A short guide to ROS 2 Humble Hawksbill

Matti Kortelainen, PhD

School of Computing, University of Eastern Finland, Kuopio, Finland

March 22, 2023

PREFACE AND ACKNOWLEDGEMENTS

This document is based on a collection of notes I wrote while setting up and learning to use ROS 2 Humble Hawksbill. To emulate the setup and challenges similar to those that students might encounter during their studies in robotics, I purchased a Raspberry Pi 4 Model B computer with 4 GB of RAM. For OS, I chose the latest long-term support version of Ubuntu (v.22.04.1) because it is a Linux distribution recommended for ROS 2 and because it became relatively recently available for Raspberry Pi, too.

I expect the reader to be familiar with basic concepts of object-oriented programming, and to possess a rudimentary knowledge of Linux operating systems. That being said, I aim to give (verbatim) terminal commands related to installation, configuration and use of ROS 2. The reader should also be aware that, as ROS 2 and its associated tools undergo further development, the validity of information presented in this document cannot be completely guaranteed beyond March 2023.

This document was created as a part of the Experts in Medical Computing project, funded by the European Social Fund (ESF; S21823) and the Centre for Economic Development, Transport, and the Environment South Savo. I express thanks to Professor Roman Bednarik and Professor Pasi Karjalainen for their supervision. I also express thanks to Dr. Paavo Vartiainen for his guidance in programming the Arduino board and helpful comments on the draft of this document.

Kuopio, March 20, 2023

Matti Kortelainen

TABLE OF CONTENTS

1 Background

Essentially, ROS 2 (Robot Operating System 2) [1] is an open-source software framework based on Data Distribution Service (DDS) which provides standardized tools for

- organizing the code of your application into modules with clearly-defined purposes,
- distributing the (concurrent) execution of this code over multiple available executors, and
- communication between the above-mentioned modules during the (concurrent) execution.

Several tools provided by ROS 2 are based on utilities already available in their base programming languages (such as usage of lock/mutex); however, in ROS 2, these utilities have been used to build libraries which have been found useful by numerous robotics practitioners in both industry and academia. That is, the use of ROS 2 allows you to skip several steps in developing your application as you would probably end up creating quite similar tools to enable concurrent computing in a distributed system. In addition, the use of ROS 2 improves compatibility of applications by different practitioners due to uniform interfaces provided by ROS 2.

ROS 2 provides two client libraries to wrap your code into ROS 2 application: the `rclpy` library for Python, and the `rclcpp` library for C++ [2]. In addition, there are community-maintained client libraries for, e.g., Rust and C#. These client libraries provide programming language-specific interfaces for users to access common core functionalities of ROS 2, provided by the ROS Client Library (`rcl`) written in C [3].

Applications built on ROS 2 follow a straightforward architecture: the application's code is wrapped in `Node` modules, which are spun by executors when the application is run via **executable** files. These executables, in turn, are part of a certain **package**.

1.1 NODES

Both `rclpy` and `rclcpp` libraries provide a class called `Node`. This class provides a wrapper for your code to access ROS 2 functionalities. Basically, you only need to arrange your code into classes which are subclasses/child classes of `Node` to access its (non-private) members. Dividing your code into multiple nodes enables parallel execution of multiple segments of your application, and methods inherited from class `Node` enable inter-nodal (and intra-nodal) communication.

1.2 TOPICS

Topics are the most straightforward means of communication between nodes. A topic is a bus that transmits messages from **publisher** objects to **subscriber** objects which are members of nodes. A topic can have any number of publishers and any number of subscribers (Figure 1.1).

Both `rclpy` and `rclcpp` libraries provide methods to create publisher objects with a `publish()` method which sends a message to the topic, and methods to create subscriber objects whose callback functions are called upon receiving a message from the topic. Examples can be found in References [4] and [5].

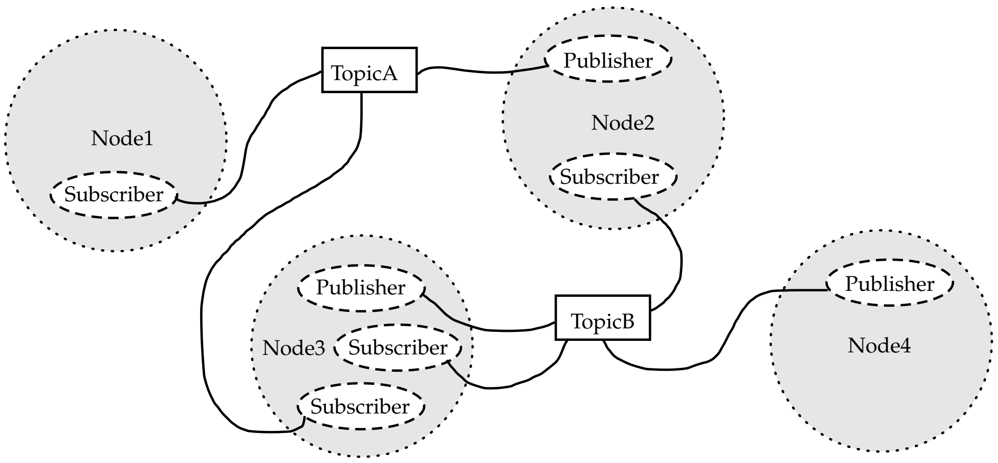

Figure 1.1: Nodes communicating over topics. Node2 publishes messages to TopicA, and Node1 and Node3 subscribe to this topic to receive these messages. Meanwhile, Node3 and Node4 publish messages to TopicB, and Node2 and Node3 subscribe to this topic to receive these messages. That is, it is technically possible for a node to send messages to itself, if needed.

1.3 SERVICES

Compared to topics, services are a slightly more complex means of communication between nodes. In service-client model, one node has a **service** object that receives requests from one or several **client** objects. Upon request, service object's callback function is called, which returns a response which is then sent to the client object (Figure 1.2). While there can be many client objects using the same service, there can only be one service object for said service in the ROS 2 environment. Examples can be found in References [6] and [7].

1.4 ACTIONS

Actions are an advanced means of communication between nodes. They are conceptually composed of services and topics (Figure 1.3). An **action server** object provides a two-phase service for the **action client** objects with optional feedback while providing the service. In the first phase, the goal, the client requests a certain task and gets a response on whether this task can be carried out. If yes, the client proceeds to the second phase to request a result. During the execution of the task, the server can send feedback messages to the client. Finally, the server sends a result response to the client to complete the action. Examples can be found in References [8] and [9].

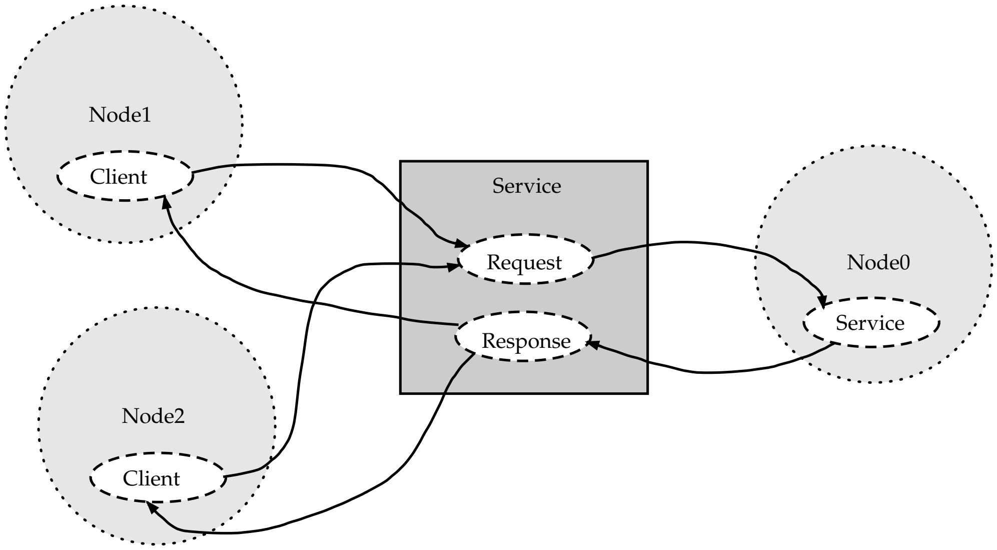

Figure 1.2: Nodes communicating over a service. Node0 acts as a service server, and Node1 and Node2 act as clients. Clients send requests to service, and service replies with responses.

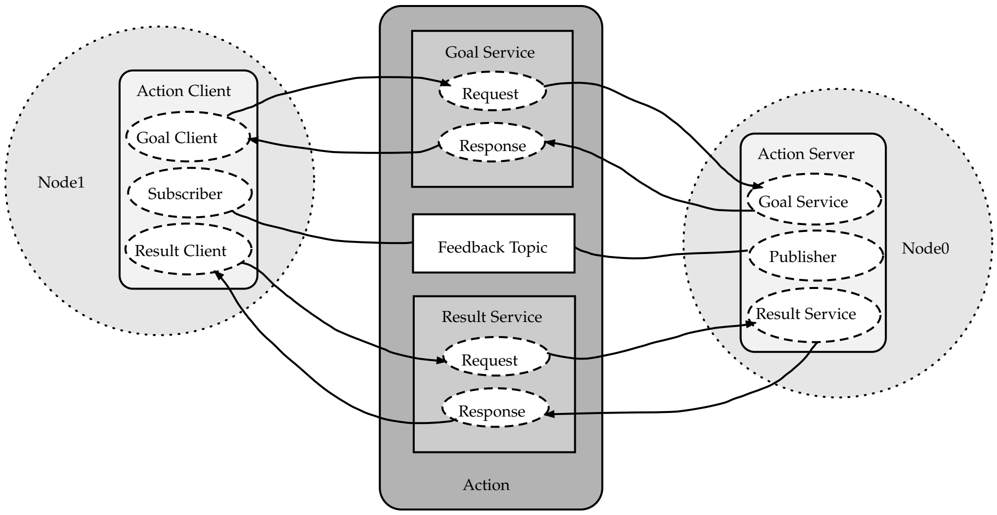

Figure 1.3: Nodes communicating over an action. Node0 acts as an action server, and Node1 acts as an action client. The action client sends a goal request to the action server, which responds by either accepting or rejecting the goal. If the goal is accepted, the action client sends a result request to the action server. While the action server processes the result request, it can publish messages in the feedback topic to keep the action client updated. Finally, the action server sends a result response to the action client.

1.5 COMMUNICATION INTERFACES

To use topics, services and actions for communication, their interfaces must be defined. These interfaces can be defined in a separate package which contains directories for topic messages (`msg`), services (`srv`) and actions (`action`) [10,11].

The structure for a topic message is defined in a `.msg` text file:

```
<data_type0> <variable_name0>
<data_type1> <variable_name1>
.
.
.
<data_typeN> <variable_nameN>
```

`<data_type>` s can be primitive types (such as `int32` or `float64`) or data structures defined in another package.

The structure for a service is defined in a `.srv` text file:

```
<request_data_type0> <request_variable_name0>
<request_data_type1> <request_variable_name1>
.
.
.
<request_data_typeN> <request_variable_nameN>
---
<response_data_type0> <response_variable_name0>
<response_data_type1> <response_variable_name1>
.
.
.
<response_data_typeN> <response_variable_nameN>
```

That is, the service is essentially defined by two messages separated by three dashes (`---`). The first message contains data sent by the client object in request and the second message contains data sent by the service object in response.

The structure for an action is defined in a `.action` text file:

```
<goal_request_data_type0> <goal_request_variable_name0>
<goal_request_data_type1> <goal_request_variable_name1>
.
.
.
<goal_request_data_typeN> <goal_request_variable_nameN>
---
<result_response_data_type0> <result_response_variable_name0>
<result_response_data_type1> <result_response_variable_name1>
.
.
.
<result_response_data_typeN> <result_response_variable_nameN>
---
<feedback_data_type0> <feedback_variable_name0>
<feedback_data_type1> <feedback_variable_name1>
.
.
.
<feedback_data_typeN> <feedback_variable_nameN>
```

That is, the action is defined by three messages separated by dashes. The first message contains data sent by the action client object in the goal request, the second contains data sent by the action server object in the result response, and the third message contains data sent by the action server object as feedback.

1.6 EXECUTORS AND CALLBACK GROUPS

ROS 2 provides a class called `Executor` in both `rclpy` and `rclcpp` libraries. Executors are objects that use one or several threads to coordinate the execution of callback functions [12]. The two types of executors available for both Python and C++ are **single-threaded executor**, which is used by default, and **multi-threaded executor**. In addition, there is a **static single-threaded executor** for C++ to optimize runtime costs.

To handle parallel execution by multiple threads, callback functions run by a multi-threaded executor can (and should) be assigned a specific **callback group**. The two types of callback groups are **mutually exclusive callback group** and **reentrant callback group** [12]. Callback functions assigned in the same mutually exclusive callback group cannot be executed in parallel, while callback functions assigned in the same reentrant callback group can be. In addition, callback functions in different callback groups (of any kind) can be executed in parallel. Note that, if the callback function is not assigned a callback group, it gets assigned to the default, mutually exclusive callback group. That is, if no callback function is assigned to a non-default callback group, the multi-threaded executor essentially acts like a single-threaded executor.

The use of callback groups gives the user more liberty and responsibility to adjust the parallel execution of different modules of his/her application. Careless use of callback groups can lead to data races, which in turn can lead to undefined behaviour of the application. Use of callback groups can also be used as an alternative for asynchronous calls to avoid deadlocks when using services and actions [13].

1.7 SECURITY

Based on DDS, ROS 2 also utilizes specifications of DDS-Security. Out of the five Service Plugin Interfaces (SPIs) defined by DDS-Security, security-related tools of ROS 2, called **Secure ROS 2** (SROS 2), utilize authentication, access control and cryptography to enable secure communication within networks [14]. Guides to enabling SROS 2 -tools in ROS 2 environment are given in Reference [15].

2 Installation and setup of ROS 2

2.1 SETTING UP RASPBERRY PI

In this section, we mainly follow the instructions given in [16]. Do the following:

1. Format your microSD card (with preferably 32 GB or more space). exFAT with an allocation unit size of 32K seems to work fine.
2. Download the Raspberry Pi imager
3. Use Raspberry Pi imager to write an image of the OS on the microSD card, for example, Ubuntu 22.04.1 LTS (Desktop).
4. Insert the microSD card into your Raspberry Pi. Connect your keyboard and mouse to USB ports, and your monitor to the micro HDMI port. Connect the power cable to boot your Raspberry Pi.
5. During installation, choose the English language and **a keyboard layout corresponding to the one you plan to use**. Using the space provided by the installation program, check that letters and special characters print out as you would expect, using the chosen keyboard layout. Complete the installation.
6. (*Optional*) Turn on the firewall with the command: `sudo ufw enable`
7. Connect to the Internet either with Wi-Fi or Ethernet cable. You may need to manually adjust authentication settings upon connecting. For example, using `eduroam` at the UEF Kuopio campus requires setting:
   - Authentication: Protected EAP (PEAP)
   - No CA certificate is required
   - PEAP version: Automatic
   - Inner authentication: MSCHAPv2
8. In the terminal, run:
   ```
   sudo apt update
   sudo apt upgrade
   ```
9. (*Optional*) Install `htop` to monitor the use of resources (CPU and RAM). Run with command `htop`

2.2 (*OPTIONAL*) INSTALLING VISUAL STUDIO CODE ON UBUNTU RUNNING ON RASPBERRY PI

While perhaps not necessary, you may find it useful to install Visual Studio Code on your Raspberry Pi. It can be more straightforward to synchronize your code on Raspberry Pi with your code on GitHub by using the Source Control functionality on Visual Studio Code; however, I do not recommend to use the Visual Studio Code on your Raspberry Pi as the primary IDE to edit your code because you may well find it way slower than editing code on your primary desktop PC or laptop.

With that said, make sure you download the `ARM 64` version of the `.deb` installation file at the Visual Studio Code homepage for your Raspberry Pi.

2.3 INSTALLING ROS 2 ON UBUNTU RUNNING ON RASPBERRY PI (DEBIAN PACKAGES APPROACH)

In this section, we follow the instructions given in [17]; the terminal commands are reproduced verbatim, with the longer commands divided with backslash (`\`) to span multiple lines for better readability. Do the following:

1. First, set locales. Run in terminal:

   ```
   sudo apt update && sudo apt install locales
   sudo locale-gen en_US en_US.UTF-8
   sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
   export LANG=en_US.UTF-8
   ```

2. Setup sources. Enable Universe repository with:

   ```
   sudo apt install software-properties-common
   sudo add-apt-repository universe
   ```

3. Add ROS 2 apt repository with:

   ```
   sudo apt update && sudo apt install curl gnupg lsb-release
   sudo curl -sSL \
   https://raw.githubusercontent.com/ros/rosdistro/master/ros.key \
   -o /usr/share/keyrings/ros-archive-keyring.gpg
   echo "deb [arch=$(dpkg --print-architecture) \
   signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] \
   http://packages.ros.org/ros2/ubuntu \
   $(source /etc/os-release && echo $UBUNTU_CODENAME) main" \
   | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
   ```

4. Install ROS 2 packages with:

   ```
   sudo apt update
   sudo apt upgrade
   sudo apt install ros-humble-desktop
   sudo apt install ros-humble-ros-base
   ```

5. Verify that C++ and Python APIs are working by opening two new terminals. In one, run:

   ```
   source /opt/ros/humble/setup.bash
   ros2 run demo_nodes_cpp talker
   ```

   and in the other one, run:

   ```
   source /opt/ros/humble/setup.bash
   ros2 run demo_nodes_py listener
   ```

If you see that `talker` is `Publishing` messages, and `listener` is confirming it `heard` those messages, then both C++ and Python APIs are functioning properly.

2.4 ENVIRONMENT CONFIGURATION

It is recommended to edit your shell startup script to prevent having to repeat certain commands every time you launch a new terminal [18]. You can source setup files, set domain ID, and limit ROS 2 communication to localhost only, by adding the following lines to the end of your shell startup script (`.bashrc` by default; it is recommended to make a backup file of the original script file before editing, just in case):

```
# ROS 2 -related configuration
source /opt/ros/humble/setup.bash
export ROS_DOMAIN_ID=<your_id>
export ROS_LOCALHOST_ONLY=1
```

In the above, `<your_id>` should be an integer between 0 and 101, inclusive. Naturally, if multiple computers need to access your ROS environment, you can just comment the line `export ROS_LOCALHOST_ONLY=1` .

You will likely need to use `colcon` to build workspaces in your projects, so install it with the command `sudo apt install python3-colcon-common-extensions` . Afterwards, you can add the following lines to your shell startup script to source and set `colcon` -related functions [19]:

```
source /usr/share/colcon_cd/function/colcon_cd.sh
export _colcon_cd_root=/opt/ros/humble/
source /usr/share/colcon_argcomplete/hook/colcon-argcomplete.bash
```

2.5 INSTALLING ROSDEP

`rosdep` is a tool for downloading and installing system dependencies that your ROS 2 packages rely on [20]. To install and initialize it, run in the terminal [21]:

```
sudo apt update
sudo apt install python3-rosdep
sudo rosdep init
rosdep update
```

3 Practical guidelines

In this chapter, we go through how to implement ROS 2 -based solutions in practice.

3.1 GETTING STARTED

To start your project, create a directory for your workspace, resolve dependencies, and create your first package, run:

```
mkdir -p <your_path>/<your_workspace_directory>/src
cd <your_path>/<your_workspace_directory>
rosdep install -i --from-path src --rosdistro humble -y
cd src
ros2 pkg create --build-type <your_ament_type> \
<your_package> --node-name <your_file> \
--dependencies <dep0> <dep1> ... <depN>
```

In the above,

- `<your_path>` can be, for example, just `~`, i.e., home directory.
- `<your_ament_type>` can be either `ament_cmake` or `ament_python`, depending on whether the nodes in your package are written in C++ or Python, respectively. It is also possible to build a package which contains nodes written with both C++ and Python [22].
- `--node-name` option creates a dummy file named `<your_file>` that can be found in `src/<your_package>/src` for C++, and in `src/<your_package>/<your_package>` for Python.
- `--dependencies` option automatically adds dependency lines w.r.t. packages `<dep0>`, `<dep1>`,..., `<depN>` to the file `package.xml` (and to the file `CMakeLists.txt` in C++).

Afterwards, you can create other packages within your `src` folder with the `ros2 pkg create` command, depending on whether your project needs them.

3.2 ADDING PACKAGE DESCRIPTION, LICENSE, AND MAINTAINER

It is a good practice to populate the `package.xml` file with correct information to reduce ambiguity regarding the authorship and licensing of your ROS 2 package. The three fields of note are `description`, `license`, and `maintainer`. If you write your package in Python, you need to copy this information to appropriate fields in the file `setup.py`.

3.3 SETTING DEPENDENCIES

For every package, system dependencies need to be added manually or at least checked if declared in the package creation (Section 3.1). This is achieved by editing the `package.xml` files. There are five types of dependencies [20]: `<depend>`, `<test_depend>`, `<exec_depend>`, `<build_depend>`, and `<build_export_depend>`. `<depend>` is used for mixed purposes, including building and execution. In addition, for C++ packages, the dependencies need to be added to the file `CMakeLists.txt`.

`rosdep` works by checking the `package.xml` file for rosdep keys which refer to the packages and libraries on which your package depends and cross-referencing these keys against a central index to find and install the correct ones [20]. The keys are listed in files `rosdistro/<distro>/distribution.yaml`, `rosdep/base.yaml`, and `rosdep/python.yaml`; however, if no keys exist for your combination of ROS 2 version and OS for a specific library, you can either make a formal pull request and wait for a merge, or just copy this library into your `src` folder in your ROS 2 workspace directory. In this approach, however, it is important **to note the licensing terms of the packages whose source code is copied**. Afterwards, when building the project, you can use the command [20]:

```
rosdep install -i --from-paths src -y --ignore-src --rosdistro humble
```

The option `--ignore-src` ensures that `rosdep` will not try to install dependencies for packages that are already in the workspace.

3.4 ADDING ENTRY POINTS

To run your package in ROS 2 once it is built, you need to tell the entry points to the builder:

- In Python, the addition to `setup.py` file could look like this:

```
entry_points={
        'console_scripts': [
            '<your_executable1> = <your_package>.<your_file1>:main',
            '<your_executable2> = <your_package>.<your_file2>:main',
        ],
    },
```

- In C++, the addition to `CMakeLists.txt` file could look like this:

```
add_executable(<your_executable1> src/<your_file1>.cpp)
ament_target_dependencies(<your_executable1> rclcpp std_msgs)

add_executable(<your_executable2> src/<your_file2>.cpp)
ament_target_dependencies(<your_executable2> rclcpp std_msgs)

install(TARGETS
  <your_executable1>
  <your_executable2>
  DESTINATION lib/${PROJECT_NAME})
```

3.5 BUILDING/COMPILING THE PROJECT

To build/compile your packages to be used in the workspace, run:

```
cd <your_path>/<your_workspace_directory>
rosdep install -i --from-path src --rosdistro humble -y
MAKEFLAGS="-j1 -l1" colcon build --executor \
sequential --packages-select <your_package>
```

Note that omitting the option `--packages-select` will result in every package being built.

3.5.1 A few notes on using colcon build

Make sure your Python `setuptools` version is not newer than `58.2.0`. If it is newer, run in terminal: `pip install setuptools==58.2.0`. Otherwise, the `colcon` build will fail [23]. Python version `3.10.6` seems to work.

Raspberry Pi has a limited amount of RAM; thus, building packages by using the parallel compilation of `colcon` may not be a realistic option, although `colcon` seems to use multiple executors by default. Therefore, it may be necessary to build the packages with the following command [24]:

```
MAKEFLAGS="-j1 -l1" colcon build --executor sequential
```

Adding an option `--symlink-install` will allow "the installed files to be changed by changing the files in the source space (e.g. Python files or other not compiled resourced) for faster iteration" [19]. That is, it has no effect if you write your packages using only C++.

If your OS "freezes" due to running out of memory, try to boot the system by pressing, in order, `R`, `E`, `I`, `S`, `U`, and `B` at approximately 1-second intervals while holding down keys `Alt` and `Prt Scr` / `Sys Rq` [25].

3.6 RUNNING THE PROJECT

Before running your package, you need to source it with:

```
cd <your_path>/<your_workspace_directory>
. install/local_setup.bash
```

By default, you can run one executable in one terminal with the command:

```
ros2 run <your_package> <your_executable>
```

Alternatively, you can create a launch file [26, 27] to get several executables running from one terminal.

4 Demo application

In this chapter, I present a simple demo application that probably clarifies certain concepts related to ROS 2. This application utilizes an inertial measurement unit (IMU) built by one of our former Applied Physics students [28] and the ROS 2 package `turtlesim`. Code for the application, as well as instructions on how to run it, are available on GitHub (`https://github.com/Sandmaenchen/imu_turtle`).

In short, the objective of this application is to drive an object in the `turtlesim` simulation by varying the orientation of the IMU.

4.1 ABOUT THE IMU

The IMU consists of Adafruit BNO055 Absolute Orientation Sensor connected to Arduino MRK1000 WIFI board and enclosed in a 3D-printed plastic case [28]. BNO055 sensor contains a magnetometer, an accelerometer, and a gyroscope, whose combined outputs are used to calculate the orientation of the sensor with respect to Earth and its magnetic field (absolute orientation) or with respect to its initial orientation when it was powered up (relative orientation). The IMU was programmed to output orientation data as strings containing values of unit quaternions through the serial port with a frequency of 10 Hz.

4.2 ORIENTATION REPRESENTATION WITH QUATERNIONS

Quaternions represent a hypercomplex number system, an extension of complex numbers. In other words, they are vectors/arrays with 4 real/floating point numbers in them, but normal linear algebra rules do not apply to them. Usually, quaternions are used, e.g., to improve performance in computing rotations in 3D space. A unit quaternion $\mathbf{q}$ can be expressed as [29]

$$\mathbf{q} = \left[\cos\left(\frac{\alpha}{2}\right), \mathbf{n}\sin\left(\frac{\alpha}{2}\right)\right],$$

which represents a rotation of α radians about a 3D unit vector $\mathbf{n}$. An alternative expression for this is

$$\mathbf{q} = \left[q_w, q_x, q_y, q_z\right].$$

In this demonstration, while reorienting the IMU, we continuously compute rotation from a chosen reference orientation $\mathbf{q}^0$ to the current orientation $\mathbf{q}^1$ of the IMU. This is achieved by computing the Hamilton product between the conjugate $\mathbf{q}^{0*} := \left[q^0_w, -q^0_x, -q^0_y, -q^0_z\right]$ and $\mathbf{q}^1$ [30]

$$\mathbf{q}^r = \mathbf{q}^{0*}\mathbf{q}^1, \tag{4.1}$$

which is also a unit quaternion.

Orientations and rotations in 3D space can also be presented with the Euler angles ϕ, θ, and ψ as three rotations in three mutually orthogonal planes (Figure 4.1).

While the orientation can probably be easier to perceive with the three Euler angles than with the four floating point numbers of a unit quaternion, Euler angles are slower to use in computing due to having to build a 9-element rotation matrix instead of using the unit quaternion per se. Nevertheless, conversion from a unit quaternion **q** to Euler angles can be computed with the following equations [29]

$$\phi = \arctan\left(\frac{2q_y q_z + 2q_w q_x}{2q_w^2 + 2q_z^2 - 1}\right). \tag{4.2}$$

$$\theta = -\arcsin\left(2q_x q_z - 2q_w q_y\right). \tag{4.3}$$

$$\psi = \arctan\left(\frac{2q_x q_y + 2q_w q_z}{2q_w^2 + 2q_x^2 - 1}\right). \tag{4.4}$$

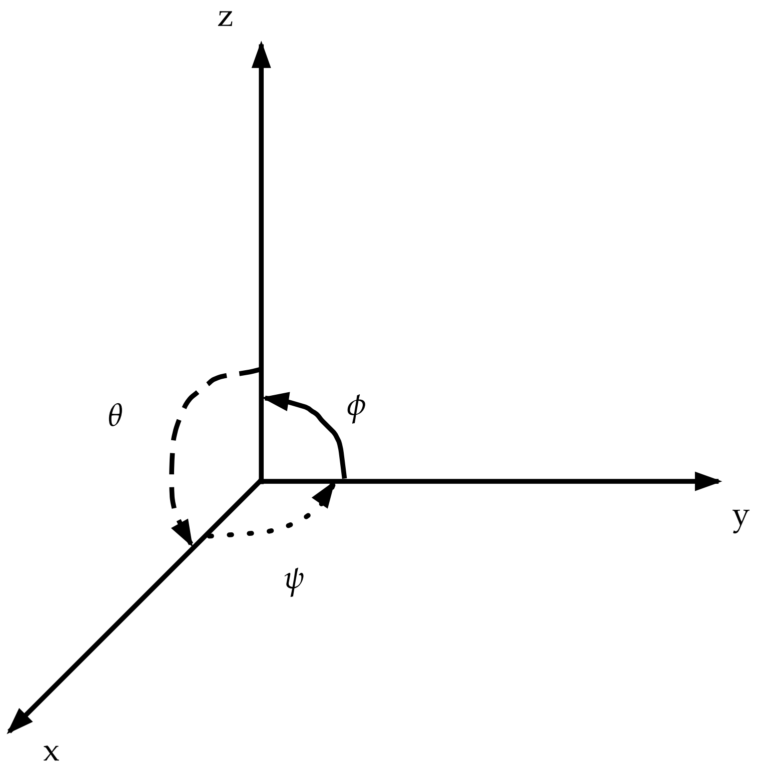

Figure 4.1: Definitions of Euler angles ϕ, θ, and ψ in 3D Cartesian coordinate system.

4.3 LIBRARIES

The demo uses the following libraries:

- Python (v. 3.10.6)
- NumPy (v. 1.23.5)
- `numpy-quaternion` (v. 2022.4.2)
- `pyserial` (v. 3.4)
  - Check to which USB port (`tty`) the IMU is connected, using the command `sudo dmesg | grep tty` . For example, it can be `ttyACM0` .
  - Add yourself to the `dialout` group, using the command `sudo adduser <your_username> dialout` .
- `pynput` (v. 1.7.6)

4.4 ABOUT THE APPLICATION

Figure 4.2 shows the ROS 2 architecture of the demo application. In short:

- `/node1` listens to the serial port (USB) connected to the IMU, strips the values of quaternions from the incoming string with regular expression, and uses these values to build a custom message (`Quaternion.msg`) which is then published to `/topic1` .
- `/node2` listens to the keyboard, and publishes a standard ROS message to `/topic2` when the user presses space.
- `/node3` subscribes to `/topic1` to receive current orientation of the IMU, and to `/topic2` to receive a message when the user wants to use the current orientation of the IMU as a reference orientation (i.e., to define this orientation as the "resting" orientation for the `turtlesim` object), computes the 3D rotation between the current and the reference orientation, uses this rotation to scale linear and angular velocity of the `turtlesim` object, and publishes these velocities to the topic that effectively controls the object in the `turtlesim` simulation.

Each node is run by a separate executor. `/node3` uses a multi-threaded executor with several callback groups. Subscription callbacks related to topics `/topic1` and `/topic2` are handled in a mutually exclusive callback group. These callbacks publish the current and reference quaternions to subscriptions in another callback group, where the computation of rotation and velocity takes place. That is, `/node3` publishes messages for itself. This was done to prevent a possible data race which could happen if one thread utilized the current and reference quaternions to compute rotation and velocity while another thread updated these values. Using a single-threaded executor, on the other hand, led to situations where presses on the keyboard sometimes failed to update the reference orientation because the subscription callback could not be executed during the computation of rotation and velocity.

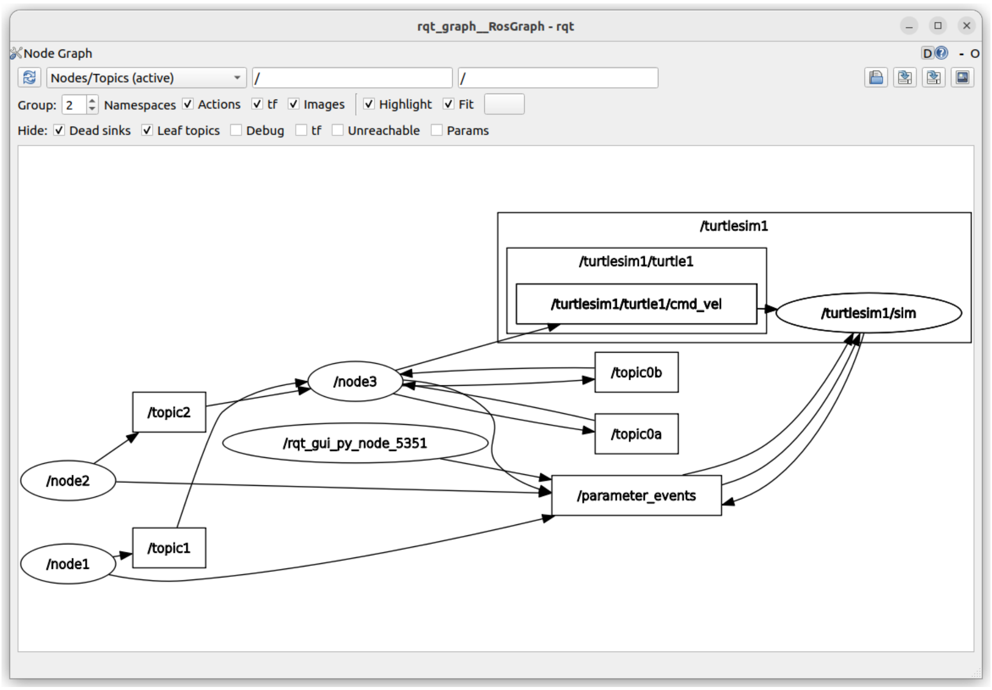

Figure 4.2: Graph of the demo application's ROS 2 environment as displayed by the `rqt_graph` tool. Arrows point to the recipients of the messages.

BIBLIOGRAPHY

[1] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot Operating System 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022. [Online]. Available: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074

[2] Open Source Robotics Foundation, Inc. ROS 2 Documentation: Humble / Concepts / About ROS 2 client libraries. [Online]. Available: https://docs.ros.org/en/humble/Concepts/About-ROS-2-Client-Libraries.html [Accessed: 2022-12-23]

[3] Open Source Robotics Foundation, Inc. rcl: Common functionality for other ROS Client Libraries. [Online]. Available: https://docs.ros2.org/latest/api/rcl/index.html [Accessed: 2022-12-23]

[4] Open Source Robotics Foundation, Inc. ROS 2 Documentation: Humble / Tutorials / writing a simple publisher and subscriber (C++). [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Cpp-Publisher-And-Subscriber.html [Accessed: 2022-12-23]

[5] Open Source Robotics Foundation, Inc. ROS 2 Documentation: Humble / Tutorials / writing a simple publisher and subscriber (Python). [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Py-Publisher-And-Subscriber.html [Accessed: 2022-12-23]

[6] Open Source Robotics Foundation, Inc. ROS 2 Documentation: Humble / Tutorials / writing a simple service and client (C++). [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Cpp-Service-And-Client.html [Accessed: 2022-12-23]

[7] Open Source Robotics Foundation, Inc. ROS 2 Documentation: Humble / Tutorials / writing a simple service and client (Python). [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Py-Service-And-Client.html [Accessed: 2022-12-23]

[8] Open Source Robotics Foundation, Inc. ROS 2 Documentation: Humble / Tutorials / Writing an action server and client (C++). [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Intermediate/Writing-an-Action-Server-Client/Cpp.html [Accessed: 2022-12-23]

[9] Open Source Robotics Foundation, Inc. ROS 2 Documentation: Humble / Tutorials / Writing an action server and client (Python). [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Intermediate/Writing-an-Action-Server-Client/Py.html [Accessed: 2022-12-23]

[10] Open Source Robotics Foundation, Inc. ROS 2 Documentation: Humble / Tutorials / Creating custom msg and srv files. [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Custom-ROS2-Interfaces.html [Accessed: 2022-12-23]

[11] Open Source Robotics Foundation, Inc. ROS 2 Documentation: Humble / Tutorials / Creating an action. [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Intermediate/Creating-an-Action.html [Accessed: 2022-12-23]

[12] Open Source Robotics Foundation, Inc. ROS 2 Documentation: Humble / Concepts / Executors. [Online]. Available: https://docs.ros.org/en/humble/Concepts/About-Executors.html [Accessed: 2022-12-23]

[13] Karelics OY. Deadlocks in rclpy and how to prevent them with use of callback groups. [Online]. Available: https://karelics.fi/deadlocks-in-rclpy/ [Accessed: 2023-01-01]

[14] K. Fazzari. ROS 2 DDS-Security integration. [Online]. Available: https://design.ros2.org/articles/ros2_dds_security.html [Accessed: 2023-03-17]

[15] Open Source Robotics Foundation, Inc. ROS 2 Documentation: Humble / Tutorials / Security. [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Advanced/Security/Security-Main.html [Accessed: 2023-03-17]

[16] Canonical Ltd. How to install Ubuntu Desktop on Raspberry Pi 4. [Online]. Available: https://ubuntu.com/tutorials/how-to-install-ubuntu-desktop-on-raspberry-pi-4#1-overview [Accessed: 2022-11-03]

[17] Open Source Robotics Foundation, Inc. ROS 2 Documentation: Humble / Installation / Ubuntu (Debian). [Online]. Available: https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debians.html [Accessed: 2022-11-03]

[18] Open Source Robotics Foundation, Inc. ROS 2 Documentation: Humble / Tutorials. [Online]. Available: https://docs.ros.org/en/humble/Tutorials.html [Accessed: 2022-11-04]

[19] Open Source Robotics Foundation, Inc. ROS 2 Documentation: Humble / Tutorials / Using colcon to build packages. [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Colcon-Tutorial.html [Accessed: 2022-11-08]

[20] Open Source Robotics Foundation, Inc. ROS 2 Documentation: Humble / Tutorials / Managing Dependencies with rosdep. [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Intermediate/Rosdep.html [Accessed: 2022-11-08]

[21] Open Source Robotics Foundation, Inc. ROS 2 Documentation: Humble / Installation / Ubuntu (binary). [Online]. Available: https://docs.ros.org/en/humble/Installation/Alternatives/Ubuntu-Install-Binary.html [Accessed: 2022-11-10]

[22] The Robotics Back-End. Create a ROS2 package for Both Python and Cpp Nodes. [Online]. Available: https://roboticsbackend.com/ros2-package-for-both-python-and-cpp-nodes/ [Accessed: 2022-11-24]

[23] User ’noshluk2’ at ROS Answers. Setuptoolsdeprecationwarning: setup.py install is deprecated. Use build and pip and other standards-based tools. [Online]. Available: https://answers.ros.org/question/396439/setuptoolsdeprecationwarning-setuppy-install-is-deprecated-use-build-and-pip-and-other-standards-based-tools/?answer=400052#post-id-400052 [Accessed: 2022-11-23]

[24] User ’clalancette’ at ROS Answers. Compilling ROS2 on Rasperry Pi. [Online]. Available: https://answers.ros.org/question/304300/compilling-ros2-on-rasperry-pi/ [Accessed: 2022-11-08]

[25] V. Tunru. What is the equivalent of ’Control-Alt-Delete’? [Online]. Available: https://askubuntu.com/a/95202 [Accessed: 2022-11-08]

[26] Open Source Robotics Foundation, Inc. ROS 2 Documentation: Humble / Tutorials / Creating a launch file. [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Intermediate/Launch/Creating-Launch-Files.html [Accessed: 2022-12-16]

[27] Open Source Robotics Foundation, Inc. ROS 2 Documentation: Humble / Tutorials / Integrating launch files into ROS 2 packages. [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Intermediate/Launch/Launch-system.html [Accessed: 2022-12-16]

[28] M. Demjan, “Wearable inertial measurement unit device: Project Work in Applied Physics,” University of Eastern Finland, Tech. Rep., 2019.

[29] Xsens Technologies B.V., “MTi and MTx User Manual and Technical Documentation,” Tech. Rep., March 2006.

[30] A. Hänninen, “Inertiamittausmoduulien käyttö selän vapaamuotoisten liikkeiden mittauksessa,” Master’s thesis, University of Eastern Finland, May 2014. [Online]. Available: http://urn.fi/urn:nbn:fi:uef-20140785