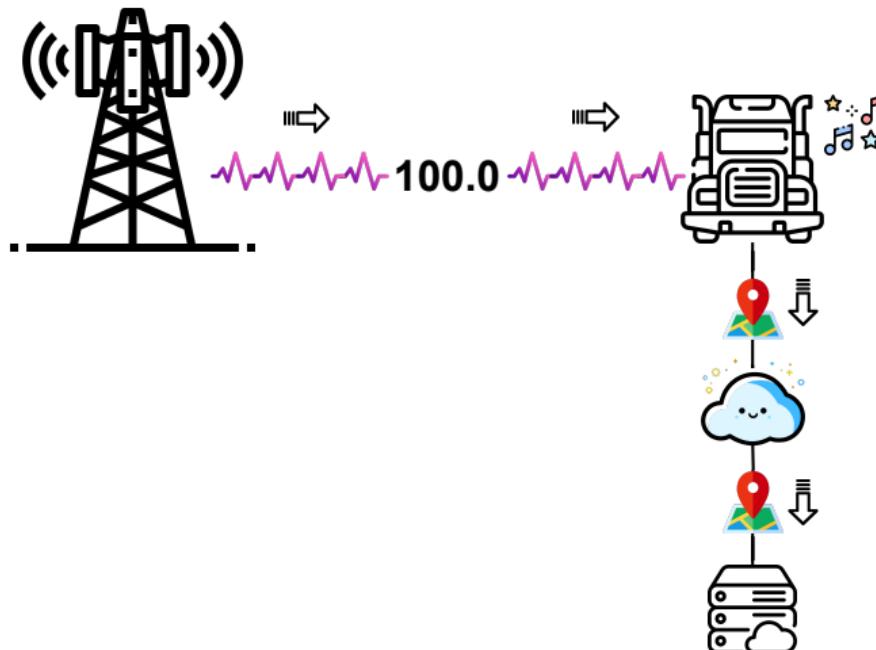
A single node can publish to multiple topics with different message types.



### Pattern 4: Bidirectional Communication



A node can act as both publisher and subscriber for different topics.



## ☰ Two Ways to Run ROS Nodes

1. `ros2 run <package> <executable>` starts *exactly one* node.
2. `ros2 launch <package> <launch file>` starts *one or more* nodes with configuration.

### When to Use Each Approach

- 
  - `ros2 run` – Quick testing, debugging single nodes, simple demonstrations.
  - `ros2 launch` – Production systems, complex multi-node applications, parameterized configurations.

### ☰ ros2 run - Single Node Execution

The `ros2 run <package> <executable>` command launches a single node where `<package>` is the ROS package name and `<executable>` is the executable file name.

- ☰ ■ ROS 2 includes the demo package  `demo_nodes_cpp` with executables for testing
- ☰ ■ The  `talker` sends messages and  `listener` receives them
- ☰ ■ Open two terminals and run:
  - ☰ ■ Terminal 1:  `ros2 run demo_nodes_cpp talker`
  - ☰ ■ Terminal 2:  `ros2 run demo_nodes_cpp listener`

### ☰ Understanding the Output

- Talker publishes “Hello World” messages with incrementing numbers.
- Listener receives and displays these messages.
- Messages flow through the  `/chatter` topic.
- This demonstrates basic publisher-subscriber communication.



Try stopping and restarting either node. Notice how the system gracefully handles nodes joining and leaving!

### ☰ ros2 launch - Multi-Node Systems

---

Launch files enable starting multiple nodes with complex configurations using

☒ **ros2 launch <package> <launch file>.**

### ☰ ToDo

---

Run both talker and listener with a single command:

☒ **ros2 launch demo\_nodes\_cpp talker\_listener.launch.py**

### ☰ Launch File Benefits

---

- Automation: Start entire robot systems with one command
- Configuration: Set parameters, namespaces, and remappings
- Coordination: Manage node dependencies and startup order
- Flexibility: Support different deployment scenarios

### ☰ Launch File Types

---

-  **\*.launch.py**: Python-based (most flexible and common)
-  **\*.launch.xml**: XML-based (declarative, good for simple cases)
-  **\*.launch.yaml**: YAML-based (human-readable configuration)

## ☰ ROS 2 Introspection and Debugging Tools

---

1. **ros2 node <option>** - Node management and information
  - **ros2 node list** - Lists all active nodes
  - **ros2 node info <node\_name>** - Detailed node information
  - **ros2 node kill <node\_name>** - Gracefully stop a node
2. **ros2 topic <option>** - Topic inspection and interaction
  - **ros2 topic list [-t]** - Lists topics [with message types]
  - **ros2 topic info <topic\_name> [-v]** - Topic details [verbose]
  - **ros2 topic echo <topic\_name>** - Monitor live messages
  - **ros2 topic hz <topic\_name>** - Measure message frequency
  - **ros2 topic pub <topic> <msg\_type> <values>** - Publish test messages
3. **ros2 interface <option>** - Message and service type inspection
  - **ros2 interface list** - All available message types
  - **ros2 interface show <msg\_type>** - Message structure details
4. **rqt\_graph** - Visual network topology display

# The colcon Build Tool

**colcon** (collective construction) is the official build tool for ROS 2, replacing the ROS 1 build tools (`catkin_make`, `catkin_tools`).

## ☰ Key Features

---

- **Language Agnostic:** Builds C++, Python, and other package types.
- **Parallel Execution:** Builds multiple packages simultaneously for speed.
- **Isolated Builds:** Each package built in its own space (no cross-contamination).
- **Extensible:** Plugin architecture for custom build types.
- **Cross-Platform:** Works on Linux, Windows, and macOS.

## ☰ Installation

---

- Install: `sudo apt install python3-colcon-common-extensions`
- Verify: `colcon version-check`

## ☰ Essential colcon Commands

---

- **colcon build**: Build all packages in workspace
- **colcon test**: Run tests for packages
- **colcon test-result**: Display test results summary
- **colcon list**: List packages in workspace
- **colcon graph**: Show package dependency graph

## ☰ Useful Build Options

---

- **--symlink-install**: Use symlinks for Python/config files (faster development)
- **--packages-select <pkg1> <pkg2>**: Build only specified packages
- **--packages-up-to <pkg>**: Build package and all its dependencies
- **--packages-above <pkg>**: Build all packages depending on specified package
- **--parallel-workers <N>**: Limit parallel build jobs (default: CPU cores)
- **--cmake-args <args>**: Pass arguments to CMake
- **--event-handlers console\_direct+**: Show live build output
- **--build-base <dir>**: Custom build directory location



**Memory-Limited Systems:** Use **--parallel-workers 1** on systems with limited RAM to prevent build failures due to memory exhaustion.



## Shell Integration: Enable colcon command completion by adding to your shell:

- Bash: `source /usr/share/colcon_argcomplete/hook/colcon-argcomplete.bash`
- Zsh: `eval "(register - python - argcompletecolcon)"`

# ROS 2 Workspaces

A ROS workspace is a directory structure that contains all the packages, dependencies, and build artifacts needed to develop and run ROS 2 applications.

- **Modular Development:** Organize code into logical, reusable packages.
- **Dependency Management:** Automatic resolution and building of package dependencies.
- **Overlay System:** Multiple workspaces can be layered for different projects.
- **Isolation:** Keep different projects and their dependencies separate.



### Create a workspace for this course

- Create the workspace directory: `mkdir -p ~/ros702_ws/src`
  - The `src` folder is where you will create custom ROS packages or clone existing ones from GitHub
- Navigate to workspace root: `cd ~/ros702_ws`
- Build the workspace: `colcon build`

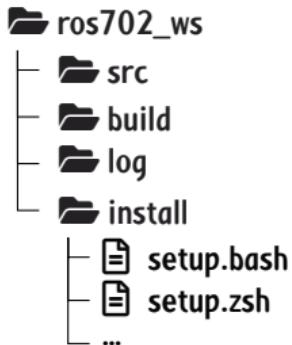


You must be in the workspace root directory to build it.



The build process scans `src` for packages and builds them. Even with no packages, build directories are created.

## ☰ Workspace Directory Structure



- **src**: Contains source code for custom and external packages.
- **build**: Stores intermediate build artifacts (object files, CMake cache).
- **log**: Contains build logs for debugging compilation issues.
- **install**: Houses final build products (executables, libraries, headers).
  - Includes `setup.bash/zsh` files for environment configuration.
  - This is what gets "sourced" to use your workspace.

### Build Process Flow

- Source code ( **src**) → Compilation ( **build**) → Installation ( **install**)
- Build logs stored in **log** for troubleshooting

### ☰ Workspace Sourcing

---

To use packages in your workspace, you must *source* the setup file to configure environment variables:

 `source ~/ros702_ws/install/setup.<bash|zsh>`



Update your shell function to include the workspace.

### ☰ Understanding Workspace Overlays

---

- **Base Layer:** System ROS installation (📁 /opt/ros/<distribution>)
- **Overlay Layer:** Your workspace (📁 ~/ros702\_ws)
- **Override Behavior:** Workspace packages override system packages with same name
- **Multiple Overlays:** Chain multiple workspaces for complex projects

#### Sourcing Order Matters!



Always source the base ROS installation first, then your workspace. Later sources take precedence over earlier ones.

# ROS 2 Packages

Software in ROS 2 is organized into packages - self-contained units that provide specific functionality and can be easily shared and reused.

- **Modularity:** Each package serves a specific purpose
- **Reusability:** Packages can be shared across different projects
- **Dependency Management:** Clear declaration of package dependencies
- **Build System:** Standardized building and installation process

### ☰ Package Build Systems

---

ROS 2 supports multiple build systems:

- **ament\_cmake**: For C++ packages (most common for performance-critical code)
- **ament\_python**: For Python packages (rapid prototyping, scripting)
- **ament\_cmake with Python**: Mixed language packages

### ☰ Build System Evolution

---

- ROS 1: Used *catkin* build system
- ROS 2: Uses *ament* - evolution of catkin with improved features
- **colcon**: Build tool that works with ament (replaces catkin\_make)
- **Advantages**: Better dependency resolution, parallel builds, improved testing

#### Key Insight



**ament** provides the build system rules, while **colcon** is the tool that executes the build process across multiple packages.

### ToDo

Create your first C++ ROS 2 package:

1. Navigate to source directory: `cd ~/ros702_ws/src`
2. Create the package:
  - `ros2 pkg create first_package --build-type ament_cmake --dependencies rclcpp`
  - Alternative shorter syntax: `ros2 pkg create first_package --dependencies rclcpp`
3. Build the package: `cd ~/ros702_ws && colcon build`
4. Source the workspace: `source ~/ros702_ws/install/setup.<bash|zsh>`
5. Verify creation: `ros2 pkg list | grep first_package`



**First-Time Build:** Always source your workspace after building a new package for the first time. Subsequent builds of the same package don't require re-sourcing.



**Troubleshooting:** If the package isn't listed, check: (1) Was the build successful? (2) Did you source the workspace? (3) Are you in the correct terminal session?

### ≡ C Package Structure

---



- **include/first\_package/**: Header files (.hpp) for public interfaces.
- **src/**: Source files (.cpp) for implementation.
- **CMakeLists.txt**: Build configuration (executables, libraries, dependencies).
- **package.xml**: Package metadata and dependencies.

### Package Discovery

Use  `ros2 pkg` commands to interact with packages:

-   `ros2 pkg list` - List all available packages.
-  `ros2 pkg executables <package_name>` - Show package executables.
-  `ros2 pkg xml <package_name>` - Display  `package.xml` content.
-  `ros2 pkg prefix <package_name>` - Show package installation path.

### ☰ Package Manifest (📄 package.xml)

The 📄 **package.xml** file is the package's "birth certificate": it defines metadata, dependencies, and build information.

### ☰ Critical Functions

1. **Dependency Resolution:** ament uses this to determine build order
2. **Automated Installation:** Install missing dependencies with:
  - ▀ `cd ~/ros2_ws`
  - ▀ `rosdep install --from-paths ./src --ignore-packages-from-source -y`
3. **Package Index:** Information for ROS Index and package distribution

### ☰ Essential Metadata to Update

- ▀ **<description>**: Clear, concise description of package functionality
- ▀ **<maintainer>**: Your name and email for support questions
- ▀ **<license>**: Legal terms (Apache-2.0, MIT, BSD, etc.)
- ▀ **<version>**: Semantic versioning (e.g., 1.0.0)
- ▀ Dependencies: **<depend>**, **<build\_depend>**, **<exec\_depend>**



Always edit 📄 **package.xml** immediately after creating a package!

### ☰ CMakeLists.txt Overview

---

📄 **CMakeLists.txt** is used by *ament\_cmake* to build C++ executables, libraries, and install files.

### ☰ Key CMake Components

---

- `cmake_minimum_required(VERSION 3.8)`: Minimum CMake version
- `project(first_package)`: Package name (must match package.xml)
- `find_package(rclcpp REQUIRED)`: Declare dependencies
- `add_executable()`: Create executable targets
- `ament_target_dependencies()`: Link ROS dependencies
- `install()`: Specify what to install and where
- `ament_package()`: Finalize package setup (must be last!)



**Important:** Every time you add a new node or modify build targets, you must edit **CMakeLists.txt**. This tells the build system how to compile and install your code.

# Writing Your First Node

Let's create a simple node that demonstrates the basic structure and lifecycle of a ROS 2 C++ node.

- **Goal:** Create a node that prints “Hello from ROS 2!” to the terminal.
- **Concepts:** Node initialization, logging, and proper shutdown.
- **Build Process:** CMake configuration and compilation.
- **Execution:** Running nodes with `ros2 run`



1. Create the package first\_package
2. Create the source file: first\_package/src/main.cpp
3. Implement the node hello
4. Configure the build system to generate hello\_demo.
5. Build and run the executable.

```
#include <rclcpp/rclcpp.hpp>

int main(int argc, char **argv) {
    // Initialize ROS 2
    rclcpp::init(argc, argv);

    // Create a node
    auto node = std::make_shared<rclcpp::Node>("hello");

    // Log a message
    RCLCPP_INFO_STREAM(node->get_logger(), "Hello from ROS 2!");

    // Shutdown ROS 2
    rclcpp::shutdown();
}
```

## ☰ Understanding the Code

---

- `rclcpp :: init(argc, argv)`: Initializes ROS 2 runtime and middleware.
  - Sets up global resources, parses ROS arguments.
  - Must be called before creating any ROS entities.
- `std :: make_shared<rclcpp :: Node>("hello")`: Creates a shared pointer to a Node.
  - Node name must be unique within the ROS network.
  - Shared pointer enables safe memory management.
- `RCLCPP_INFO_STREAM(node->get_logger(), "message")`: Logs an info-level message. See [Logging Tutorial](#).
  - Uses the node's logger with automatic timestamping.
  - Other levels: `RCLCPP_DEBUG`, `RCLCPP_WARN`, `RCLCPP_ERROR`
- `rclcpp :: shutdown()`: Cleanly shuts down ROS 2.
  - Destroys all nodes and cleans up resources.
  - Automatically called on SIGINT (Ctrl+C).



This is a **one-shot node** - it runs once and exits. Later we'll see nodes that run continuously using `rclcpp :: spin()`.



Configure the build system by editing CMakeLists.txt

```
# Add this after find_package() statements
add_executable(hello_demo src/main.cpp)
ament_target_dependencies(hello_demo rclcpp)

# Add this before ament_package()
install(TARGETS
    hello_demo
    DESTINATION lib/${PROJECT_NAME}
)
```

## ☰ Build and Run

---

1. Build the package: `cd ~/ros702_ws && colcon build --packages-select first_package`
2. Source workspace: `source install/setup.<bash|zsh>`
3. Run the node: `ros2 run first_package hello_demo`

## ☰ Expected Output

---

| [1699123456.789] [hello]: Hello from ROS 2!

## ☰ ROS 2 Node Inheritance Pattern

---

- Inherit from `rclcpp :: Node` to access all node functionality.
- Constructor calls base class constructor with node name.
- Use `this->` to access node methods and properties.
- Encapsulate node-specific logic in class methods.



**Professional Development:** Real-world ROS 2 applications almost always use OOP.  
Learn this pattern early to build good habits.



Refactor the node using object-oriented design.

## ☰ Step 1: Create Header File

---

Create `first_package/include/first_package/hello_node.hpp`:

```
#pragma once

#include <rclcpp/rclcpp.hpp>

namespace first_package
{
    class HelloNode : public rclcpp::Node{
        public:
            explicit HelloNode(const std::string& node_name);

        private:
            void print_hello();
    };
}
```

## ☰ Step 2: Implement the Class

---

Create `first_package/src/hello_node.cpp`:

```
#include "first_package/hello_node.hpp"

namespace first_package
{
    HelloNode::HelloNode(const std::string& node_name) : Node(node_name){
        RCLCPP_INFO(this->get_logger(), "HelloNode constructor called");
        print_hello();
    }

    void HelloNode::print_hello(){
        RCLCPP_INFO_STREAM(this->get_logger(),
                           "Hello from " << this->get_name() << "!");
    }
} // namespace first_package
```

## ☰ Step 3: Update main.cpp

---

```
#include "first_package/hello_node.hpp"

int main(int argc, char **argv){
    rclcpp::init(argc, argv);
    auto node = std::make_shared<first_package::HelloNode>("hello_oop");
    rclcpp::shutdown();
}
```

### ☰ Step 4: Update CMakeLists.txt

---

Update the build configuration in  CMakeLists.txt:

### ☰ Step 5: Build and Test

---

1. Build:  `colcon build --packages-select first_package`
2. Run:  `ros2 run first_package hello_oop`

## ☰ Useful Colcon Build Options

---

- **colcon build --symlink-install**: Create symbolic links instead of copying files
  - Changes to Python files take effect immediately
  - Useful during development for faster iteration
- **colcon build --packages-select <pkg1> <pkg2>**: Build only specific packages
  - Faster builds when working on specific packages
  - Useful in large workspaces with many packages
- **colcon build --packages-up-to <pkg>**: Build package and its dependencies
  - Ensures all dependencies are built before target package
  - Good for dependency troubleshooting
- **colcon build --continue-on-error**: Build all packages, don't stop on first error
  - Useful for finding multiple build issues at once
  - Good for CI/CD pipelines



**Pro Tip:** Add alias `build='colcon build --symlink-install'` to your shell function for faster development cycles.

## ☰ Node Lifecycle Management

---

- Initialization: `rclcpp :: init()` sets up ROS 2 context
- Node Creation: Constructor runs, initializes publishers/subscribers
- Execution: `rclcpp :: spin()` keeps node alive (we will cover this next)
- Shutdown: `rclcpp :: shutdown()` cleans up resources

## ☰ Logging Best Practices

---

- Use appropriate levels: DEBUG → INFO → WARN → ERROR → FATAL
- Include context: Node name, function name, relevant data
- Be concise but informative: "Camera initialized with resolution 640x480"
- Use stream syntax for complex messages:  
`RCLCPP_INFO_STREAM(logger, "Value: " << value)`

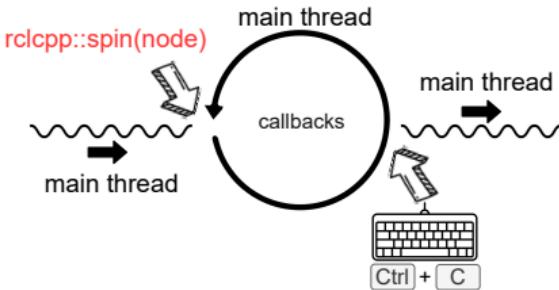
## ☰ Common Pitfalls to Avoid

---

- Forgetting to source workspace after builds
- Not updating  `CMakeLists.txt` when adding new files
- Using the same node name for multiple instances
- Not calling `rclcpp :: init()` before creating nodes

# **Node Spinning**

**Spinning** keeps a node alive and responsive by processing callbacks for incoming messages, service requests, and timer events.



## ■ Why Spin?

- Process incoming messages
- Execute timer callbacks
- Handle service requests
- Respond to actions
- Keep the node alive

## ■ Spinning Methods

- `rclcpp :: spin(node)` - Blocks forever
- `rclcpp :: spin_some(node)` - Process available work
- `rclcpp :: spin_until_future_complete()` - Conditional spinning
- Executor-based spinning (advanced)



Without spinning, your node will exit immediately after initialization, and callbacks won't be processed!

## Basic Node Spinning Pattern

```
int main(int argc, char **argv){  
    // Initialize ROS 2  
    rclcpp::init(argc, argv);  
    // Create node  
    auto node = std::make_shared<MyNode>("my_node");  
    // Spin the node - blocks until shutdown  
    rclcpp::spin(node);  
    // Clean up  
    rclcpp::shutdown();  
}
```

## What Happens During Spin?

1. **Event Loop:** Continuously checks for new messages, timer events, service calls
2. **Callback Execution:** Invokes appropriate callbacks when events occur
3. **Thread Management:** Handles single or multi-threaded callback execution
4. **Signal Handling:** Responds to shutdown signals (Ctrl+C)

## Spinning Alternatives

### ■ Spin Some: Process available callbacks without blocking

```
while (rclcpp::ok()) {  
    rclcpp::spin_some(node);  
    // Do other work here  
    std::this_thread::sleep_for(std::chrono::milliseconds(100));  
}
```

### ■ Spin Until Future Complete: Wait for specific condition

```
auto future = std::async(std::launch::async, some_async_operation);  
rclcpp::spin_until_future_complete(node, future);
```

### ■ Executor-based Spinning: More control over execution

```
rclcpp::executors::SingleThreadedExecutor executor;  
executor.add_node(node);  
executor.spin();
```



Use `rclcpp::spin()` for simple nodes. Use executors when you need fine-grained control over multiple nodes or threading.

# ROS 2 Messages

Messages are the fundamental data structures used for communication between nodes in ROS 2.

## ☰ Key Characteristics

---

- **Strongly Typed:** Each field has a specific data type
- **Language Agnostic:** Same message works in C++, Python, etc.
- **Serializable:** Automatically converted to/from bytes for transport
- **Versioned:** Changes require new message types
- **Efficient:** Optimized for real-time communication

## (Resources)

---

- [About ROS 2 Interfaces](#)
- [Common Interfaces Repository](#)

## ☰ Message Categories

---

- Standard Messages ( `std_msgs`): Basic data types
- Common Messages (,  `geometry_msgs`, `sensor_msgs`): Robotics-specific
- Package-Specific: Custom messages for specific packages



Think of messages as “contracts” between nodes: both publisher and subscriber must agree on the message structure.

### ☰ Standard Messages Package

---

The  **std\_msgs** package provides basic message types for common data.

#### ■ Primitive Types

-  **Bool** - Boolean value
-  **Byte** - Single byte
-  **Char** - Character
-  **String** - Text string
-  **Empty** - No data (events)

#### ■ Numeric Types

-  **Int8**,  **Int16**,  **Int32**,  **Int64**
-  **UInt8**,  **UInt16**,  **UInt32**,  **UInt64**
-  **Float32**,  **Float64**

#### ■ Array Types

-  **Int32MultiArray** - Multi-dimensional int32
-  **Float32MultiArray** - Multi-dimensional float32
-  **ByteMultiArray** - Multi-dimensional bytes

#### ■ Special Types

-  **Header** - Timestamp and frame ID
-  **ColorRGBA** - Color with alpha

### ☰ ROS 2 CLI for Messages

The  **ros2 interface** command provides powerful message introspection capabilities.

### ☰ List All Available Messages

- All interfaces:  **ros2 interface list**
- Only messages:  **ros2 interface list -m**
- From specific package:  **ros2 interface package std\_msgs**

Try the following:



-  **ros2 interface show std\_msgs/msg/String**
-  **ros2 interface show std\_msgs/msg/Header**

### ☰ Find Messages by Type

---

Search for all messages containing “String” in their name:  `ros2 interface list | grep String`

### ☰ Show Message Package

---

Find which package provides a message:  `ros2 interface package std_msgs/msg/String`

### ☰ Proto Format

---

Get message definition in proto format (useful for code generation):

 `ros2 interface proto std_msgs/msg/String`



**Development Workflow:** Always check the exact message structure with `ros2 interface show` before using it in your code to ensure field names and types are correct.

### ☰ Including Messages in C++

---

Each message type has a corresponding C++ header file:

```
// Include format: <package>/msg/<message_name>.hpp
#include <std_msgs/msg/string.hpp>
#include <std_msgs/msg/int32.hpp>
#include <std_msgs/msg/float64.hpp>
#include <std_msgs/msg/bool.hpp>
#include <std_msgs/msg/header.hpp>
#include <std_msgs/msg/empty.hpp>
```

### ☰ Message Type Names

---

- Full type: std\_msgs :: msg :: String
- With SharedPtr: std\_msgs :: msg :: String ::SharedPtr
- Const SharedPtr: std\_msgs :: msg :: String :: ConstSharedPtr

### Creating and Populating Messages

```
// String message
auto string_msg = std_msgs::msg::String();
string_msg.data = "Hello ROS 2";

// Numeric messages
auto int_msg = std_msgs::msg::Int32();
int_msg.data = 42;

// Header message (with timestamp)
auto header_msg = std_msgs::msg::Header();
header_msg.stamp = this->now(); // Current ROS time
header_msg.frame_id = "base_link";
```

 Messages are simple **structs**: access fields directly with the dot operator.

### ☰ Adding Message Dependencies

---

To use messages in your package, declare dependencies properly:

### ☰ In package.xml

---

```
<depend>std_msgs</depend>
<depend>geometry_msgs</depend>    <!-- If using geometry messages -->
<depend>sensor_msgs</depend>        <!-- If using sensor messages -->
```

### ☰ In CMakeLists.txt

---

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

# For your node
ament_target_dependencies(my_node
    rclcpp
    std_msgs
)
```

## ☰ Testing with Command Line

---

You can publish messages directly from the terminal for testing:

## ☰ Basic Publishing

---

- String:  `ros2 topic pub /chatter std_msgs/msg/String "data: 'Hello World'"`
- Integer:  `ros2 topic pub /number std_msgs/msg/Int32 "data: 42"`
- Float:  `ros2 topic pub /temperature std_msgs/msg/Float64 "data: 23.5"`
- Boolean:  `ros2 topic pub /enabled std_msgs/msg/Bool "data: true"`

## ☰ Publishing Options

---

- Once:  `ros2 topic pub --once /topic std_msgs/msg/String "data: 'test'"`
- Rate:  `ros2 topic pub --rate 10 /topic std_msgs/msg/String "data: 'test'"`
- Count:  `ros2 topic pub --times 5 /topic std_msgs/msg/String "data: 'test'"`

### ☰ Message Format Guidelines

---

When publishing from CLI, use YAML format:

Message Type	CLI Format	Example
String	"data: '<text>'"	"data: 'Hello'"
Numeric	"data: <number>"	"data: 3.14"
Boolean	"data: true/false"	"data: false"
Empty	""	""
Arrays	"data: [1, 2, 3]"	"data: [1.0, 2.0]"
Nested	Use indentation	See geometry_msgs

### ☰ Complex Message Example

---

For messages with multiple fields:

```
ros2 topic pub /header std_msgs/msg/Header{"stamp: {sec: 0, nanosec: 0}, frame_id: 'map"}
```

# Publishers

Publishers send messages to topics, enabling one-to-many communication between nodes.

## ☰ Publisher Workflow

---

1. **Create Publisher:** Specify message type, topic name, and QoS
2. **Create Message:** Instantiate and populate message object
3. **Publish:** Send message to all subscribers
4. **Repeat:** Often done periodically using timers

## ☰ Common Use Cases

---

- Sensor data streaming (cameras, lidar, IMU)
- Robot state broadcasting (position, velocity, battery)
- Control commands (velocity commands, joint positions)
- Status updates and heartbeats

## ▣ Resources

---

- [Writing a Simple Publisher and Subscriber \(C++\)](#)
- [About Topics](#)

 Create a node with a publisher component that publishes string messages.

Package	 <b><i>publisher_demo</i></b>
Header	 <b><i>include/publisher_demo/publisher_demo.hpp</i></b>
Source	 <b><i>src/publisher_demo.cpp</i></b>
Node	 <b><i>publisher_demo</i></b>
Executable	 <b><i>publisher_demo</i></b>

```
 ros2 pkg create publisher_demo --build-type ament_cmake --dependencies rclcpp std_msgs
```



Don't forget to update  ***package.xml*** with proper metadata and  ***CMakeLists.txt*** with build instructions!

## ☰ Publisher Node Header

---

Create `include/publisher_demo/publisher_demo.hpp`:

```
#pragma once

#include <rclcpp/rclcpp.hpp>
#include <std_msgs/msg/string.hpp>
#include <chrono>
#include <memory>

class PublisherDemo : public rclcpp::Node{
public:
    explicit PublisherDemo(const std::string& node_name);

private:
    void timer_callback();

    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
    rclcpp::TimerBase::SharedPtr timer_;
    size_t count_;
};
```

### ☰ Publisher Implementation

---

Create `src/publisher_demo.cpp`:

```
#include "publisher_demo/publisher_demo.hpp"

PublisherDemo::PublisherDemo(const std::string& node_name)
: Node(node_name), count_{0} {
    // Create publisher with message type, topic name, and QoS depth
    publisher_ = this->create_publisher<std_msgs::msg::String>("leia", 10);

    // Create timer for periodic publishing (2Hz = 500ms)
    timer_ = this->create_wall_timer(
        std::chrono::milliseconds(500),
        std::bind(&PublisherDemo::timer_callback, this));

    RCLCPP_INFO(this->get_logger(), "Publisher initialized");
}
```

### ☰ Key Components

---

- `create_publisher<MessageType>()`: Creates publisher object
- Topic name: "leia": Where messages will be published
- QoS depth: 10 - Message queue size
- Timer: Triggers callback every 500ms (2Hz)

## ☰ Understanding QoS (Quality of Service)

When you pass an integer to `create_publisher()`, it sets the queue depth with default QoS settings:

```
// Simple integer parameter
publisher_ = create_publisher<std_msgs::msg::String>("topic", 10);

// Equivalent explicit QoS
rclcpp::QoS qos(10); // Queue depth of 10
qos.reliability(rclcpp::ReliabilityPolicy::Reliable);
qos.durability(rclcpp::DurabilityPolicy::Volatile);
qos.history(rclcpp::HistoryPolicy::KeepLast);
publisher_ = create_publisher<std_msgs::msg::String>("topic", qos);
```

## ☰ Queue Behavior

Scenario	Queue Behavior
Subscribers keep up	Messages delivered immediately, queue stays empty
Slow/disconnected subscriber	Messages queued up to depth limit
Queue full	Oldest messages dropped (KeepLast policy)

### ☰ Timer Callback Implementation

---

```
void PublisherDemo::timer_callback(){
    // Create message
    auto message = std_msgs::msg::String();

    // Populate message data
    message.data = "Help me Obi-Wan Kenobi, you're my only hope #" +
                   std::to_string(count_++);

    // Log what we're publishing (optional)
    RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());

    // Publish the message
    publisher_->publish(message);
}
```

### ☰ Main Function

---

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

### ☰ Build and Run

---

-  `cd ~/ros702_ws colcon build --packages-select publisher_demo`
-  `source install/setup.<bash|zsh>`
-  `ros2 run publisher_demo publisher_demo`

# Subscribers

Subscribers receive messages from topics, enabling nodes to react to data published by other nodes.

## Subscriber Workflow

---

1. **Create Subscriber:** Specify message type, topic, callback, and QoS
2. **Define Callback:** Function to process received messages
3. **Spin Node:** Keep node alive to receive messages
4. **Process:** Handle messages as they arrive asynchronously

## Common Use Cases

---

- Processing sensor data for navigation
- Reacting to control commands
- Monitoring system status
- Data logging and visualization
- Triggering actions based on events

### ToDo

---

Create a subscriber node that receives messages from the `teleia` topic.

Package	 <b>subscriber_demo</b>
Header	 <code>include/subscriber_demo/subscriber_demo.hpp</code>
Source	 <code>src/subscriber_demo.cpp</code>
Node	 <b>subscriber_demo</b>
Executable	 <b>subscriber_demo</b>

```
 ros2 pkg create subscriber_demo --build-type ament_cmake --dependencies rclcpp std_msgs
```

### ☰ Subscriber Node Header

---

Create  `include/subscriber_demo/subscriber_demo.hpp`:

```
#pragma once

#include <rclcpp/rclcpp.hpp>
#include <std_msgs/msg/string.hpp>
#include <memory>

class SubscriberDemo : public rclcpp::Node{
public:
    explicit SubscriberDemo(const std::string& node_name);

private:
    void topic_callback(const std_msgs::msg::String::SharedPtr msg);

    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
};
```

### ☰ Subscriber Implementation

---

Create `src/subscriber_demo.cpp`:

```
#include "subscriber_demo/subscriber_demo.hpp"

SubscriberDemo::SubscriberDemo(const std::string& node_name)
: Node(node_name){
    subscription_ = this->create_subscription<std_msgs::msg::String>(
        "leia", // Topic name
        10, // QoS queue depth
        std::bind(&SubscriberDemo::topic_callback, this, std::placeholders::_1));
    RCLCPP_INFO(this->get_logger(), "Subscriber initialized, listening to 'leia'");
}
```

### ☰ Key Components

---

- `create_subscription<MessageType>()`: Creates subscriber
- Topic: "leia": Must match publisher's topic
- QoS: 10 - Subscriber's message queue size
- Callback: `std :: bind()` - Links callback method

## ☰ Callback Implementation

```
void SubscriberDemo :: topic_callback(const std_msgs::msg::String::SharedPtr msg){  
    RCLCPP_INFO(this->get_logger(), "Received: '%s'", msg->data.c_str());  
  
    // Process the message as needed:  
    // - Store data for later use  
    // - Trigger other actions  
    // - Update internal state  
    // - Publish transformed data  
}
```

## ☰ Main Function

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



The callback is executed asynchronously whenever a message arrives. Make sure callback execution is faster than message arrival rate to avoid queue overflow.

### ☰ Build and Test Publisher-Subscriber System

---

Update **CMakeLists.txt** for subscriber (same pattern as publisher).

### ☰ Run the Complete System

---

1. Terminal 1 - Start the publisher:

- **ros2 run publisher\_demo publisher\_demo**

2. Terminal 2 - Start the subscriber:

- **ros2 run subscriber\_demo subscriber\_demo**

3. Terminal 3 - Inspect the system:

- **ros2 topic list**
- **ros2 topic info /leia**
- **ros2 topic echo /leia**
- **ros2 topic hz /leia**



Always verify topic names match between publishers and subscribers. Use **ros2 topic list** to debug connection issues.

# Communication Scenarios

Understanding how messages flow between publishers and subscribers under different conditions.

## ☰ Scenario #1: No Subscriber

---

- Publisher rate: 2 Hz (1 message every 0.5s)
- **No subscriber present**
- QoS depth: 10 (publisher)

Time	Publisher Action	DDS Behavior	Result
t=0.0s	Publishes msg1	No matching DataReader	Message discarded
t=0.5s	Publishes msg2	No matching DataReader	Message discarded
t=1.0s	Publishes msg3	No matching DataReader	Message discarded

## ☰ Key Points

---

- DDS is smart - doesn't waste resources on undeliverable messages
- Publisher continues running normally
- No performance impact from missing subscribers
- Messages are lost forever (unless using durability QoS)

## ☰ Scenario #2: Fast Subscriber

---

- Publisher rate: 2 Hz (500ms period)
- Subscriber callback: **Fast - 100ms**
- QoS depth: 10 (both sides)

Time	Pub Action	Sub Queue	Callback	Notes
t=0.0s	Publish msg1	[msg1] → []	Starts immediately	Quick processing
t=0.1s	—	[]	Completes	Ready for next
t=0.5s	Publish msg2	[msg2] → []	Starts immediately	No queue buildup
t=0.6s	—	[]	Completes	System healthy

## ☰ Analysis

---

- Ideal scenario - subscriber keeps pace with publisher
- Queue never builds up (always 0 or 1 message)
- Low latency between publish and process
- System remains responsive

## ☰ Scenario #3: Slow Subscriber

---

- Publisher rate: 2 Hz (500ms period)
- Subscriber callback: **Slow - 1.7s**
- QoS depth: 3 (both sides)

Time	Pub	Sub Queue	Callback	Result
t=0.0s	msg1	[msg1]	Processing msg1	Queue: 0
t=0.5s	msg2	[msg2]	Still on msg1	Queue: 1
t=1.0s	msg3	[msg2, msg3]	Still on msg1	Queue: 2
t=1.5s	msg4	[msg2, msg3, msg4]	Still on msg1	Queue full!
t=1.7s	—	[msg3, msg4]	Done, starts msg2	—
t=2.0s	msg5	[msg3, msg4, msg5]	Processing msg2	msg3 dropped!

## ☰ Consequences

---

- Message loss when queue overflows
- Increasing latency between publish and process
- Subscriber always behind real-time
- May need to increase queue size or optimize callback

## ☰ Useful Commands

---

- Check if messages are being published:

-  `ros2 topic echo /topic_name`
  -  `ros2 topic hz /topic_name`

- Analyze publisher-subscriber connections:

-  `ros2 topic info /topic_name -v`
  -  `ros2 node info /node_name`

- Monitor system performance:

-  `ros2 doctor --report`
  - Use RCLCPP\_DEBUG logging in callbacks

## ☰ Performance Tips

---

- Keep callbacks fast - offload heavy processing
- Use appropriate queue sizes based on data rate
- Consider async processing for CPU-intensive tasks
- Profile your code to find bottlenecks

# Next Class

---

- Lecture 10: ROS (Part 2).