ROS 2: Communication, Packages, Configurations

Course: Robotic Programming Environments

Michał Drwięga
michal.drwiega@pwr.edu.pl
www.mdrwiega.com/edu/rpe

Department of Cybernetics and Robotics
Wrocław Univeristy of Science and Technology

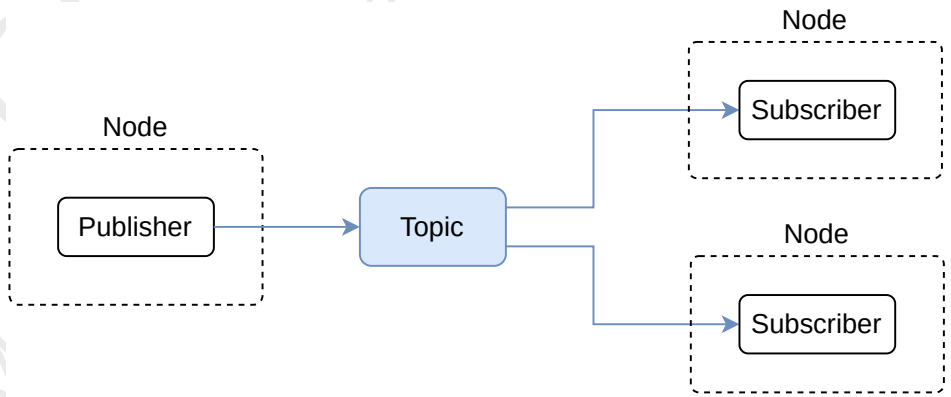
Wrocław University
of Science and Technology

Content

Topics (QoS)

Communication - topics

- Publish-subscribe model
- Message based
- Messages defined as *.msg* files
- Underlaying layer: DDS
- Communication frequency depends on the publisher



QoS - Quality of Service (in DDS)

QoS profiles - defines a set of policies[1]

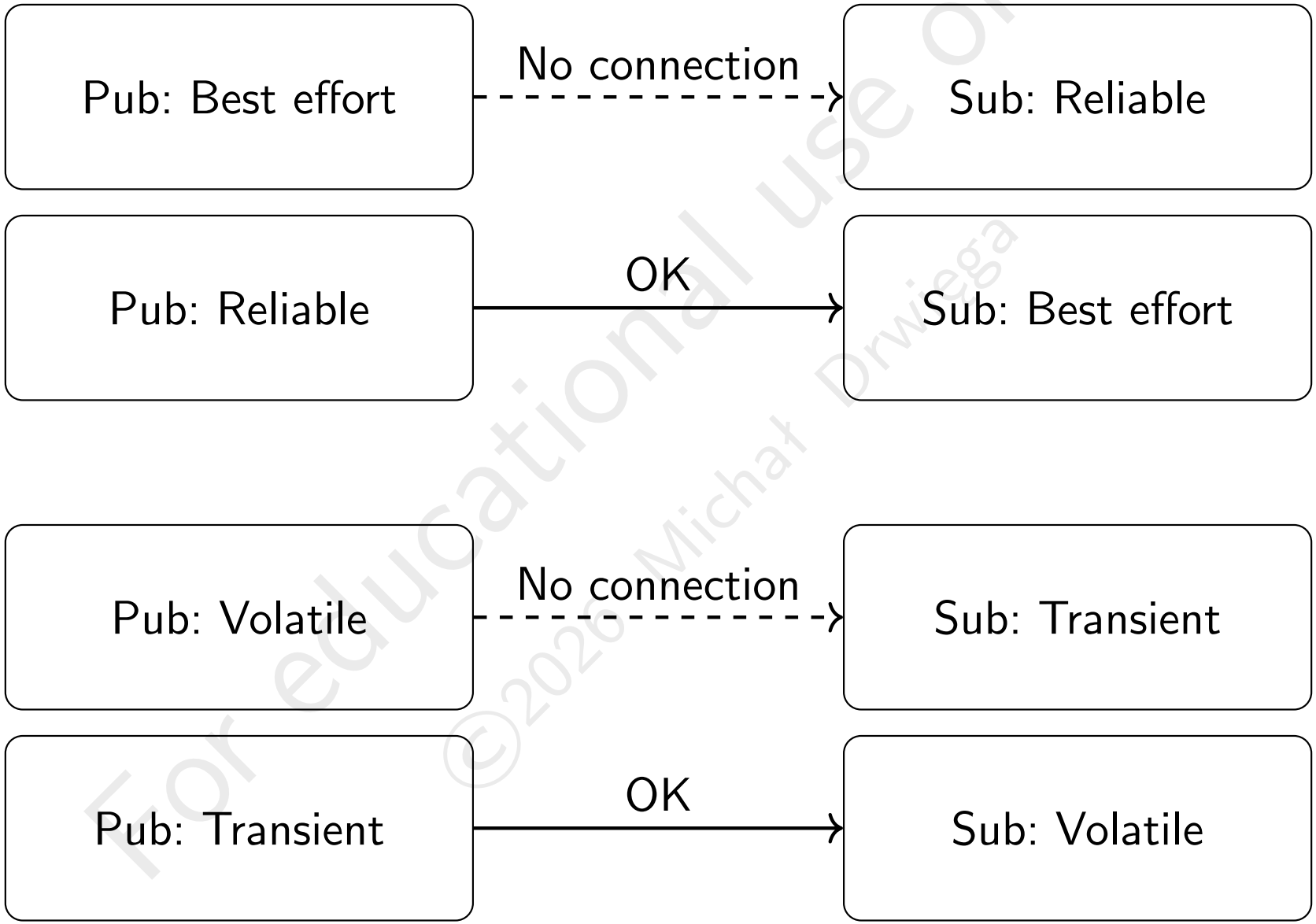
- History
  - Keep last - store only up to N messages
  - Keep all - store all messages
- Depth - size of the queue
- Reliability
  - Best effort - attempt to deliver messages (may be lost if network is not reliable enough)
  - Reliable - guarantee that messages are delivered (try multiple times)
- Durability
  - Transient local - persisting messages until a corresponding receiver is available
  - Volatile - no attempt is made to persist samples

[1] https://design.ros2.org/articles/qos.html

QoS intuition

- **Best effort** = like UDP (fast, may lose data)
- **Reliable** = like TCP (guaranteed delivery)
- **Transient local** = "remember last message"
- **Volatile** = "only live data matters"

QoS Compatibility Overview



QoS Compatibility Rules

- **Subscriber defines minimum requirements**
- **Publisher must meet or exceed them**

When to use which QoS?

- **Sensor data (e.g. LiDAR, camera)**
  - History: Keep last (small depth)
  - Reliability: Best effort
  - Durability: Volatile
  - *Reason: high frequency, data can be dropped*
- **Robot commands (e.g. velocity control)**
  - History: Keep last
  - Reliability: Reliable
  - Durability: Volatile
  - *Reason: must arrive, but no need to store old commands*
- **Map / static data**
  - History: Keep last
  - Reliability: Reliable
  - Durability: Transient local
  - *Reason: new subscribers need last map immediately*

QoS settings - code example

```python
import rclpy
from std_msgs.msg import String

rclpy.init()
node = rclpy.create_node('qos_example_node')

# Define QoS settings for the publisher
publisher_qos = rclpy.qos.QoSPublisherProfile(
    history=rclpy.qos.QoSHistoryPolicy.KEEP_LAST,
    depth=10,
    reliability=rclpy.qos.QoSReliabilityPolicy.RELIABLE,
    durability=rclpy.qos.QoSDurabilityPolicy.TRANSIENT_LOCAL
)

publisher = node.create_publisher(String, 'topic_name',
    qos_profile=publisher_qos)
```

2

[2] https://github.com/ros2/examples/blob/rolling/rclpy/topics/minimal_subscriber

ROS 2 Message Ordering: Single Publisher

Single Publisher → Subscriber: order guaranteed

Publisher: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$

Subscriber receives: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$

ROS 2 Message Ordering: Multiple Publishers

Multiple Publishers → Single Subscriber: order not guaranteed

Publisher A: $0 \rightarrow 1 \rightarrow 2$

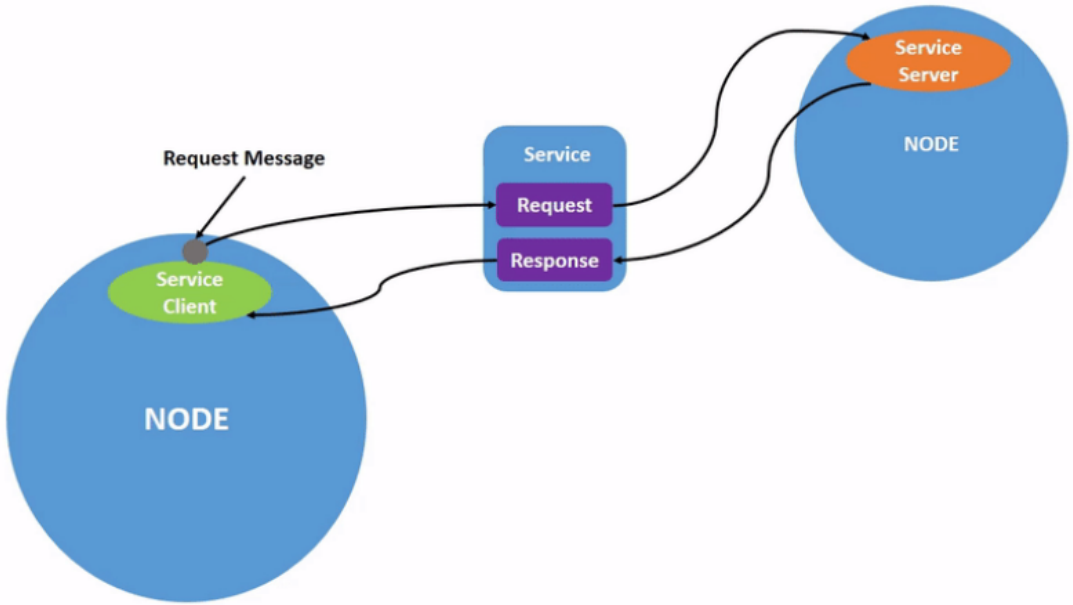
Publisher B: $a \rightarrow b \rightarrow c$

Subscriber may receive: $0 \rightarrow a \rightarrow 1 \rightarrow b \rightarrow 2 \rightarrow c$

Services

Communication - services

- Request-response model (one-to-one)
- Sevices defined as *.srv* files
- Underlaying layer: DDS
- Standard services: std_srvs package



3

[3] https://docs.ros.org

Example of the service definiton

AddTwoInts.srv

```
1     int64 a
2     int64 b
3     ---
4     int64 sum
```

Example of the service server

```
1  from example_interfaces.srv import AddTwoInts
2  import rclpy
3  from rclpy.node import Node
4
5  class ServiceNode(Node):
6      def __init__(self):
7          super().__init__('service')
8          self.srv = self.create_service(AddTwoInts, '
               add_two_ints', self.add_two_ints_callback)
9
10     def add_two_ints_callback(self, request, response):
11         response.sum = request.a + request.b
12         self.get_logger().info('Incoming request\na: %d b: %d
               ' % (request.a, request.b))
13
14         return response
```

Example of the service server

```
1  ...
2
3  def main(args=None):
4      rclpy.init(args=args)
5      service = ServiceNode()
6      rclpy.spin(service)
7      rclpy.shutdown()
8
9  if __name__ == '__main__':
10     main()
```

4

[4] https://github.com/ros2/examples/tree/rolling/rclpy/services/minimal_service

Example of the service client

```
1  class ClientAsyncNode(Node):
2      def __init__(self):
3          super().__init__('client_async')
4          self.cli = self.create_client(AddTwoInts, '
               add_two_ints')
5          while not self.cli.wait_for_service(timeout_sec=1.0):
6              self.get_logger().info('service not available..')
7          self.req = AddTwoInts.Request()
8
9      def send_request(self):
10         self.req.a = 41
11         self.req.b = 1
12         self.future = self.cli.call_async(self.req)
```

5

[5] https://github.com/ros2/examples/tree/rolling/rclpy/services/minimal_service

Example of the service client

```
...
def main(args=None):
    rclpy.init(args=args)
    client = ClientAsyncNode()
    client.send_request()

    while rclpy.ok():
        rclpy.spin_once(client)
        if client.future.done():
            response = client.future.result()
            client.get_logger().info(
                f'Result: {response.sum}')
            break

    client.destroy_node()
    rclpy.shutdown()
```

Communication mechanisms - comparison

Topics

- Purpose: continuous data streams, e.g. sensor data
- Many to many connection
- Data might be published and subscribed independently (at any time)

Services

- Purpose: remote procedure calls that can be executed quickly e.g. getting the current battery state
- One to one connection

Packages & Workspaces

ROS 2 Packages

A ROS package is the **primary unit for organizing software in ROS**.

Key points:

- Packages are the building blocks of ROS software
- Can depend on other packages via `package.xml`
- Enables modularity, reuse, and easy distribution

Structure:

- **package.xml** – metadata about the package (name, version, dependencies)
- **CMakeLists.txt** – build instructions.
- **setup.py** – installation and dependency instructions for Python nodes
- Nodes – executables implementing functionality
- Launch files – scripts to start multiple nodes
- Configuration and resource files – parameters, URDFs, meshes, etc.

Example ROS 2 package.xml

```xml
<package format="3">
  <name>my_package</name>
  <version>0.1.0</version>
  <description>A package for controlling my robot</description>
  <maintainer email="user@example.com">User Name</maintainer>
  <license>BSD</license>

  <!-- Dependencies -->
  <buildtool_depend>ament_cmake</buildtool_depend>
  <depend>rclcpp</depend>
  <depend>std_msgs</depend>
  <exec_depend>launch</exec_depend>

  <!-- Export -->
  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

Listing: ROS 2 Package XML

Example ROS 2 Package Structure

```
my_robot_controller/
    CMakeLists.txt
    package.xml
    launch/
        robot_launch.py
    config/
        robot_params.yaml
    src/
        my_robot_controller/
            __init__.py
            controller_node.py
        test/
            test_controller.py
    resource/
         my_robot_controller
```

Example ROS 2 CMakeLists.txt

```
cmake_minimum_required(VERSION 3.5)
project(my_package)

# Find dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)

# Add executable
add_executable(talker src/talker.cpp)
ament_target_dependencies(talker rclcpp std_msgs)

# Install targets
install(TARGETS
  talker
  DESTINATION lib/${PROJECT_NAME}
)

# Export package
ament_package()
```

Example ROS 2 setup.py

```python
from setuptools import setup
package_name = 'my_package'

setup(
    name=package_name,
    version='0.1.0',
    packages=[package_name],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='User name',
    maintainer_email='user@example.com',
    description='A package for controlling my robot',
    license='BSD',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'talker = my_package.talker:main',
            'listener = my_package.listener:main',
        ],
    },
)
```

C++ vs Python in ROS Packages

C++ (rclcpp) in Packages:

- Nodes implemented as compiled executables
- Build system uses `CMakeLists.txt` in the package
- Ideal for performance-critical tasks within a package (e.g., SLAM, control)
- Can create libraries reusable across multiple nodes/packages

Python (rclpy) in Packages:

- Nodes implemented as scripts inside the package
- Package provides `setup.py` for installation and dependencies
- Ideal for rapid prototyping, utilities, and simple nodes
- Easy to integrate with existing C++ nodes in the same package or workspace

ROS 2 Workspace

A ROS 2 workspace is a **directory structure** where you develop, build, and organize ROS 2 packages.

Typical structure:

`ros2_ws/`

- `src/` – contains all ROS 2 packages you are developing
- `install/` – generated after building; contains built packages and environment setup scripts
- `build/` – intermediate build files
- `log/` – build and runtime logs

ROS 2 Workspace - Workflow

1. Create workspace: `mkdir -p ~/ros2_ws/src`
2. Add packages to `src/`
3. Build workspace: `colcon build`
4. Source setup: `source install/setup.bash`
5. Run nodes and launch files

ROS 2 Workspace - Notes

- Multiple workspaces can be layered by sourcing multiple `setup.bash` files
- Standard ROS 2 development uses `colcon` instead of `catkin_make`

Build Tool for ROS 2 Packages – `colcon build`

Key features:

- Manages dependencies between packages.
- Supports parallel building to reduce build times.

Common commands:

- `colcon build`
  - Builds all packages in the workspace.
- `colcon build --packages-select <pkg_name>`
  - Builds only the specified package(s).
- `colcon build --symlink-install`
  - Creates symbolic links instead of copying files to the install directory.

Tips:

- Source the workspace setup file (`source install/setup.bash`) after a successful build.
- Use `colcon build --cmake-args -DCMAKE_BUILD_TYPE=Release` for optimized builds.

Nodes Parametrization

Nodes parametrization

- Supported types:
  - bool, int64, float64, string
  - byte[], bool[], int64[], float64[], string[]
- Defined in YAML files
- Parameters are node specific (different than in ROS1 where it was a parameter server)

Useful commands (CLI)

- `ros2 param list`
- `ros2 param get <node_name> <param_name>`
- `ros2 param set <node_name> <param_name> <value>`

6 7

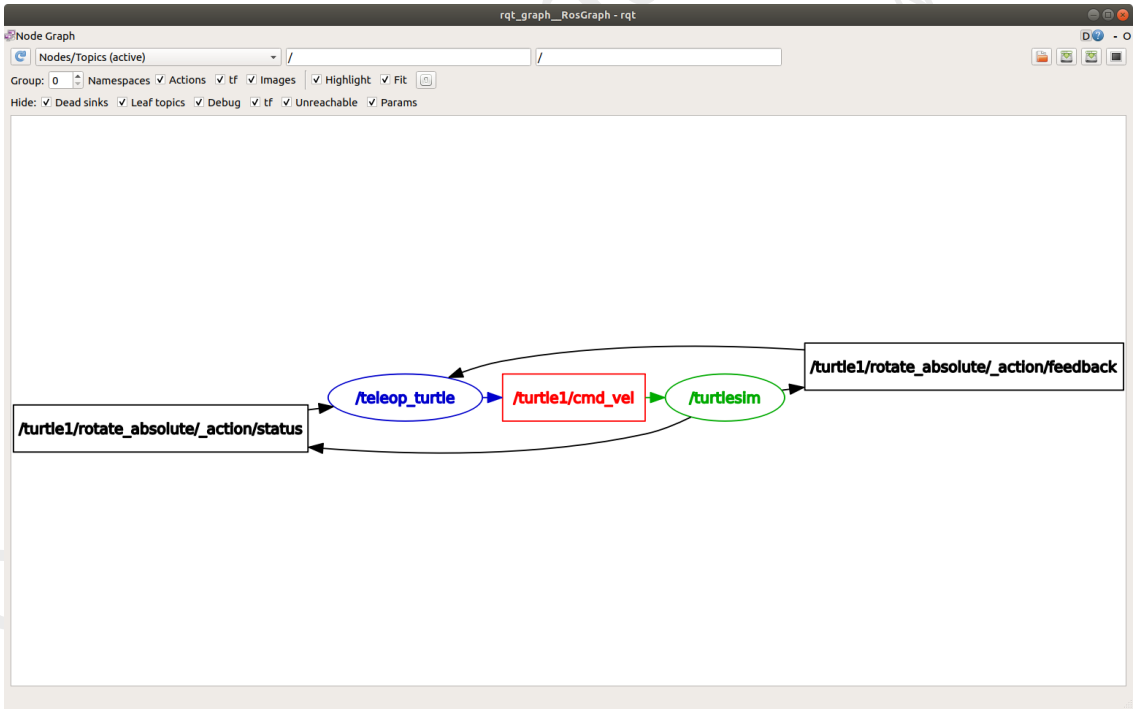
[6] https://docs.ros.org/en/foxy/Concepts/About-ROS-2-Parameters.html
[7] https://design.ros2.org/articles/ros_parameters.html

Parameters - example

```
laserscan_kinect:
  ros__parameters:
    # Frame id for the output laserscan.
    output_frame_id: camera_depth_frame
    # Minimum sensor range (m).
    range_min: 0.4
    # Maximum sensor range (m).
    range_max: 5.0
    # Height of the scan (part of the depth image) in px.
    scan_height: 400
```

ROS tools - rqt_graph

- rqt_graph can visualize the ROS graph of the application.
- The ROS graph contains all running nodes and the communication between them. Also, the topics and nodes are displayed inside their namespace.



[8]

[8] https://roboticsbackend.com/rqt-graph-visualize-and-debug-your-ros-graph/

Summary

1. Topics (QoS)
2. Services
3. Packages & Workspaces
4. Nodes Parametrization

The end

Thank you for your attention!