ENPM702

Introductory Robot Programming

L11: Robot Operating System (ROS) - Part III

v1.0

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:

- **Executors** - Covers single/multi-threaded executors and callback groups.
- **Name Remapping** - Includes namespaces, node/topic/parameter remapping via CLI and launch files.
- **Lifecycle Nodes** - Understand state machine transitions, configure/activate/deactivate callbacks, and controlled startup/shutdown.
- **Quality of Service (QoS)** - Configure reliability, durability, and history policies for different communication requirements.

Scaling Beyond Single Tasks

The proportional controller demo (previous lecture) 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.

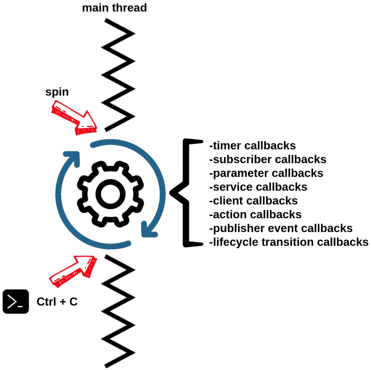
Build the package:

```
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.

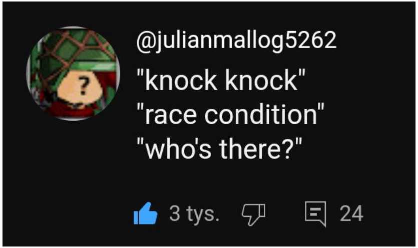
- **Thread pool**: A multi-threaded executor creates a pool of threads (you can often specify how many). Each thread can independently process callbacks from nodes added to the executor.
- **Callback scheduling**: When events occur, like a message arriving on a topic, a timer firing, or a service request, 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.

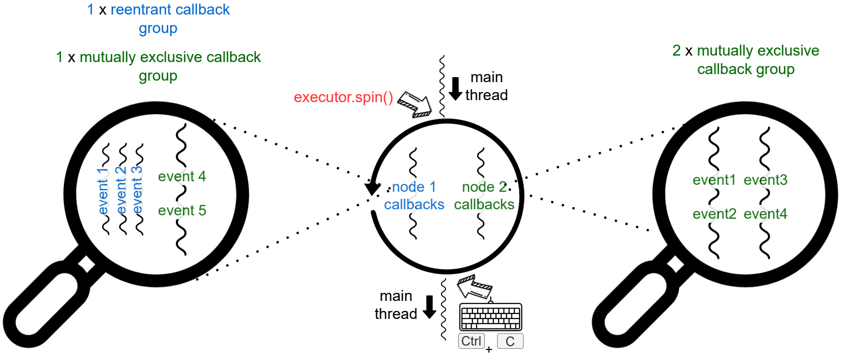
Use case: Independent tasks that don't interfere with each other, maximizing concurrency.

> 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.

Name Remapping

Name remapping in ROS2 is a feature that allows you to change the names of ROS entities (topics, services, parameters, and node names) at runtime without modifying the source code. This provides flexibility in how nodes communicate and helps avoid naming conflicts in complex systems.

1. `colcon build --symlink-install --packages-select remapping_demo`
2. Source the workspace.

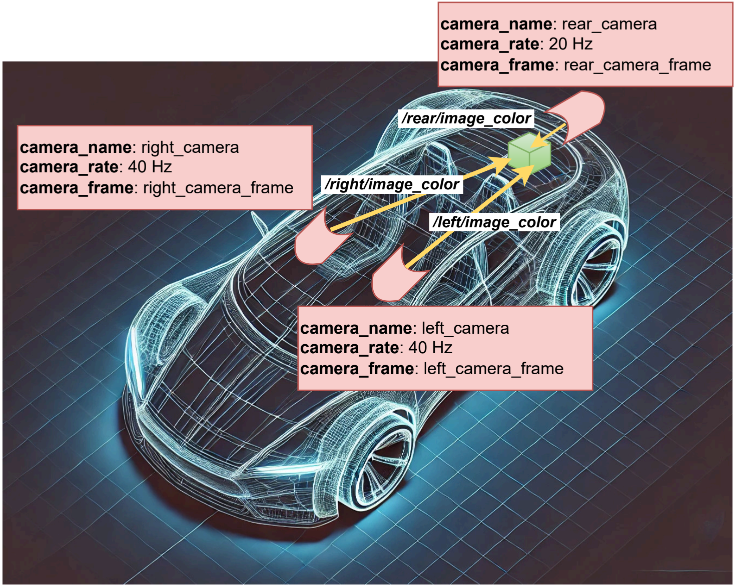
Resources

- ROS 2 Documentation: Node Name Arguments
- ROS 2 Design: Topic and Service Names
- ROS 2 Documentation: Using Launch Files
- ROS 2 Documentation: Node Name Remapping
- Robotics Backend: ROS 2 YAML Parameter Files

Remapping Types

- **Node names:** Run the same node multiple times with a different name.
- **Topic names:** Redirect publishers and subscribers to use different topic names.
- **Service names:** Change the names of services that nodes provide or use.
- **Parameter names:** Change the names of parameters a node looks for.

Name Remapping



Namespaces

Namespaces provide a hierarchical organization system that helps manage complexity in robotic applications. They function as **prefixes** that group related nodes, topics, services, and parameters together under a common path.

- When you apply a namespace to a node, all of its associated ROS entities (topics, services, parameters) automatically inherit that namespace prefix.
- You can apply namespaces through launch files, command-line arguments, or programmatically within nodes.

Advantages

- Prevents naming conflicts when running multiple instances of the same node
- Creates logical groupings of related components
- Simplifies visualization and debugging of complex systems
- Enables clear separation between subsystems

Demonstration

```
ros2 run remapping_demo camera_demo --ros-args -r __ns:=/rear_camera
```

```
rqt_graph
```

- The node name becomes n **/rear_camera/camera_demo**
- The topic becomes t **/rear_camera/camera/image_color** (**if defined without a leading slash**).
- Parameters are stored as p **/rear_camera/camera_demo.<parameter_name>**
  - The parameters are still technically **namespaced** but in a different way: they are organized under the fully-qualified node name, not directly under the namespace prefix.
  - `ros2 param list /rear_camera/camera_demo`
  - `ros2 param get /rear_camera/camera_demo camera_name`

Start the same node two more times with different namespaces.

- `ros2 run remapping_demo camera_demo --ros-args -r __ns:=/right_camera`
- `ros2 run remapping_demo camera_demo --ros-args -r __ns:=/left_camera`

Example: Launch File

```python
right_camera_node = Node(
        package='remapping_demo',
        executable='camera_demo',
        namespace='right_camera',
        output='screen',
        emulate_tty=True
)

left_camera_node = Node(
    package='remapping_demo',
    executable='camera_demo',
    namespace='left_camera',
    output='screen',
    emulate_tty=True
)

rear_camera_node = Node(
    package='remapping_demo',
    executable='camera_demo',
    namespace='rear_camera',
    output='screen',
    emulate_tty=True
)
```

Demonstration

```
ros2 launch remapping_demo demo1.launch.py
```

```
ros2 node list
```

```
rqt_graph
```

Node Remapping

Node remapping allows you to change the name of a node at runtime without modifying the code. This is crucial when running multiple instances of the same node type in a system.

Each node in a ROS 2 system must have a unique name.

Demonstration

Node remapping via command line:

```
ros2 run remapping_demo camera_demo --ros-args -r __node:=rear_camera_demo
```

- The **__node** argument is a special remapping target for the node name.
- The node that would normally be named n **camera_demo** is now n **rear_camera_demo**
- All parameters will be associated with the new node name.
- ROS commands must refer to the remapped name: `ros2 node info /rear_camera_demo`

ToDo

Remap the two other nodes.

Example: Launch File

```python
right_camera_node = Node(
    package='remapping_demo',
    executable='camera_demo',
    name='right_camera',
    output='screen',
    emulate_tty=True
)

left_camera_node = Node(
    package='remapping_demo',
    executable='camera_demo',
    name='left_camera',
    output='screen',
    emulate_tty=True
)

rear_camera_node = Node(
    package='remapping_demo',
    executable='camera_demo',
    name='rear_camera',
    output='screen',
    emulate_tty=True
)
```

Demonstration

```
ros2 launch remapping_demo demo2.launch.py
```

```
ros2 node list
```

```
rqt_graph
```

Topic Remapping

Topic remapping allows you to change the names of topics that nodes publish or subscribe to without modifying the node code.

When to Use Topic Remapping?

- Creating logical topic hierarchies, e.g., t **sensors/rear_camera/image_color**
- Connecting nodes with incompatible topic name expectations.
- Creating topic names that better describe the data they carry.

Demonstration

```
ros2 run remapping_demo camera_demo --ros-args -r __node:=rear_camera_demo \
-r /camera/image_color:=/rear_camera/image_color
```

- The format is **-r original_topic:=new_topic**
- The topic that would normally be t **/camera/image_color** is now t **/rear_camera/image_color**
- Subscribers must use the remapped topic name to receive the data.
- Check active topics with: `ros2 topic list`

Remap the two other nodes and topics.

Example: Launch File

```
right_camera_node = Node(
    package="remapping_demo",
    executable="camera_demo",
    name="right_camera",
    remappings=[
        ("/camera/image_color", "/right/image_color")
    ],
    output="screen",
    emulate_tty=True,
)
```

Demonstration

- `ros2 launch remapping_demo demo3.launch.py`
- `ros2 node list`
- `ros2 topic list`
- `rqt_graph`

Parameter Remapping

Parameter remapping is the action of changing the value at runtime. This was covered in the previous lecture.

Combining Remapping Approaches

You can combine all remapping types in a single command.

Demonstration

```
ros2 run remapping_demo camera_demo --ros-args \
    -r __ns:=/vehicle \
    -r __node:=right_demo \
    -r camera/image_color:=right/image_color \
    -p camera_name:=right_camera \
    -p camera_rate:=40 \
    -p camera_frame:=right_camera_frame
```

Example

```
right_camera_node = Node(
    package="remapping_demo",
    executable="camera_demo",
    namespace='vehicle',
    name="right_camera",
    remappings=[
        ("camera/image_color", "right/image_color"),
    ],
    parameters=[{
    'camera_name': 'right_camera',
    'camera_rate': right_camera_rate,
    'camera_frame': right_camera_frame
    }],
    output="screen",
    emulate_tty=True,
)
```

Demonstration

- `ros2 launch remapping_demo demo5.launch.py`
- `ros2 node list`
- `ros2 topic list`
- `rqt_graph`

Lifecycle Nodes

ROS2 lifecycle nodes provide managed state transitions for controlled startup, shutdown, and runtime management of nodes. They follow a standardized state machine pattern for:

- Controlled initialization sequences.
- Graceful shutdown procedures.
- Runtime activation/deactivation.
- Error handling and recovery.

Key Benefits

- **Deterministic Startup**: Control node initialization order.
- **Resource Management**: Proper cleanup of resources.
- **Runtime Control**: Activate/deactivate without restarting.
- **System Coordination**: Synchronize multiple nodes.

Primary States

- **Unconfigured**: Initial state, minimal resources.
- **Inactive**: Configured but not processing.
- **Active**: Fully operational and processing.
- **Finalized**: Cleanly shut down.

State Transitions

- **Configure:** Unconfigured → Inactive
- **Activate:** Inactive → Active
- **Deactivate:** Active → Inactive
- **Cleanup:** Inactive → Unconfigured
- **Shutdown:** Any state → Finalized

Resources

- Managed Nodes

Create a "Sensor Publisher" Lifecycle Node

1. Implement a lifecycle node that publishes sensor data.
2. Override transition callbacks (configure, activate, deactivate, cleanup).
3. Test manual state transitions using ROS2 services.
4. Demonstrate controlled startup and shutdown.

Include Headers

```cpp
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_lifecycle/lifecycle_node.hpp"
#include "std_msgs/msg/string.hpp"
```

Class Declaration

```cpp
class SensorPublisher : public rclcpp_lifecycle::LifecycleNode
{
public:
  SensorPublisher() : LifecycleNode("sensor_publisher")
  {
    RCLCPP_INFO(get_logger(), "Sensor Publisher created");
  }
```

Configure Transition

```cpp
rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn
on_configure(const rclcpp_lifecycle::State& previous_state) override
{
  RCLCPP_INFO(get_logger(), "Configuring from: %s",
              previous_state.label().c_str());

  // Create publisher (but don't activate yet)
  publisher_ = create_publisher<std_msgs::msg::String>("sensor_data", 10);

  RCLCPP_INFO(get_logger(), "Configuration complete");
  return rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::
       CallbackReturn::SUCCESS;
}
```

Activate Transition

```
rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn
on_activate(const rclcpp_lifecycle::State& previous_state) override
{
  RCLCPP_INFO(get_logger(), "Activating from: %s",
              previous_state.label().c_str());

  // Activate publisher and start timer
  publisher_->on_activate();

  timer_ = create_wall_timer(std::chrono::seconds(1), [this]() {
    auto msg = std_msgs::msg::String();
    msg.data = "Sensor reading: " + std::to_string(counter_++);
    publisher_->publish(msg);
    RCLCPP_INFO(get_logger(), "Published: %s", msg.data.c_str());
  });

  return rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::
         CallbackReturn::SUCCESS;
}
```

Deactivate Transition

```cpp
rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn
on_deactivate(const rclcpp_lifecycle::State& previous_state) override
{
  RCLCPP_INFO(get_logger(), "Deactivating from: %s",
              previous_state.label().c_str());

  // Stop timer and deactivate publisher
  timer_.reset();
  publisher_->on_deactivate();

  return rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::
         CallbackReturn::SUCCESS;
}
```

Cleanup Transition

```
rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn
on_cleanup(const rclcpp_lifecycle::State& previous_state) override
{
  RCLCPP_INFO(get_logger(), "Cleaning up from: %s",
              previous_state.label().c_str());

  // Reset all resources
  timer_.reset();
  publisher_.reset();
  counter_ = 0;

  return rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::
         CallbackReturn::SUCCESS;
}
```

Private Members

```cpp
private:
  rclcpp_lifecycle::LifecyclePublisher<std_msgs::msg::String>::SharedPtr
    publisher_;
  rclcpp::TimerBase::SharedPtr timer_;
  int counter_ = 0;
};
```

Main Function

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

  auto node = std::make_shared<SensorPublisher>();
  rclcpp::spin(node->get_node_base_interface());

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

- `colcon build --packages-up-to lifecycle_demo`
- `source install/setup.bash`
- `ros2 run lifecycle_demo sensor_publisher`
- `ros2 lifecycle -h`

Manual State Transitions

```
ros2 lifecycle set /sensor_publisher configure
ros2 lifecycle set /sensor_publisher activate
ros2 topic echo /sensor_data
ros2 lifecycle set /sensor_publisher deactivate
ros2 lifecycle set /sensor_publisher cleanup
```

Quality of Service

ROS2 Quality of Service (QoS) policies define how data flows between publishers and subscribers, providing fine-grained control over communication reliability, performance, and resource usage.

- Network reliability and fault tolerance.
- Real-time performance guarantees.
- Resource management and optimization.
- Compatibility with existing systems.

Key QoS Policies

- **Reliability**: Best-effort vs reliable delivery.
- **Durability**: Transient vs volatile data storage.
- **History**: Keep last N messages vs all messages.
- **Lifespan**: How long messages remain valid.

Common Use Cases

- **Sensor Data**: High-frequency, best-effort streaming.
- **Commands**: Reliable delivery with acknowledgment.
- **Configuration**: Persistent, reliable parameters.
- **Diagnostics**: Periodic status with history.

Built-in QoS Profiles

- **Default**: Reliable, volatile, keep last 10
- **Sensor Data**: Best-effort, volatile, keep last 5
- **Services**: Reliable, volatile, keep last 10
- **Parameters**: Reliable, transient local, keep last 1000
- **System Default**: Platform-specific defaults

QoS Compatibility

Publishers and subscribers must have **compatible** QoS settings to communicate:

- **Reliability**: Reliable publisher can send to best-effort subscriber.
- **Durability**: Transient publisher can send to volatile subscriber.
- **History**: Flexible matching based on depth.

Resources

- QoS Settings

Create a “Sensor Monitor” with Different QoS

1. Create a publisher with sensor data QoS profile.
2. Implement subscribers with different QoS settings.
3. Compare reliable vs best-effort delivery.
4. Test history depth and durability settings.

include/qos_demo/sensor_monitor.hpp

```cpp
#pragma once

#include "rclcpp/rclcpp.hpp"
#include "sensor_msgs/msg/temperature.hpp"
#include "rclcpp/qos.hpp"

class SensorMonitor : public rclcpp::Node
{
public:
  SensorMonitor();

private:
  void timer_callback();
  void reliable_callback(const sensor_msgs::msg::Temperature::SharedPtr msg);
  void best_effort_callback(const sensor_msgs::msg::Temperature::SharedPtr msg);
};
```

Sensor Data Publisher

```cpp
SensorMonitor::SensorMonitor() : Node("sensor_monitor")
{
  // Publisher with sensor data QoS profile
  sensor_publisher_ = create_publisher<sensor_msgs::msg::Temperature>(
    "temperature",
    rclcpp::SensorDataQoS()  // Best-effort, volatile, keep last 5
  );

  // Timer for publishing
  timer_ = create_wall_timer(
    std::chrono::milliseconds(100),
    std::bind(&SensorMonitor::timer_callback, this)
  );

  RCLCPP_INFO(get_logger(), "Sensor monitor started with SensorDataQoS");
}
```

Reliable Subscriber

```cpp
// Reliable subscriber - will miss some messages from best-effort publisher
auto reliable_qos = rclcpp::QoS(10)
  .reliability(rclcpp::ReliabilityPolicy::Reliable)
  .durability(rclcpp::DurabilityPolicy::Volatile);

reliable_subscriber_ = create_subscription<sensor_msgs::msg::Temperature>(
  "temperature", reliable_qos,
  std::bind(&SensorMonitor::reliable_callback, this, std::placeholders::_1)
);
```

Best-Effort Subscriber

```cpp
// Best-effort subscriber - compatible with sensor data QoS
best_effort_subscriber_ = create_subscription<sensor_msgs::msg::Temperature>(
  "temperature", rclcpp::SensorDataQoS(),
  std::bind(&SensorMonitor::best_effort_callback, this, std::placeholders::_1)
);
```

Custom QoS Profile

```cpp
// Custom QoS with specific requirements
auto custom_qos = rclcpp::QoS(rclcpp::KeepLast(50))
  .reliability(rclcpp::ReliabilityPolicy::Reliable)
  .durability(rclcpp::DurabilityPolicy::TransientLocal)
  .deadline(std::chrono::milliseconds(200))
  .lifespan(std::chrono::seconds(10));

custom_subscriber_ = create_subscription<sensor_msgs::msg::Temperature>(
  "temperature", custom_qos,
  [this](const sensor_msgs::msg::Temperature::SharedPtr msg) {
    RCLCPP_INFO(get_logger(), "Custom QoS received: %.2f°C",
                msg->temperature);
  }
);
```

Temperature Publishing

```cpp
void SensorMonitor::timer_callback()
{
  auto msg = sensor_msgs::msg::Temperature();
  msg.header.stamp = now();
  msg.header.frame_id = "sensor_frame";

  // Simulate temperature readings with noise
  static double base_temp = 25.0;
  base_temp += (std::rand() % 100 - 50) * 0.01;  // ±0.5°C noise
  msg.temperature = base_temp;
  msg.variance = 0.1;

  sensor_publisher_->publish(msg);

  // Log every 10th message to avoid spam
  static int count = 0;
  if (++count % 10 == 0) {
    RCLCPP_INFO(get_logger(), "Published temperature: %.2f°C",
                msg.temperature);
  }
}
```

Subscriber Callbacks

```cpp
void SensorMonitor::reliable_callback(
  const sensor_msgs::msg::Temperature::SharedPtr msg){
  static int reliable_count = 0;
  reliable_count++;

  if (reliable_count % 20 == 0) {
    RCLCPP_WARN(get_logger(),
                "Reliable subscriber (incompatible): received %d messages",
                reliable_count);
  }
}

void SensorMonitor::best_effort_callback(
  const sensor_msgs::msg::Temperature::SharedPtr msg){
  static int best_effort_count = 0;
  best_effort_count++;

  if (best_effort_count % 20 == 0) {
    RCLCPP_INFO(get_logger(),
                "Best-effort subscriber: received %d messages",
                best_effort_count);
  }
}
```

- `colcon build --packages-select qos_demo`
- `source install/setup.bash`
- `ros2 run qos_demo sensor_monitor`
- `ros2 topic info /temperature --verbose`

QoS Inspection Commands

- `ros2 topic info /temperature --verbose`
- `ros2 node info /sensor_monitor`
- `ros2 topic echo /temperature --qos-reliability best_effort`
- `ros2 topic hz /temperature`

Common QoS Issues

- **No Connection:** Incompatible reliability policies
- **Missing Messages:** History depth too small
- **High Latency:** Reliable delivery with network issues
- **Memory Usage:** Large history depth with transient durability

Debugging Commands

```
# Check QoS compatibility
ros2 topic info /topic_name --verbose

# Monitor connection status
ros2 node info /node_name

# Test different QoS settings
ros2 topic echo /topic_name --qos-reliability reliable
ros2 topic echo /topic_name --qos-history keep_all
```

Thank you for the semester!!