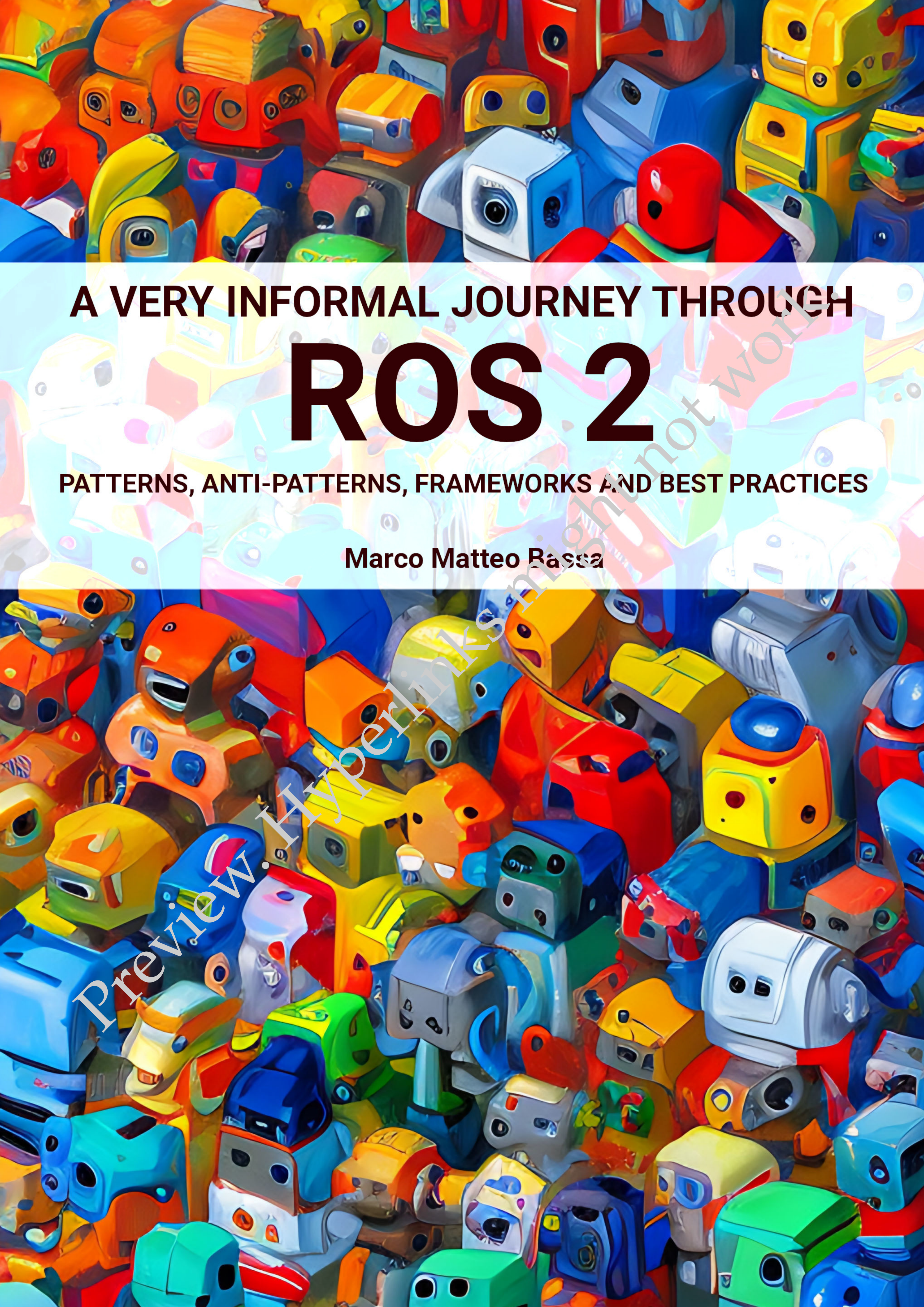


A VERY INFORMAL JOURNEY THROUGH

ROS 2

PATTERNS, ANTI-PATTERNS, FRAMEWORKS AND BEST PRACTICES

Marco Matteo Bassa

A very informal journey through ROS 2

patterns, anti-patterns, frameworks and best practices

Marco Matteo Bassa

Robotics is the art of bringing some piece of your own brain to work on a machine.
I hope you aren't a psychopath.

ISBN: 978-3-00-074685-7

Authored and published by:
Bassa Marco Matteo
Landsberger Str. 247
80687, München
bassamarco91@gmail.com

Disclaimer
The author doesn't assume any liability for the topicality, correctness, completeness or quality of the information provided in this book. Liability claims against the author, of material or immaterial nature, which were caused by the use or disuse of the information provided or by the use of incorrect and incomplete information are excluded, unless intent or gross negligence on the part of the author is proven.

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.

Contents

Introduction 1

1.1 ROS 2

The Robot Operating System, widely known as ROS, was for the first time released in 2007, and has since then revolutionized the way developers create, and most importantly share, software for modern robots around the world. Roboticists tend to have mixed feelings about this framework: while it allows for the fast development of code for robotic platforms, reusing existing modules that are common between different use-cases, it is often criticized for not being an "industry ready solution", good only for prototyping and for research purposes. In the past few years, many companies have been fighting against this idea, bringing on the market solutions based on ROS capable of running with good reliability over an extended period of time. However, over time, the ROS community has developed an awareness of the limitations of the original architecture, and realized that it would not be possible to overcome all of them through incremental upgrades.

As a result, in 2017, a new version of the operating system appeared in the robotics world: ROS 2. Along the book more insights will be given about why this transition happened / is happening, for now the reader should know that support for the last release of "ROS1"[1], 'Noetic Ninjemis', is set to end in May 2025. As a consequence, the whole community was invited to continue the development of new features using the new version of ROS. Another important aspect of the transition is that, while the architecture of ROS 2 resembles in many aspects that of the original ROS, the two are not compatible. Even though some ROS packages can be converted to/from the new version of ROS with relatively little effort, a successful transition often involves significant architectural changes across the entire system.

1: I will sometimes use this notation instead of "ROS" to make de distinction between the old and the new version clearer.

1.2 Why this book?

The ROS community made significant efforts to provide users with tutorials [1], documentation [2] and open discussion forums like discourse.ros.org[3] or answers.ros.org[4] in order to make the learning and the troubleshooting of ROS 2 as smooth as possible. Chances are that, before reading this, you already went through most of it. If not, please do it. This book is not a programming guide that teaches ROS 2 from the basics, as the previously mentioned resources already do that exceptionally well. There is plenty of free, dynamic and entertaining material for beginners that will enable you to create your first ROS 2 nodes in no time. However, while the ROS documentation is really good at explaining how to perform "single operations", if you'll be dealing with the development of a large system, you will likely soon be asking yourself questions like: "is this the proper way of doing stuff?", "How does the rest of the world usually tackle this problem using ROS?". These questions almost inevitably arise because ROS often provides the developers with multiple ways to achieve the same objective. Oftentimes, the answer to these dilemmas will not be unique and universal, but will depend from the requirements and the constrains of the specific system. However, it is also true that some patterns can be identified that generally make the architecture more reusable, modular and less error-prone. At the same time, anti-patterns exist that, if introduced in your code, will most likely make you struggle and lose time / performance as the system expands.

1.3 How and when to read this book

Given the big amount of quality material about ROS 2 already existing on the web, this book will strive to avoid repeating basic examples and concepts, and will instead make extensive use of references and links to direct you to the most current information available. As this is a book about programming, the author assumes that you have access to some form of digital device that will facilitate your ability to follow the references and try out examples (so please don't print this and save some tree).

Each chapter ends with a short homework assignment. While you may be tempted to (and most likely will) skip it and quickly move on with the book, it's important to note that "The biggest drop in retention happens soon after learning. Without reviewing or reinforcing our learning, our ability to retain information plummets"[5]. In the context of programming tools, I believe it's crucial to put what you've learned into practice by writing your own code. With time, I will provide the solution to some of the exercises on this github repository and in the last chapter. If you don't find what you are looking for, feel free to contact me to discuss any solution.

Regardless of whether you are an expert robotic engineer used to work with the "old ROS", a developer used to work with other kinds of frameworks, or someone venturing for the first time into the magic world of robotics, this book will hopefully provide you with some good tips on how to choose and properly use the correct tools in the ROS 2 world.

Many robots have been cursed, hit, smashed into walls and boxes and (partially) burned while gaining the material on which this book was based. I hope I can help to make your journey less troubled, but just as fun as mine.

2 The Node

In ROS 2, just like in the dear old ROS, the basic unit of execution for each program is called a Node. While it would look like there's not much to say about it [1][6], the way you'll create your Nodes deeply affects the way your application can be managed, integrated into the rest of the system, and the functionalities it will be able to provide.

1: A description of the core Node concept can be found here

2.1 How to (properly) instantiate it

As you probably already figured out, there are two main ways to create a Node. The most immediate one, if you are using rclcpp, will look something like: [2]

2: Most of the code samples provided through this book are written in C++. The described concepts, however, almost always generalize to the other languages that can be used with ROS 2. Later in the book we will see how ROS 2 facilitates the support of most of its functionalities across different programming languages.

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

The node is created as a shared pointer to a rclcpp::Node class instantiated with your favourite name. This pointer can now be used to read parameters, create publishers, subscribers, and all the other fancy features[7] that a good node should provide. You can also share this little guy (that somewhat resembles the retired nodehandle[8]) with other functions or classes, who can optionally store a copy of it. After all, it's a shared pointer, why not? We'll soon see how, in many cases, this solution can prove to be problematic.

Another way to instantiate a node, is through a class deriving from rclcpp::Node :

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

class OurClass : public rclcpp::Node {
```

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

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

The first thing that you'll notice is that we now have to write more code, so why bother to do this? The key answer to this is reusability. Once you created your own Node class, it is easy to isolate it into a library that is independent from the execution of a process. This allows us to easily replicate and share whatever logic we decided to implement in our Node, at least at compile-time. ROS 2 actually allows us to do something even better than that: the execution of a node can be controlled at runtime. In order to achieve that, another couple of steps are required.

2.1.1 Components and when to use them

Components are **the recommended way to create nodes in ROS 2**. Once a Node is created as a component, its execution can (optionally) be controlled by an executor[3], who can run multiple components at the same time. Let's take a look at how to make our class a component.

3: In Chapter 3 you will find a detailed description of executors.

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

namespace our_namespace
{

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

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

In addition to this, some trick needs to be applied in the CMakeLists.txt file of our package:

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

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

ament_target_dependencies(our_lib
  "rclcpp"
  "rclcpp_components")

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

What we did here is creating a way to allow the component to be discoverable when its library is being loaded into a running process (for example the executor). A component can be loaded into an executor at runtime using command line commands(see here[9] for the details) , through a launchfile (see this example[10]), or can be manually integrated into your program at compile time, just like we did with our class above. Since it is compiled as a shared library, you can easily export it and link other packages against it[11]. If you are compiling for Windows, you might want to check these tips [12].

While it is common for components to derive from rclcpp::Node, this is not mandatory. The requirements for a class to be exported as a component are:

- Have a constructor that takes a single argument that is a rclcpp::NodeOptions instance
- Have a method of the signature:

```
rclcpp::node_interfaces::NodeBaseInterface::SharedPtr
    get_node_base_interface(void)
```

When should we implement our nodes as components? The short answer is: always.[4]

4: Ok, you can actually skip this for small scripts and tests, but it's a good idea to use it for your core components.

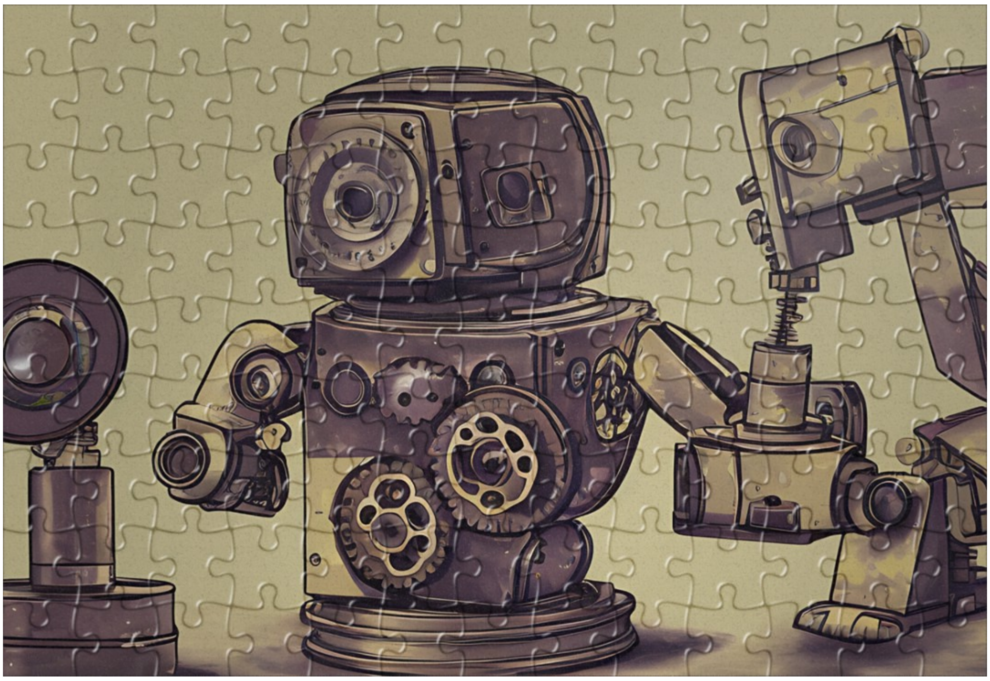
You might think that it's not worth the overhead of doing all these declarations if, at the end of the day, you don't plan to put your node in an executor at runtime. However, you should not assume that the way you are using your code today is the way it will be used tomorrow. One day, your grandchildren might stumble upon your old component and decide that it should run in an executor. Since there is no relevant drawback (at least that I know) from implementing it as a component, you shouldn't be too lazy and adopt from the beginning a structure that will make the life of your grandchildren easier.

Now that we've structured our node as a component, you might be asking yourself: "Why would my grandchildren want to run multiple components within a single executor?"; there are three possible motivations I can think about (but let me know if you come up with more):

- While an executor can spawn as many threads as it likes, all the managed components will be running in the same process. This reduces a little bit the effort made by your operating system when performing context switching, since switching between threads of the same process is slightly faster[13]. Some experiments[14] have shown how the communication latency between two nodes is significantly lower if these are running within the same process.
- When two components are running within the same process, they will be sharing the same memory space, and hence cool features like the zero copy publish and subscribe mechanism[15] can be used [5]. This is especially useful when dealing with devices producing a big amount of data, like for example cameras: if both the drivers and the consumer are implemented as components and run in the same executor, they will be able to transfer data using smart pointers. If they are executed in different processes, ROS will instead perform data serialization without any need for modifications in your code.
- Putting many components with a limited workload into an executor that is using a small number of threads (or only one), can

5: In case you are wondering, yes, this is the ROS 2 equivalent of the old nodelet[16], however in ROS 2 their role is much more important reduce the total number of threads running on the system, which results, especially on low-end platforms, in a happier scheduler and increased overall performance.

There is anyway an implicit danger when putting multiple components on the same process. Sharing the same memory space implies that errors like a segmentation fault will no longer crash only the affected component, but the entire executor with his family. Consequently, when "merging" two components in the same executor, you should ask yourself: "Should one of them run into trouble, is it important for the other to continue normal operation?". Let's consider for example a component implementing a camera driver and a component using the data from the camera to detect people in a room. If the camera driver dies, the other component probably becomes useless, and it's ok for it to stop working. However, if the camera is additionally used for something else, you'll have to choose if you want to keep those functionalities active in the case of a people-detector malfunction. You probably shouldn't put your camera driver together with the driver of your mobile base. If the camera driver crashes, you still want to be able to drive and maybe charge your robot while waiting for help. Another consequence of using the components structure, is that it forces the programmer to give-up the control over the spinning of the callbacks. Components are meant to be paired with an executor; calling functions like *spin_some*, *spin_once*, and *spin_until_future* on an already paired node results in an exception. In Chapter 3, I will explain how to properly manage this inconvenience.



2.2 The Node types

After taking a look at the ROS 2 tutorials and at some examples[17], you might be tricked into thinking that, in order to write a ROS node, your class will have to derive from rclcpp::Node, or rclpy.node. There's nothing wrong in deriving from these classes, and this is indeed the most convenient way to create your node; however, you should not assume that any node you'll be dealing with will be using them. Other classes exist, that allow you to create a node with special functionalities. You could also choose to create your own customized version.

2.2.1 Managed Nodes

The most common kind of "special node" you'll likely encounter while strolling around the ROS 2 world is the lifecycle node[18][6]. When using this as base for your class, the resulting node will become a "managed node", whose internal state can be controlled through a set of service calls. A description of all the states and transitions that such a node implements, can be found on the corresponding design article[20].

6: A python based implementation is available here[19]

The transitions of a set of managed nodes are usually handled by a dedicated manager, which can implement any kind of customized logic to choose the order in which the nodes are configured, activated, and eventually deactivated. A complete example can be found in the Nav 2 lifecycle manager[21], that, in addition to handle the initialization of its components, keeps track of their status using a heartbeat strategy.

Starting from ROS 2 Iron, it is possible to trigger these transitions directly from a launchfile using a dedicated roslaunch action: LifecycleTransition. Here is an example of launcher using this functionality to configure and activate two nodes:

```python
from launch import LaunchDescription
from launch_ros.actions import LifecycleNode
from launch_ros.actions import LifecycleTransition
from lifecycle_msgs.msg import Transition

def generate_launch_description():
  return LaunchDescription([
    LifecycleNode(package='package_name', executable='a_managed_node',
     name='node1'),
    LifecycleNode(package='package_name', executable='a_managed_node',
     name='node2'),
    LifecycleTransition(
      lifecycle_node_names=['node1', 'node2'],
      transitions_ids=[
        Transition.TRANSITION_CONFIGURE,
        Transition.TRANSITION_ACTIVATE]
    )
])
```

This strategy, however, should be used only for test purposes, since it can't perform proper error management.

Lifecycle-managed nodes are very handy when the program is dealing with components implementing a long initialization routine. If other components in the system need to wait for the initialization of the previous ones, a waiting strategy needs to be implemented. The managed nodes provide in this case a clean, standardized way to control their ordered initialization or their deactivation.

If you take a closer look at the implementation of the lifecycle node, you'll notice that this doesn't inherit from rclcpp::Node, but only uses some of

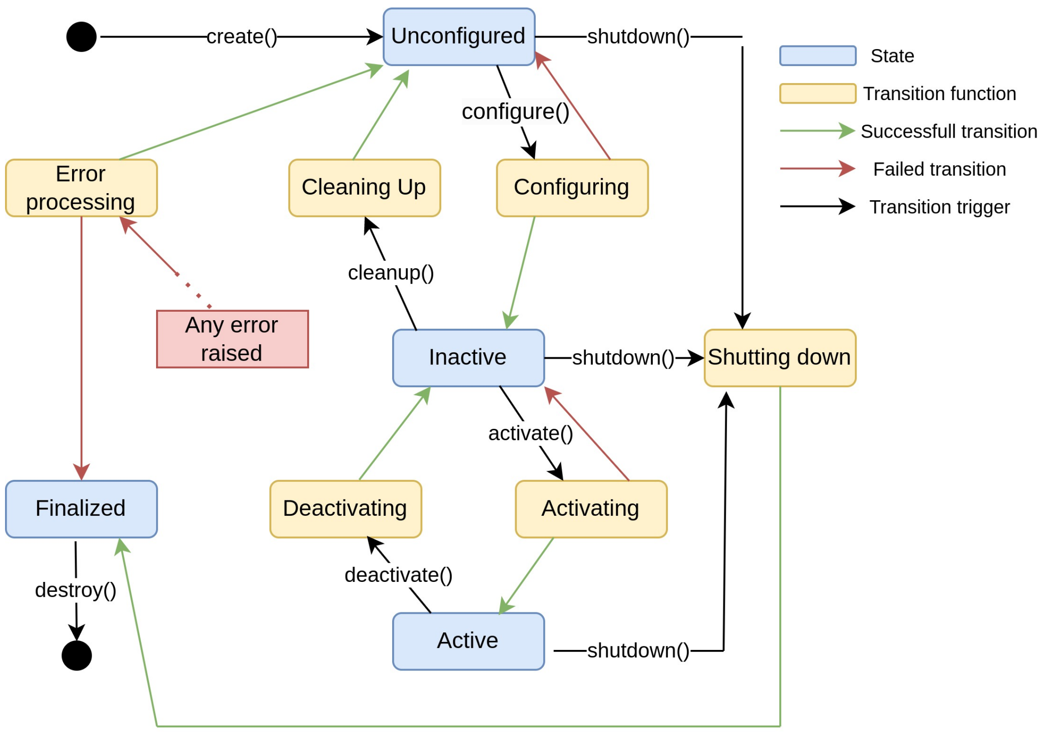


Figure 2.1: Diagram of the state machine implemented by lifecycle managed nodes. Based on `https://design.ros2.org/articles/node_lifecycle.html`

its components, while providing a new set of interfaces to access them. We'll see why in the coming section.

2.2.2 Custom base nodes

If you are not satisfied with the current implementation or rclcpp::Node, and you want to create a new class that provides additional or modified functionalities, you have multiple options:

The first thing that could come to your mind, is to simply inherit from rclcpp::Node and extend its public interface with additional functionalities. This works perfectly fine until the moment your application needs to provide some of these functionalities to a nested class or library. [7] In this scenario, you won't be able to share your custom node using smart pointers, since a call to *shared from this* will resolve the base rclcpp::Node. This partially explains why the implementation of the lifecycle node doesn't inherit from Node, but verbosely reproduces its interfaces. The process is a bit longer, but the resulting node is much more versatile.

7: It is a common practice to implement libraries / classes that, even if interacting with the ros apis, are not intended to be used as a stand-alone node, but within the context of an existing one. If you need to be able to load these classes at runtime, plugins[22] might help you.

To provide a node with the functionalities necessary to interface with ROS 2 without deriving from rclcpp::Node, a possible way is to instantiate the required node interfaces[23]. Most of these classes require to first instantiate a NodeBaseInterface object, which can be seen as the "core" of the node, and then use it to provide the different functionalities that you would usually access using a node pointer. For example, the NodeTopics class will allow you to create publishers and subscriptions to topics, the NodeParameters to interact with parameters and so on. Once these classes are instantiated, you can wrap and combine them to create your own custom interface. If you respect the rules described in the components section, and declare your class using *enable_shared_from_this*[24], your node will be ready to be run as base for your applications.

Is there a downside in doing all this? Of course, building your system around a custom node implementation drastically reduces your capability of sharing it: whenever you will want to distribute a component based on it, the node implementation will have to follow. Custom nodes also create a very strong "dependency point": having many components depending on it implies that modifying its functionalities while ensuring to not break the existing behaviour of the system is going to be more and more difficult as your system expands. Moral of the story: unless you have a very good reason, stick with the existing default node-classes.

2.3 Using nodes in the interfaces

As we've just seen, even if the rclcpp::Node is the most used and convenient base class when creating a node, it is not the only one you might encounter during your ROS 2 adventure. As a consequence, whenever you implement a library or a class that requires ros functionalities, providing them through a pointer to rclcpp::Node will reduce its re-usability, since different kind of nodes won't be able to use it [8]. Using a pointer to your own custom node will make it even less re-usable, and re-usability is one of the things that make ROS great in the first place.

8: An example of component suffering from this issue can be found here.

So what can we do to generalize our solution? Here are some alternatives:

- Use the aforementioned node interfaces classes: if your library, for example, only requires to access parameters, provide a NodeParameters object instead of a full node. These interface classes will most likely be instantiated and exposed by any kind of node, so you should be safe (for example[25] executors were designed to take the node base interface as input).
- Rethink your design to use multiple small-nodes instead of integrating small libraries or plug-ins within the same program. Using ROS interfaces (topics, services, actions) makes it easier for other components to access the shared information without having to perform any change on the code of the data-producers. This doesn't apply if you need very deterministic low latency interfaces between the components, since real time ROS interfaces[26] are still only experimental.
- Try to make your component ROS-agnostic. If the ROS specific functionalities used by your code are limited and easy to replace, keeping it ROS-agnostic might not only force you to minimize its interface, but also make it more resilient to updates in the ROS libraries. This should make your packages (or at least a piece of them) easier to maintain, especially while transitioning to a new ROS version like ROS3.

Since sometimes none of these approaches leads to a clean implementation, a new strategy was recently developed for dealing with different kinds of nodes in a generic way: the "Generic node interfaces".

2.3.1 Generic node interfaces

Starting from ROS 2 Iron, it is possible to wrap different node interfaces into a unified object: the NodeInterfaces class[27]. Using this object, it is possible to provide to a function or a class multiple node-functionalities using only one parameter.

To better explain this, I'll report an example from here. Suppose that we have a function and that we want to use some of the functionalities provided by a node within it. Using the classic node interfaces, you would write a function definition like:

```

void fn(NodeBaseInterface::SharedPtr base_interface, NodeClockInterface::
    SharedPtr clock_interfaces);
```

You would later call fn like this:

```

rclcpp::Node node("some_node");
fn(node->get_node_base_interface(),
   node->get_node_clock_interface());
```

Using the NodeInterfaces class we can instead define:

```

void fn(NodeInterfaces<NodeBaseInterface, NodeClockInterface> interfaces);
```

The function fn can now be called in two different ways:

```

// explicitly
auto ni = NodeInterfaces<NodeBaseInterface, NodeClockInterface>(node);
fn(ni);

// implicitly
fn(node);
```

The implicit form is probably the one you will want to use: the function call is now much more compact and elegant, resembling what in ROS1 was done with a node-handle. If you need it, it is additionally possible to subset a node interface, to build it from aggregation or to extract its internal interfaces:

```

// subsetting
auto ni_base = NodeInterfaces<NodeBaseInterface>(ni);
auto ni_aggregated = NodeInterfaces<NodeBaseInterface, NodeClockInterface
    >(
  node->get_node_base_interface(),
  node->get_node_clock_interface()
)

auto base = ni.get<NodeBaseInterface>();
auto clock = ni.get_clock_interface();
```

Homework

Create two managed nodes as components, and a third node that, upon getting a request through a service, will initialize and start them sequentially. Put all these components within the same executor using a launchfile. You can find hints in the linked documentation.

The solution to the homework can be found here

Bibliography

[1] *ROS2 tutorials*. URL: https://docs.ros.org/en/rolling/Tutorials.html (cited on page 1).

[2] *ROS2 documentation*. URL: https://docs.ros.org/en/humble/index.html (cited on page 1).

[3] *Discourse.ros.org*. URL: https://discourse.ros.org (cited on page 1).

[4] *Answers.ros.org*. URL: https://answers.ros.org/questions (cited on page 1).

[5] *Ebbinghaus's Forgetting Curve*. URL: https://www.mindtools.com/a9wjrjw/ebbinghauss-forgetting-curve (cited on page 2).

[6] *ROS2 Concepts*. URL: https://docs.ros.org/en/rolling/Concepts.html (cited on page 3).

[7] *Rclcpp API, Node*. URL: https://docs.ros.org/en/rolling/p/rclcpp/generated/classrclcpp_1_1Node.html (cited on page 3).

[8] *ROS nodehandle*. URL: http://wiki.ros.org/roscpp/Overview/NodeHandles (cited on page 3).

[9] *Composition tutorials*. URL: https://docs.ros.org/en/rolling/Tutorials/Intermediate/Composition.html (cited on page 5).

[10] *Composition demo*. URL: https://github.com/ros2/demos/blob/master/composition/launch/composition_demo.launch.py (cited on page 5).

[11] *Best practice for sharing libraries*. URL: https://discourse.ros.org/t/ament-best-practice-for-sharing-libraries/3602 (cited on page 5).

[12] *Windows tips and tricks*. URL: https://docs.ros.org/en/rolling/The-ROS2-Project/Contributing/Windows-Tips-and-Tricks.html#windows-symbol-visibility (cited on page 5).

[13] *Steps in context switching*. URL: https://stackoverflow.com/questions/7439608/steps-in-context-switching (cited on page 5).

[14] *ROS 2 Latencies for High Frequencies using the EventExecutor*. URL: https://bit-bots.de/en/2023/03/ros-2-latencies-for-high-frequencies-using-the-eventexecutor/ (cited on page 5).

[15] *Intra process communication tutorial*. URL: https://docs.ros.org/en/rolling/Tutorials/Demos/Intra-Process-Communication.html (cited on page 5).

[16] *ROS nodelet*. URL: http://wiki.ros.org/nodelet (cited on page 5).

[17] *ROS2 examples*. URL: https://github.com/ros2/examples (cited on page 7).

[18] *Rclcpp lifecycle node*. URL: https://github.com/ros2/rclcpp/tree/master/rclcpp_lifecycle (cited on page 7).

[19] *Rclpy lifecycle node*. URL: https://github.com/ros2/rclpy/tree/rolling/rclpy/rclpy/lifecycle (cited on page 7).

[20] *Node lifecycle design*. URL: http://design.ros2.org/articles/node_lifecycle.html (cited on page 7).

[21] *Nav2 lifecycle manager*. URL: https://navigation.ros.org/configuration/packages/configuring-lifecycle.html (cited on page 7).

[22] *Pluginlib docs*. URL: https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Pluginlib.html (cited on page 8).

[23] *Node interfaces*. URL: https://docs.ros2.org/beta1/api/rclcpp/namespacerclcpp_1_1node__interfaces.html (cited on page 8).

[24] *Shared from this cppreference*. URL: https://en.cppreference.com/w/cpp/memory/enable_shared_from_this (cited on page 9).

[25] *Rclcpp Executor class, add node*. URL: https://docs.ros2.org/beta2/api/rclcpp/classrclcpp_1_1executor_1_1Executor.html#a363491ae55c619db310861a3ef4cc4b0 (cited on page 9).

[26] *ROS2 realtime beta api*. URL: https://github.com/ros-realtime (cited on page 9).

[27] *The NodeInterfaces class*. URL: https://docs.ros.org/en/ros2_packages/rolling/api/rclcpp/generated/classrclcpp_1_1node__interfaces_1_1NodeInterfaces.html#exhale-class-classrclcpp-1-1node-interfaces-1-1nodeinterfaces (cited on page 10).

[28] *Writing a simple publisher and subscriber*. URL: https://docs.ros.org/en/rolling/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Cpp-Publisher-And-Subscriber.html (cited on page 12).

[29] *ROS2 timer demo*. URL: https://github.com/ros2/demos/blob/master/demo_nodes_cpp/src/timers/one_off_timer.cpp (cited on page 12).

[30] *ROS2 service tutorial*. URL: https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Cpp-Service-And-Client.html (cited on pages 12, 32).

[31] *ROS2 add two ints demo*. URL: https://github.com/ros2/demos/blob/master/demo_nodes_cpp/src/services/add_two_ints_client_async.cpp (cited on page 12).

[32] *Executor concepts*. URL: https://docs.ros.org/en/rolling/Concepts/About-Executors.html (cited on page 13).

[33] *Executor spin functions*. URL: https://docs.ros2.org/galactic/api/rclcpp/classrclcpp_1_1Executor.html (cited on page 13).

[34] *Single threaded executor rclcpp class reference*. URL: https://docs.ros2.org/galactic/api/rclcpp/classrclcpp_1_1executors_1_1SingleThreadedExecutor.html (cited on page 13).

[35] *Multi threaded executor rclcpp class reference*. URL: https://docs.ros2.org/galactic/api/rclcpp/classrclcpp_1_1executors_1_1MultiThreadedExecutor.html (cited on page 13).

[36] *Callbacks and executor example*. URL: https://github.com/nirwester/ros2_journey_examples/tree/master/callbacks_and_executors (cited on page 13).

[37] *Wiki page for Deadlock*. URL: http://wiki.c2.com/?DeadLock (cited on page 15).

[38] *How to use callbacks docs*. URL: https://docs.ros.org/en/humble/How-To-Guides/Using-callback-groups.html (cited on page 15).

[39] *Advantages and Disadvantages of a Multithreaded/Multicontexted Application*. URL: https://docs.oracle.com/cd/E13203_01/tuxedo/tux71/html/pgthr5.htm (cited on page 15).

[40] *Difference between concurrency and parallelism*. URL: https://www.geeksforgeeks.org/difference-between-concurrency-and-parallelism (cited on page 16).

[41] *Why too many threads hurt performance and what to do about it*. URL: https://www.codeguru.com/cplusplus/why-too-many-threads-hurts-performance-and-what-to-do-about-it (cited on page 16).

[42] *The microros framework home page*. URL: https://micro.ros.org (cited on page 18).

[43] *Execution management in micro-ROS*. URL: https://micro.ros.org/docs/concepts/client_library/execution_management (cited on page 19).

[44] Christoph M. Kirsch and Ana Sokolova. 'The logical execution time paradigm'. In: (2012). DOI: 10.1007/978-3-642-24349-3_5 (cited on page 19).

[45] *Data Distribution Service specifications*. URL: https://www.omg.org/spec/DDS (cited on page 21).

[46] *Concepts for internal interfaces*. URL: https://docs.ros.org/en/humble/Concepts/About-Internal-Interfaces.html (cited on page 22).

[47] *Roscpp*. URL: http://wiki.ros.org/roscpp (cited on page 21).

[48] *Rospy*. URL: http://wiki.ros.org/rospy (cited on page 21).

[49] *Working with multiple RMW implementations*. URL: https://docs.ros.org/en/humble/How-To-Guides/Working-with-multiple-RMW-implementations.html (cited on page 22).

[50] *ROS2 Galactic distribution*. URL: https://docs.ros.org/en/galactic/index.html (cited on page 22).

[51] *RMW CycloneDDS*. URL: https://index.ros.org/r/rmw_cyclonedds (cited on page 22).

[52] *ROS2 Humble distribution*. URL: https://docs.ros.org/en/rolling/Releases/Release-Humble-Hawksbill.html (cited on page 22).

[53] *RMW FastRTPS*. URL: https://github.com/ros2/rmw_fastrtps (cited on page 22).

[54] *EProsima FastDDS*. URL: https://www.eprosima.com/index.php/products-all/eprosima-fast-dds (cited on page 22).

[55] *Eclipse Cyclone DDS*. URL: https://projects.eclipse.org/projects/iot.cyclonedds (cited on page 22).

[56] *RTI Connext DDS*. URL: https://www.rti.com/products (cited on page 22).

[57] *Gurum DDS*. URL: https://gurum.cc/index_eng (cited on page 22).

[58] *What is a socket connection?* URL: https://www.ibm.com/docs/en/zvse/6.2?topic=SSB27H_6.2.0/fa2ti_what_is_socket_connection.html (cited on page 23).

[59] *Typical udp-socket session*. URL: https://www.ibm.com/docs/en/zos/2.2.0?topic=services-typical-udp-socket-session (cited on page 23).

[60] *Concepts about Quality-of-Service Settings*. URL: https://docs.ros.org/en/rolling/Concepts/About-Quality-of-Service-Settings.html#qos-policies (cited on page 23).

[61] *Publishing and subscription options*. URL: https://wiki.ros.org/roscpp/Overview/Publishers%5C%20and%5C%20Subscribers#Publisher_Options (cited on page 23).

[62] *Quality of service demo, deadline*. URL: https://github.com/ros2/demos/blob/master/quality_of_service_demo/rclcpp/src/deadline.cpp (cited on page 24).

[63] *Quality of service demo, lifespan*. URL: https://github.com/ros2/demos/blob/master/quality_of_service_demo/rclcpp/src/lifespan.cpp (cited on page 24).

[64] *Liveliness demo*. URL: https://github.com/ros2/demos/blob/master/quality_of_service_demo/rclcpp/src/liveliness.cpp (cited on page 24).

[65] *Quality of service profiles*. URL: https://docs.ros.org/en/rolling/Concepts/About-Quality-of-Service-Settings.html#qos-profiles (cited on page 24).

[66] *Quality of service communication policies*. URL: https://github.com/ros2/rmw/blob/rolling/rmw/include/rmw/qos_profiles.h (cited on page 24).

[67] *Quality of service compatibility tables*. URL: https://docs.ros.org/en/rolling/Concepts/About-Quality-of-Service-Settings.html#qos-compatibilities (cited on page 25).

[68] *Rviz 2*. URL: https://github.com/ros2/rviz (cited on page 26).

[69] *Understanding ROS2 topics*. URL: https://docs.ros.org/en/rolling/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html (cited on page 26).

[70] *How to create a simple publisher and a subscriber*. URL: https://docs.ros.org/en/rolling/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Cpp-Publisher-And-Subscriber.html (cited on page 26).

[71] *Introducing Tf2*. URL: https://docs.ros.org/en/galactic/Tutorials/Intermediate/Tf2/Introduction-To-Tf2.html (cited on page 28).

[72] *Recording and playing back data tutorial*. URL: https://docs.ros.org/en/rolling/Tutorials/Beginner-CLI-Tools/Recording-And-Playing-Back-Data/Recording-And-Playing-Back-Data.html (cited on page 28).

[73] *Topics and service name constrains*. URL: https://design.ros2.org/articles/topic_and_service_names.html#ros-2-topic-and-service-name-constraints (cited on page 28).

[74] *Logging verbosity levels*. URL: http://wiki.ros.org/Verbosity%5C%20Levels (cited on page 28).

[75] *Launch files in different formats*. URL: https://docs.ros.org/en/rolling/How-To-Guides/Launch-file-different-formats.html (cited on page 29).

[76] *Static remapping*. URL: https://design.ros2.org/articles/static_remapping.html (cited on page 30).

[77] *Understanding ROS2 services*. URL: https://docs.ros.org/en/ros2_documentation/rolling/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.html (cited on page 31).

[78] *ROS1 services*. URL: http://wiki.ros.org/roscpp/Overview/Services (cited on page 31).

[79] *Understanding ROS2 actions*. URL: https://docs.ros.org/en/rolling/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html (cited on page 35).

[80] *Actions documentation*. URL: https://design.ros2.org/articles/actions.html (cited on page 36).

[81] *Actions tutorial*. URL: https://docs.ros.org/en/galactic/Tutorials/Intermediate/Writing-an-Action-Server-Client/Cpp.html (cited on page 35).

[82] *Behavior trees Wikipedia page*. URL: https://en.wikipedia.org/wiki/Behavior_tree_(artificial_intelligence,_robotics_and_control) (cited on page 37).

[83] *The Nav2 project*. URL: https://navigation.ros.org/concepts/index.html (cited on pages 37, 69).

[84] *The "common interfaces" repository*. URL: https://github.com/ros2/common_interfaces (cited on page 37).

[85] *Define custom interfaces*. URL: https://docs.ros.org/en/rolling/Tutorials/Beginner-Client-Libraries/Single-Package-Define-And-Use-Interface.html (cited on page 37).

[86] *The ROS1 bridge package*. URL: https://github.com/ros2/ros1_bridge (cited on page 38).

[87] *Type adaptation feature description*. URL: https://ros.org/reps/rep-2007.html (cited on page 39).

[88] *Type negotiation feature description*. URL: https://ros.org/reps/rep-2009.html (cited on page 41).

[89] *Description of the separation of concerns principle*. URL: https://nalexn.github.io/separation-of-concerns/ (cited on page 43).

[90] *What is a PLC?* URL: https://www.amci.com/industrial-automation-resources/plc-automation-tutorials/what-plc/ (cited on page 45).

[91] *Dbpedia page for Finite State Machine*. URL: https://dbpedia.org/page/Finite-state_machine (cited on page 47).

[92] *A Bitter Brew- Coffee Production, Deforestation, Soil Erosion and Water Contamination*. URL: https://ohiostate.pressbooks.pub/sciencebites/chapter/a-bitter-brew-coffee-production-deforestation-soil-erosion-and-water-contamination (cited on page 47).

[93] *Introduction to hierarchical-state-machines*. URL: https://barrgroup.com/embedded-systems/how-to/introduction-hierarchical-state-machines (cited on page 47).

[94] *You don't need a library for state-machines*. URL: https://dev.to/davidkpiano/you-don-t-need-a-library-for-state-machines-k7h (cited on page 48).

[95] *Tinyfsm library*. URL: https://github.com/digint/tinyfsm (cited on page 48).

[96] *Boost state machine library*. URL: https://www.boost.org/doc/libs/?view=category_state (cited on page 48).

[97] *Python-statemachine library*. URL: https://pypi.org/project/python-statemachine (cited on page 48).

[98] *Pytransitions library*. URL: https://github.com/pytransitions/transitions (cited on page 48).

[99] *SMACC2*. URL: https://github.com/robosoft-ai/SMACC2 (cited on pages 48, 52).

[100] *Yasmin*. URL: https://github.com/uleroboticsgroup/yasmin (cited on page 48).

[101] *SMACC, hierarchical states*. URL: https://smacc.dev/hierarchical-states (cited on page 48).

[102] *SMACC, Orthogonals*. URL: https://smacc.dev/orthogonals (cited on page 48).

[103] *The blackboard model in BTs*. URL: https://docs.cryengine.com/display/CEPROG/Behavior+Tree+Blackboard (cited on page 50).

[104] *Behavior Trees in robotics and AI, an introduction*. URL: https://www.researchgate.net/publication/319463746_Behavior_Trees_in_Robotics_and_AI_An_Introduction (cited on page 50).

[105] *Behavior Trees CPP library*. URL: https://www.behaviortree.dev (cited on pages 51, 54).

[106] *Pytrees library*. URL: https://py-trees.readthedocs.io/en/devel/index.html (cited on page 51).

[107] *Navigation2 Behavior tree package*. URL: https://github.com/ros-planning/navigation2/tree/main/nav2_behavior_tree/include/nav2_behavior_tree (cited on page 51).

[108] *Wikipedia page for "Automated planning and scheduling"*. URL: https://en.wikipedia.org/wiki/Automated_planning_and_scheduling (cited on page 52).

[109] *The FlexBE framework"*. URL: https://github.com/FlexBE/flexbe_behavior_engine (cited on page 52).

[110] *The Skiros2 framework"*. URL: https://github.com/RVMI/skiros2/wiki (cited on pages 52, 53).

[111] Francesco Rovida, Bjarne Grossmann, and Volker Kruger. 'Extended behavior trees for quick definition of flexible robotic tasks'. In: (2017). DOI: 10.1109/IROS.2017.8206598 (cited on page 52).

[112] *Planning.Wiki - The AI Planning and PDDL Wiki"*. URL: https://planning.wiki/ref/pddl (cited on page 53).

[113] *The Plansys2 framework"*. URL: https://plansys2.github.io/index.html (cited on page 54).

[114] *The ROSPlan framework"*. URL: https://kcl-planning.github.io/ROSPlan (cited on page 54).

[115] Francisco Martín et al. 'PlanSys2: A Planning System Framework for ROS2'. In: (2021) (cited on pages 54, 55).

[116] *SMACC vs behavior-trees*. URL: https://smacc.dev/smacc-vs-behavior-trees (cited on page 57).

[117] *Wikipedia page for control system*. URL: https://en.wikipedia.org/wiki/Control_system (cited on page 58).

[118] *Wikipedia page for proportional control*. URL: https://en.wikipedia.org/wiki/Proportional_control (cited on page 59).

[119] *Wikipedia page for Real-time computing*. URL: https://en.wikipedia.org/wiki/Real-time_computing (cited on page 64).

[120] *Ethercat.org*. URL: https://www.ethercat.org/en/technology.html (cited on page 64).

[121] *Real-time programming demos*. URL: https://docs.ros.org/en/rolling/Tutorials/Demos/Real-Time-Programming.html (cited on page 65).

[122] *Real-time work group guides*. URL: https://ros-realtime.github.io/Guides/guides.html (cited on page 65).

[123] *The real-time tools package*. URL: https://github.com/ros-controls/realtime_tools (cited on page 66).

[124] *ROS2 control home page*. URL: https://control.ros.org/master/index.html (cited on page 67).

[125] *ROS2 control demos*. URL: https://github.com/ros-controls/ros2_control_demos#diffbot (cited on page 67).

[126] *Wikipedia page for PID controller*. URL: https://en.wikipedia.org/wiki/PID_controller (cited on page 67).

[127] *Nav2 plugins*. URL: https://navigation.ros.org/plugins/index.html#plugins (cited on page 70).

[128] *MoveIt2 documentation*. URL: https://moveit.picknik.ai/humble/index.html (cited on page 71).

[129] *Wikipedia page for unit testing*. URL: https://en.wikipedia.org/wiki/Unit_testing (cited on page 75).

[130] *Gtests*. URL: https://github.com/google/googletest (cited on page 75).

[131] *Pytest*. URL: https://docs.pytest.org/en/7.1.x (cited on pages 75, 81).

[132] *Gmock for dummies*. URL: https://google.github.io/googletest/gmock_for_dummies.html (cited on page 75).

[133] *Wikipedia page for dependency injection*. URL: https://en.wikipedia.org/wiki/Dependency_injection (cited on page 77).

[134] *GMock cook book*. URL: https://google.github.io/googletest/gmock_cook_book.html (cited on page 77).

[135] *ROS2 coding guidelines*. URL: https://docs.ros.org/en/rolling/The-ROS2-Project/Contributing/Code-Style-Language-Versions.html (cited on page 80).

[136] *Colcon documentation*. URL: https://colcon.readthedocs.io/en/released/user/how-to.html (cited on page 80).

[137] *Autoware contribution guidelines, testing*. URL: https://autowarefoundation.gitlab.io/autoware.auto/AutowareAuto/contributor-guidelines.html#contributors-guidelines-run-tests (cited on page 80).

[138] *Unittest*. URL: https://docs.python.org/3/library/unittest.html (cited on page 81).

[139] *Rclpy test*. URL: https://github.com/ros2/rclpy/tree/master/rclpy/test (cited on page 82).

[140] *Rosbag2*. URL: https://github.com/ros2/rosbag2 (cited on page 83).

[141] *Launch testing*. URL: https://github.com/ros2/launch/tree/master/launch_testing (cited on page 83).

[142] *Launch Pytest*. URL: https://index.ros.org/p/launch_pytest (cited on page 84).

[143] *Domain ID concepts*. URL: https://docs.ros.org/en/rolling/Concepts/About-Domain-ID.html (cited on page 87).

[144] *Ament cmake ROS package*. URL: https://github.com/ros2/ament_cmake_ros/blob/rolling/ament_cmake_ros (cited on page 87).

[145] *Simulation tutorials*. URL: https://docs.ros.org/en/rolling/Tutorials/Advanced/Simulators/Simulation-Main.html (cited on page 88).

[146] *Introducing turtlesim*. URL: https://docs.ros.org/en/rolling/Tutorials/Beginner-CLI-Tools/Introducing-Turtlesim/Introducing-Turtlesim.html (cited on page 89).

[147] *Gazebo classic simulator, ROS2 overview*. URL: https://classic.gazebosim.org/tutorials?tut=ros2_overview (cited on page 89).

[148] *Gazebo ignition simulator*. URL: https://gazebosim.org/home (cited on page 89).

[149] *Webots*. URL: http://www.cyberbotics.com (cited on page 89).

[150] *CoppeliaSim*. URL: https://www.coppeliarobotics.com/coppeliaSim (cited on page 89).

[151] *Unity*. URL: https://unity.com (cited on page 89).

[152] *Ros2 for Unity*. URL: https://github.com/RobotecAI/ros2-for-unity (cited on page 89).

[153] *SVL simulator*. URL: https://www.svlsimulator.com (cited on page 89).

[154] *Nvidia Omniverse*. URL: https://docs.omniverse.nvidia.com (cited on page 89).

[155] *Isaac ROS bridge*. URL: https://docs.omniverse.nvidia.com/app_isaacsim/app_isaacsim/ext_omni_isaac_ros_bridge.html (cited on page 89).

[156] *Omniverse system requirements*. URL: https://docs.omniverse.nvidia.com/app_view_deprecated/app_view_deprecated/requirements.html (cited on page 89).

[157] *Flatland*. URL: https://github.com/avidbots/flatland (cited on page 90).

[158] *Map simulator*. URL: https://github.com/oKermorgant/map_simulator (cited on page 90).

[159] *ROS2 Gazebo simulation for the UR robots*. URL: https://github.com/UniversalRobots/Universal_Robots_ROS2_Gazebo_Simulation (cited on page 92).

[160] *Iiwa ros2 package*. URL: https://github.com/ICube-Robotics/iiwa_ros2 (cited on page 92).

[161] *Understanding ROS2 parameters*. URL: https://docs.ros.org/en/rolling/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Parameters/Understanding-ROS2-Parameters.html (cited on page 93).

[162] *ROS parameters server*. URL: http://wiki.ros.org/Parameter%5C%20Server (cited on page 93).

[163] *The AsynchParametersClient class*. URL: https://docs.ros2.org/galactic/api/rclcpp/classrclcpp_1_1AsyncParametersClient.html (cited on page 94).

[164] *Monitoring for parameter changes*. URL: http://docs.ros.org/en/rolling/Tutorials/Intermediate/Monitoring-For-Parameter-Changes-CPP.html (cited on page 95).

[165] *Dynamic reconfigure package*. URL: http://wiki.ros.org/dynamic_reconfigure (cited on page 96).

[166] *Yaml cpp library*. URL: https://github.com/jbeder/yaml-cpp (cited on page 101).

[167] *PyYaml library*. URL: https://pyyaml.org (cited on page 101).

[168] *Persistent parameter server*. URL: https://github.com/fujitatomoya/ros2_persist_parameter_server (cited on page 101).

[169] *About logging and logger configuration*. URL: https://docs.ros.org/en/rolling/Concepts/About-Logging.html (cited on page 103).