1 2 9 0

UNIVERSIDADE Ð COIMBRA

João Pedro Chaves Castilho

ROS 2.0 – STUDY AND EVALUATION OF ROS 2 IN COMPARISON WITH ROS 1

Master's Dissertation in Electrical and Computer Engineering, in the field of Robotics, Control and Artificial Intelligence supervised by Professor Doctor Rui Paulo Pinto da Rocha, co-supervised by Doctor David Bina Siassipour Portugal and presented to the Department of Electrical and Computer Engineering of Faculty of Sciences and Technology of University of Coimbra.

September of 2022

· U C ·

FCTUC FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE DE COIMBRA

ROS 2.0 – Study and Evaluation of ROS 2 in comparison with ROS 1

João Pedro Chaves Castilho

Coimbra, September 2022

· U C ·

ROS 2.0 – Study and Evaluation of ROS 2 in comparison with ROS 1

Supervisor:

Prof. Doutor Rui Paulo Pinto da Rocha

Co-Supervisor:

Doutor David Bina Siassipour Portugal

Dissertation submitted in partial fulfillment for the degree of Master of Science in Electrical and Computer Engineering.

Coimbra, September 2022

Acknowledgements

I would like to thank my supervisor, Professor Rui Rocha, who provided me with the best guidance I could have asked, for all the ideas and advices given throughout the development of this work, for providing all the necessary material so I would feel comfortable in the Laboratory and, most importantly, for the vast knowledge I acquired while working with him.

To my co-supervisor, Dr. David Portugal, who was always ready to help me with any problem or doubt. Thank you for showing me where to direct my work when I seemed lost and did not know what was the best path to take. I would also like to thank you for giving me the opportunity to participate as a teaching assistant in the 2022 edition of Robotics and Machine Learning Summer Course.

I would also like to thank my family for their unconditional support and for always motivating me to successfully complete this journey. A special thank you to my mother and father for reminding me that I can accomplish anything I set my mind to.

To my girlfriend, Leonor, who was always there for me when I needed the most and for reminding me to take the necessary breaks. Without your love and care this work would not be possible.

Finally, I would like to thank my friends, *Eletrões*, for their support, their friendship, and for all the night-outs and dinners we have had these past five years. A special thanks to my housemates Duarte, Isidro, Gonçalo and Diogo for the friendship we built and for all the moments we had together. I will cherish those moments forever.

Resumo

O Robot Operating System (ROS) é um middleware robótico de código aberto bem estabelecido utilizado para a prototipagem rápida de aplicações robóticas. No entanto, o ROS tem alguns pontos fracos, tais como a falta de suporte para sistemas de tempo real e limitações significativas em sistemas de múltiplos robôs. Para resolver estes problemas, o ROS sofreu uma importante atualização e, em 2015, foi lançada a primeira versão alfa do ROS 2.

A principal diferença no ROS 2 em relação ao ROS 1, é que a necessidade de um nó central, *ROS Master*, já não está presente. Isto porque o ROS 2 utiliza Data Distribution Service (DDS) como a principal camada de comunicação entre processos. Como o suporte a longo prazo do ROS 1 chegará ao fim em maio de 2025, é mais importante do que nunca analisar e explorar as características do ROS 2.

Este trabalho concentra-se na investigação do desempenho do ROS 2 em comparação com o ROS 1, com foco em sistemas multi-robô (MRS). Para este fim, realizámos primeiro um estudo dirigido à comunidade ROS para compreender as suas necessidades, determinar o nível de adoção de ROS 2, e identificar o que está a impedir a comunidade de migrar as suas aplicações de ROS 1 para ROS 2.

Subsequentemente, foi migrado para ROS 2 um software multi-robô desenvolvido em ROS 1 e são apresentadas neste estudo considerações importantes acerca desta migração. Finalmente, para avaliar ambas as versões do ROS, foram realizadas experiências em ambiente de simulação de forma a avaliar a eficiência de comunicação e utilização de recursos computacionais. Os resultados demonstram um desempenho promissor para ROS 2 em termos de escalabilidade no número de robôs e eficiência de comunicação.

Palavras-Chave: ROS 1, ROS 2, Data Distribution Service, Sistemas Multi-Robô, Inquérito a Utilizadores, Latência, Consumo de recursos computacionais.

Abstract

The Robot Operating System (ROS) is a well-established open-source robotics middleware used for rapid prototyping of robotic applications. However, ROS has several weaknesses, such as lack of support for real-time systems and significant limitations when working with multiple robots. To address this issue, ROS underwent a major update and the first alpha version of ROS 2 was released in 2015.

The main difference with ROS 2 is that the need for a central node, *ROS Master*, is no longer present. This is because ROS 2 uses Data Distribution Service (DDS) as the main communication layer between processes. As long-term support for ROS 1 will come to an end on May 2025, it is more important than ever to analyze and explore the features of ROS 2.

This work focuses on studying the performance of ROS 2 compared to ROS 1 with emphasis on multi-robot systems (MRS). To this end, we first conduct a user study targeting the ROS community to understand their needs with respect to ROS 2, determine the level of adoption of ROS 2, and identify what is holding the community back from migrating their ROS 1 applications to ROS 2.

Subsequently, a ROS 1 multi-robot simulation software was migrated to ROS 2 and guidelines and considerations important to such a migration are given. Finally, experiments were conducted in a simulation environment to evaluate both versions in terms of communication efficiency and resource usage. The results showed promising performance of ROS 2 in terms of scalability and communication efficiency.

Keywords: ROS 1, ROS 2, Data Distribution Service, Multi-Robot Systems, User Study, Latency, Resource Usage.

"I have not failed, but found 1000 ways to not make a light bulb."

— Thomas A. Edison

Contents

List of Acronyms

3D	Three Dimensional
API	Application Programming Interface
CPU	Central Processing Unit
DDS	Data Distribution Service
IP	Internet Protocol
HTTP	Hypertext Transfer Protocol
MAC	Media Access Control
MRS	Multi-Robot Systems
OS	Operating System
RAM	Random Access Memory
ROS	Robot Operating System
RPC	Remote Procedure Call
SLAM	Simultaneous Localization and Mapping
TCP	Transmission Control Protocol
URDF	Unified Robot Description Format
VPN	Virtual Private Network
XML	Extensible Markup Language

List of Figures

List of Tables

1 Introduction

Robots are very complex systems, often distributed through several machines and consisting of many hardware components (sensors and actuators) and software modules (controllers). All these components that make up a robot, need to work together in a seamless way to perform a specific task, such as exploring an unknown environment, autonomous navigation, human assistance, and more. As such, the need for frameworks that facilitate the development of robotic software from the ground up has always been a constant requirement in robotics research. Several robotic research groups have worked on developing such frameworks. Some end products from these efforts include robotics middlewares such as: MIRO [1], CARMEN [2], ORCA [3], Player [4], MOOS [5] and ROS [6, 7]. Most of these are discontinued, and the most relevant is ROS, which is the main focus of this dissertation.

The Robot Operating System [6, 7], ROS for short, was first introduced in 2007 by a team of engineers at Willow Garage who were working at the time with the PR2 robot. ROS first challenge was to tackle the immense difficulty robotics engineers and researchers faced when having to write software for robotic applications. For instance, not only was it needed for a developer to write a proper path planning algorithm, one also had to write low level drivers for sensor interfacing, a computer vision algorithm, and other features like perception and reasoning. Thus, one had to spend time developing components that had most likely been developed in the past by other researchers.

As the ROS community grew increasingly[1], new use cases have emerged to meet everyone's needs. The new use cases have focused on teams of multiple robots, support for small embedded platforms, support for real-time control directly in ROS, production quality, working on non-ideal networks, and prescribed design patterns for building and structuring systems, such as life cycle management. As such, API changes have been made to the code of ROS core and the first alpha version of ROS 2 [9] was released on August 2015. Two years later, the first official distribution of ROS 2 was released on December 2017, named *Ardent Apalone*. The current stable version of ROS 1, *Noetic*, is set to be the last version of ROS 1

[1]reaching a total of 600,660 packages on `https://index.ros.org` and, 40,203 users on `https://answers.ros.org/` according to the 2021 ROS Metrics report [8]

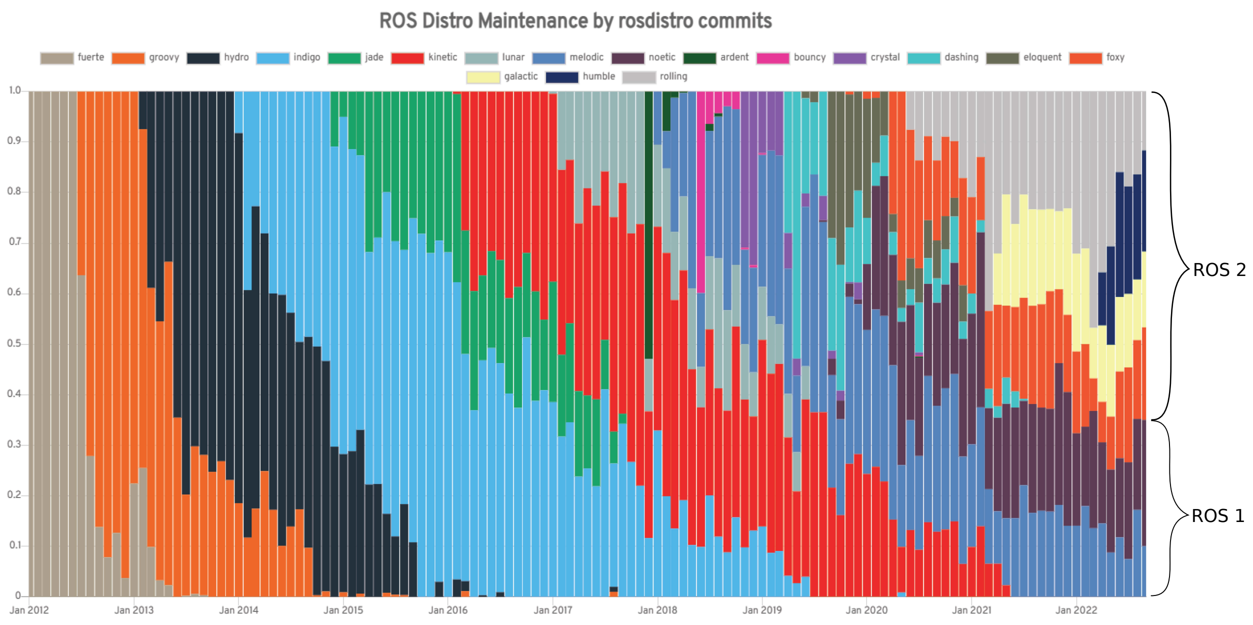


Figure 1.1: Relative maintenance of ROS distributions by commits to `https://github.com/ros/rosdistro`. Taken from `https://metrics.ros.org/`

[10] with support ending on May 2025 [11]. Therefore, as the support for ROS 1 is ending, there is a need to evaluate the current state of ROS 2, compare it with the widely used ROS 1, learn how to migrate packages from ROS 1 to ROS 2 and how to build new packages with ROS 2.

A clear indication of the end of support for ROS 1 distributions can be seen in Figure 1.1. This figure shows us the relative number of commits[2] for each ROS distribution. It can be seen that the relative number of commits for ROS 1 distributions has decreased, while the relative number of commits for ROS 2 shows an upward trend with more than half the percentage, as of early 2021.

1.1 Main Aim and Contributions

With the above considerations in mind, the main goal of this dissertation is to, in a first stage, study the design of ROS 1 and ROS 2 in order to understand their differences and compare them not only from a technical point of view, but also from a usability and user-friendliness perspective. After consolidating this knowledge, a user study is conducted to understand how widely ROS 2 is currently used and how ROS 2 is viewed by the robotics community. Then, some ROS 2 use cases, such as ROS 2 for teams of robots are evaluated and tested with practical cases. Note that from those use cases mentioned above, this work

[2]A commit in version control systems is an action that adds the most recent source code changes to the repository, including them into the repository's head revision.

provides deeper attention to ROS 2 for Multi-Robot Systems (MRS). For this purpose, we turn to a MRS that the *Institute for Systems and Robotics* of University of Coimbra has been working for about a decade [12]. Since this system is implemented in ROS 1, an effort is going to be made to migrate the system to ROS 2. Then, both implementations will be compared in terms of efficiency and scalability. With this, we expect that this work takes a step towards the adoption of ROS 2 as the middleware of choice for the development of robotic applications.

The contributions of this dissertation are:

1. Evaluation of ROS 2 performance when compared with ROS 1 in related case studies;
2. Evaluation of the process of migrating ROS 1 code to ROS 2 compatible code;
3. Evaluation of ROS 2 for developing robotic solutions for real-world applications;
4. Realization and post-evaluation of the a User Study, asserting the needs of the ROS community regarding ROS 2, the current state of its adoption, and what is holding back adoption;
5. Simulation and evaluation of ROS 2 on a MRS application

To the best of the author's knowledge, there is no previous academic work that evaluates ROS 2 in comparison with ROS 1, in terms of scalability and efficiency, in the scope of MRS.

1.2 Document Outline

The outline of the document is as follows. Chapter 2 studies the design of ROS 1 and ROS 2 and identifies their differences. Current efforts and limitations in supporting MRS in ROS 1 is also introduced in this chapter, as well as an analysis of related work made on the evaluation of ROS 1 and ROS 2.

In Chapter 3, the "ROS 2 Adoption" User Study is presented and analyzed in order to understand the necessities of the ROS community regarding ROS 2 and its new features.

In Chapter 4, a look into the steps required to migrate a ROS 1 package to ROS 2 is taken including important considerations when doing so. Specifically, the *patrolling_sim* package [13], a multi-robot patrolling software developed in ROS, is first migrated from ROS 1 Noetic to ROS 2 Galactic.

In Chapter 5, a detailed comparison is made between the ROS 2 version and the ROS 1 version, of the *patrolling_sim* package, in terms of efficiency and scalability.

Finally, the last chapter summarizes the work of this dissertation and draws some conclusions. Suggestions for where to steer future research is also given in the last chapter.

2 Background and Related Work

The Robot Operating System (ROS) is defined as a framework/middleware for robotics development. It is thought of as a meta operating system (OS) because it provides all the services that a common OS provides, such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing and package management. All these features make ROS a safe choice when developing robotics software, as it provides a fast and stable prototyping infrastructure [6].

Before introducing ROS, it is important to clarify and explain some concepts underlying the existence of ROS. One of these concepts is the publish-subscribe design pattern.

2.1 Publish-Subscribe Design Pattern

The publish-subscribe design pattern, first described in [14], is a messaging pattern in software architecture where processes exchange messages asynchronously, which means that none of the processes participating in the exchange block while waiting for new messages. In this architecture, there are two types of processes: the publishers who publish messages, and the subscribers who receive those messages. The uniqueness of this design pattern is that publishers are not aware about what subscribers they send the messages to, and there can even be no subscribers that receive the messages. Publishers send their messages in a categorized way, and subscribers can subscribe to those categories and receive the messages. As such, a subscriber can be interested in receiving data of a category that no other process is publishing to.

There are two main methods to exchange messages. One is called *topic-based* where messages are published to topics - they can be thought as channels - and subscribers receive every message that is published to the topic that they are subscribed to. This approach is similar to the one used by the ROS middleware, and this specific implementation of the publish-subscribe communication pattern is studied later in this dissertation. The other approach to serialize messages of interest to a specific subscriber is called *content-based* and, as the name suggests, subscribers receive messages that contain attributes specifically defined

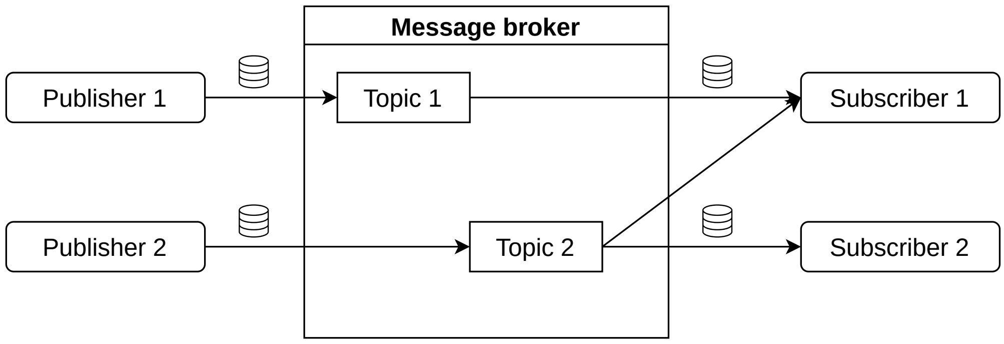


Figure 2.1: Publish - Subscribe topic-based communication.

by the subscriber.

Typically, these systems contain what is called a *message broker* who acts as a "middle man" for the messaging system. Publishers send the messages to the message broker and the subscribers register with the broker their subscription interests. The message broker then routes the messages from the publishers to the subscribers that manifest their interest.

The benefit of using a publish-subscribe design in a system is that it is easily scalable, and the processes are decoupled from one another, allowing one to change how a publisher behaves without impacting the entire system, thus making it increasingly flexible.

The ROS middleware also includes a publish-subscribe design implementation, which is discussed later in this dissertation when we examine ROS topics (Section 2.3.1).

2.2 The Client-Server Model

The client-server model [15] is also fundamental to revisit before diving into the ROS middleware. In a client-server architecture, there are two types of processes: the processes that provide a resource or service are called **servers**, and the processes that request those resources or services are called **clients**.

A client starts a communication with a server in order to request a service and the server awaits a communication request from the client and sends a response back to the client, thus making the exchange of messages synchronous. This message exchange requires both processes present in the exchange to agree on what to expect from the request and response. This is defined in the communication protocol.

A form of client-server interaction that is important to address is a **remote procedure call (RPC)** [16]. Just like the architecture described above, a RPC consists of a computer program requesting a service or resource from a different program located in a different address space in a shared network. This request is coded as if it were a normal local procedure call. The requester does not need to know the network's details to execute the remote

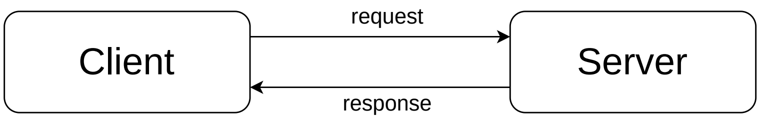


Figure 2.2: Client-Server model.

procedure call.

A specific type of RPC, called XMLRPC [17], is essential to the ROS middleware. It uses XML to encode its calls and HTTP as a transport protocol. A typical XMLRPC interaction between a client and a server proceeds as follows:

- A client makes a procedure call using the XMLRPC client; the call includes a method name, parameters, and a target server.
- The client packages the method name and parameters in XML format and issues an HTTP POST request to the target server.
- An HTTP server on the server side receives the HTML POST request and the XML content is passed to an XMLRPC listener.
- The XMLRPC listener parses the XML to get the method name and parameters and calls the appropriate method.
- The method is executed and returns a response to the XMLRPC process that packages the response in XML format.
- The HTML server sends the HTTP POST request to the HTML server in the client side.
- The XMLRPC client parses the XML to extract the returned value and passes the value to the client.

Client-server models are widely used in computer applications like email and the World Wide Web. An implementation of the client-server model is also present in ROS, and it is addressed later in this dissertation when we look into ROS services (Section 2.3).

2.3 Robot Operating System

Despite its name, ROS does not replace an operating system, it usually runs on top of UNIX-based operating systems, and it is centered on 4 principal concepts - nodes, topics, services, and master - all of which will be described in the following subsections.

In the remainder of this chapter, ROS 1 will be introduced and some core concepts will be explained.

2.3.1 ROS Computation Graph

Nodes

A typical ROS system is composed of several **nodes** [18], which can be thought of as processes with ROS capabilities that perform computations. For example, a ROS system integrated in a robot that is doing a navigation task might have a node that controls the motors of the wheels, a node to run the driver of the laser scanner and publish its data, a node that implements the path planning algorithm, and many more depending on the robot and the tasks itself. ROS nodes do not have to be running all in the same machine; some nodes can be running on different computers connected to the same network. For example, a service robot is typically composed of several onboard computers that perform lighter tasks like reading the sensors and controlling actuators, all connected via Ethernet, and it can also have some other nodes that compute more demanding tasks like Deep Learning models located in a high-power off-board machine, as illustrated in Fig. 2.3.

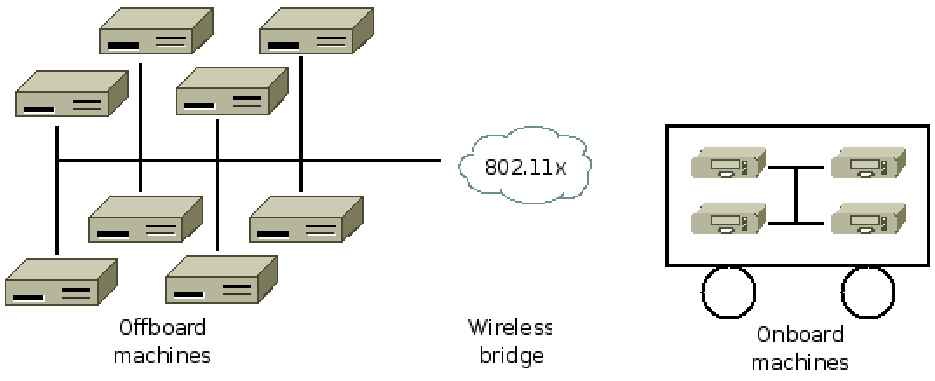


Figure 2.3: A typical ROS network configuration. Taken from [6].

ROS supports two main programming languages for its nodes: Python through the *rospy* [19] client library, and C++ through the *roscpp* [20] client library.

ROS nodes communicate with each other by passing **messages** and using a communication model, either using **topics** or **services**, thus abiding to the asynchronous publish-subscribe paradigm or to a synchronous client-server model, respectively.

Topics

ROS topics [21] are named buses over which ROS nodes communicate. Nodes can publish messages to topics or subscribe to topics. Communication using topics is nothing more than a publish-subscribe architecture such as the one introduced in Section 2.1.

Topics are used for unidirectional streaming communication, the data flow only occurs from the publisher nodes to the subscriber nodes, not vice versa. Each topic can only contain

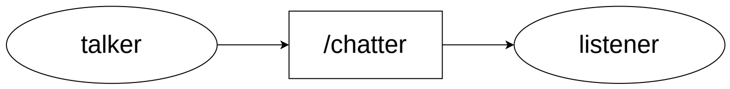


Figure 2.4: ROS nodes and topic of a simple ROS publisher-subscriber system. The node talker is publishing messages to the /chatter topic and the node listener subscribed to this topic.

one message type that is tied to the messages that are being published to it.

ROS supports two transport protocols for topics: a TCP/IP based protocol, known as TCPROS [22] and a UDP-based protocol, named UDPROS [23]. The common protocol used by ROS is TCPROS [22], while UDPROS [23] is less used and also less developed.

Services

ROS services [24] are used when a node needs to execute a remote procedure call, (cf. Section 2.2). Services are defined in a *.srv* file and must contain a pair of messages, one for the request and one for the reply. The ROS node that "hosts" the server for the service, declares the service name with a string. The ROS node that requests the service sends the request message, defined in the corresponding *.srv* file, and waits for the response. It is important to note that when a client calls a service, it becomes blocked until the server either responds or fails its execution, meaning that it is a synchronous type of communication between processes.

Messages

A message is a simple data structure that can contain fields of different data types, e.g. integers, doubles, floats, booleans, arrays, and strings. Message files are created with the extension *.msg* and are stored in a corresponding subdirectory, named *msg*. Messages can even contain other ROS messages in its data structure, creating a nested message variable.

Bag files

Bag files [25], are ROS files that store ROS message data. With tools like *rosbag*, all the messages published to one or more topics can be recorded along timestamps and then played back at runtime to the same topics, or even remapped to other topics. There is no difference between playing back a ROS bag file or running a node that publishes on the recorded topics. Therefore, bag files are the ROS way to record, save and play datasets.

Bag files are typically used to reproduce the environment recorded by an experimental setup or a simulation in a ROS-compatible simulator like Gazebo [26] or CoppeliaSim [27]. This way, different approaches can be tested with the same input data without having to run an infield testing session every time.

Master

The *ROS Master* [28] is an essential component of ROS 1. The *ROS Master* is what enables nodes to find each other and communicate. The *ROS Master* works with an XMLRPC-based API used by ROS client libraries to store and retrieve information about what nodes are running, what topics exist, what services are available, and more. The *ROS Master* has a Uniform Resource Identifier (URI), and it is stored in the ROS_MASTER_URI environmental variable. The XMLRPC server enables nodes in other machines to connect to this *ROS Master*, which makes the ROS system able to be distributed through different machines.

Establishing a topic connection between two nodes

To illustrate the role of the *ROS Master*, the following example enumerates the steps required to establish a topic-based communication between nodes.

1. The subscriber (listener node) starts and registers with the Master, informing it that it wants to subscribe to the topic */chatter*.
2. The publisher (talker node) starts and registers with the Master, informing it that it will publish a String message to the topic */chatter*.
3. The Master informs the subscriber of a new publisher publishing to the topic */chatter*.
4. The subscriber sends a connection request to the publisher.
5. The publisher returns the call with the selected protocol (i.e. TCPROS) and the TCP server for the data.
6. The subscriber proceeds with the connection to the TCP server given by the publisher and starts getting data.

2.3.2 ROS File system

Packages

ROS software is divided into packages [30]. A package can contain the necessary files that make sense to the usage of the package, such as ROS nodes source files, documentation files, configuration files, a *msg* directory to store message files, a *srv* directory to store *.srv* files, or a *bags* directory to store bag files. With the use of ROS packages, ROS software modules can easily be reused and distributed in an easy-to-consume manner.

To create packages, developers typically use tools, like *catkin_create_pkg* for the catkin build system of ROS 1, based on CMake.

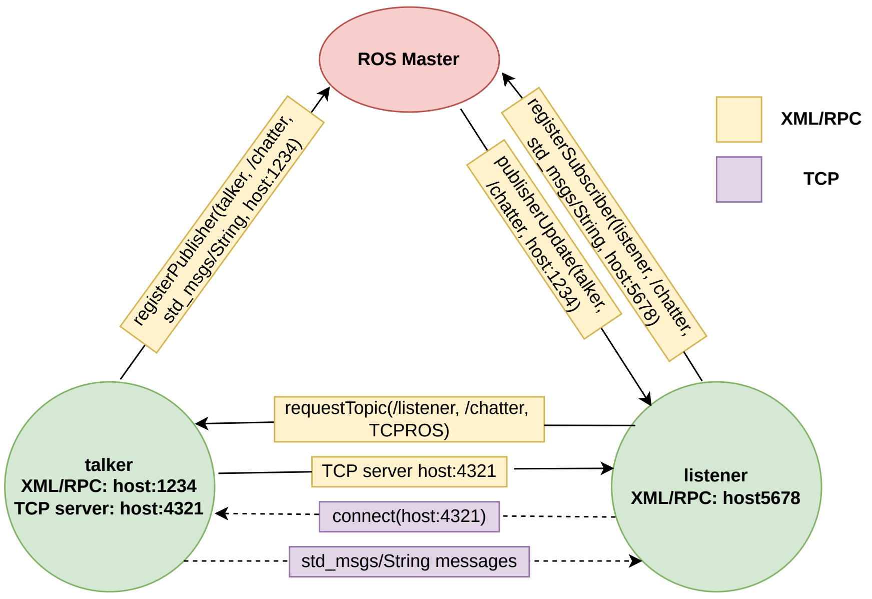

ROS Master
XML/RPC
TCP
registerPublisher(talker, /chatter, std_msgs/String, host:1234)
registerSubscriber(listener, /chatter, std_msgs/String, host:5678)
publisherUpdate(talker, /chatter, host:1234)
requestTopic(/listener, /chatter, TCPROS)
TCP server host:4321
connect(host:4321)
std_msgs/String messages
talker XML/RPC: host:1234 TCP server: host:4321
listener XML/RPC: host5678

Figure 2.5: Diagram with the steps of a topic connection. Adapted from [29].

CMakeLists.txt

The *CMakeLists.txt* is a file required by every ROS package used by the CMake build system. This file contains a set of instructions that describe how the package should be built, specifying the targets (executables and libraries) to be compiled and where to install them to.

Package manifest

Every package has a package manifest [31], which is a XML file named *package.xml*. This file defines the package name, version number, authors, maintainers, a description, licenses, and dependencies on other packages.

Launch files

Launch files are the solution to start many nodes that need to be run for a specific application. Launch files have a *.launch* extension and use a specific XML format. They are placed within a package directory, inside a *launch* folder.

2.4 Support for Multi-Robot Systems in ROS 1

As stated before, the *ROS Master* is a fundamental piece of the ROS 1 computational graph, meaning that no ROS 1 system can run without the *ROS Master* node being launched through the *roscore* command in one of the machines composing the system. Because ROS nodes could not lose connection to the *ROS Master* node, this creates a central point of failure and a bottleneck for all ROS applications, which is not suitable for Multi-Robot Systems (MRS) that need to be scalable, distributed and have no central point of failure.

With the interest in MRS, efforts have been made to make ROS support multi-master systems[1]. Multi-master systems are communicating systems that have two or more *ROS Masters* running simultaneously, each *ROS Master* in a different machine. This is useful for MRS since, if one has a *ROS Master* node running on every robot that is part of a MRS, the system has no central point of failure, thus becoming fault tolerant. Also, it allows us to minimize, or at least control, the amount of data exchanged within the network of robots. ROS does not support multi-master communication out-of-the-box, therefore, in the last several years, some packages have been developed that enable this type of communication.

One of these packages is *wifi_ comm*[2]. This package enables multi-master communication by mirroring the topics of interest in every ROS system. This means that, if a robot sends a message to a topic that is being mirrored to the other ROS systems, other robots that take part in the team, will also receive the published message by the sending robot. This solution is not scalable for large teams of robots, since it creates a lot of unnecessary traffic in the network for each robot. This package is based on *foreign_ relay*[3] to register the topics and services on the other ROS masters. It uses the *Optimized Link State Routing (OLSR)* in order to monitor which ROS masters connect and disconnect the network. This ROS package has been used in [32] to develop a multi-robot SLAM package.

Another solution used for MRS communication in ROS is a package called *tcp_ interface*[4]. This package uses a ROS node to translate ROS messages into strings sent by TCP to other ROS masters in the network with known IP address. Multiple robots (each with its own *ROS Master*) exchange messages via a topic, which is automatically converted to a string and transmitted over TCP using *tcp_ interface*. This package has been used in [33] with the Multi-Robot Patrolling package *patrolling_ sim*[5].

More recent efforts have been put into the development of what is considered the most versitile multi-master communication, the *multimaster_fkie*[6] ROS package [34]. With only two nodes, it is possible to establish and manage multi-master networks, requiring minimal configuration. The two nodes are: *master_ discovery* and *master_ sync*. The *master_ discovery* node is responsible for periodically sending multicast messages to notify about available ROS masters in the network. This node is also responsible for notifying other ROS masters when a change in the local *roscore* occurs. The *master_ sync* node is responsible for synchronizing

[1]wiki.ros.org/sig/Multimaster

[2]wiki.ros.org/wifi_comm

[3]wiki.ros.org/foreign_relay

[4]github.com/gennari/tcp_interface

[5]wiki.ros.org/patrolling_sim

[6]wiki.ros.org/multimaster_fkie

all the ROS masters in the network. This package has been used in the STOP R&D project, whose goal was to deploy a commercial security system of cooperative patrolling robots [35].

Since ROS has been developed from the ground up for a single robot use case, it is not optimized for MRS. The interest in MRS support in ROS is high, and the fact that there is no native support for MRS in ROS 1 certainly has significant shortcomings. Running multiple ROS masters leads to more computational load and more communication overhead on the whole network.

2.5 Robot Operating System 2

The ROS 2 API changes have been thought of with the objective of leveraging the positive points of ROS 1 and improving the negative ones [36]. One of the drawbacks in ROS 1 is not being able to natively support MRS and in a completely decentralized and fault-tolerant manner. This problem is on of the main focus of this dissertation. Other problems like not being natively secure [37], not supporting real-time systems nor small embedded platforms and performance drop on nonideal networks, are all downsides that the ROS 2 distribution aims to solve.

2.5.1 Technical differences between ROS 1 and ROS 2

We now look into the major differences between ROS 1 and ROS 2.

Platforms

Having ROS middleware supported on different platforms is highly desirable. Despite many large corporations opting to use Linux-based platforms like Ubuntu, a significant part of the industry also uses Windows as its developing platform.

Ubuntu [38] is the only Operating System officially supported by ROS 1. Windows is only supported through an experimental version, or through the Windows Subsystem for Linux [39]. ROS 2, on the other hand, is currently officially supported and tested on Ubuntu, Windows and macOS.

ROS 2 services

In ROS 1, services are synchronous, meaning that when a client requests upon a service server, that client waits until it receives the response, or the service execution fails. This type of interaction may be inappropriate in critical systems because it can cause a deadlock, meaning that the client waits for a long time for a response.

On the other hand, in ROS 2, services can be either synchronous or asynchronous. When a service client is asynchronous, it does not block when waiting for a server response, which prevents deadlocks. Synchronous services are only available in ROS 2 Python Client Library, *rclpy*.

ROS 2 Parameters

In ROS 1, the parameters are all hosted on the Parameter Server managed by the *ROS Master* node. This server is visible by all nodes in the system and every node can use this server to store and retrieve parameters in runtime.

In ROS 2, since there is no *ROS Master* node, each parameter is stored in the node itself. This implies that parameter naming also suffer changes. Each parameter has associated with it the name of the node that stores the parameter and the name of the parameter, such as, `/<node_name>/<parameter_name>`. The node is also responsible for accepting or denying changes to its parameters, and the lifetime of the parameter is entirely tied to the lifetime of the node.

ROS 2 actions

Actions had never been integrated in the core functionalities of the ROS graph before ROS 2. They were available through an external community-made package named *actionlib* [40]. Actions have been created to solve the need to have asynchronous service communications and get feedback from the server while the client is waiting for the response.

ROS 2 actions are now part of the ROS 2 client library, instead of being implemented in a separate library, as before. Another difference from ROS 1 actions is that, because ROS 1 services are a type of synchronous communication, ROS 1 actions had to be implemented on top of topics under a namespace taken from the action name, in order to make actions asynchronous. On the other hand, because ROS 2 services are asynchronous by default, ROS 2 actions use a combination of topics and services, as can be seen in Figure 2.6.

To sum it up, ROS actions can be thought as a group of services and topics used for long-running service-client interactions.

ROS 2 interface definition

The way one creates ROS messages, services, and actions in ROS 2 is the same as in ROS 1. Message definition files go inside a *msg* folder, services inside a *srv* folder and actions inside a *action* folder. The difference comes after compiling these files. In ROS 2, a namespace is inserted in the name of such interfaces to create separate namespaces for messages and

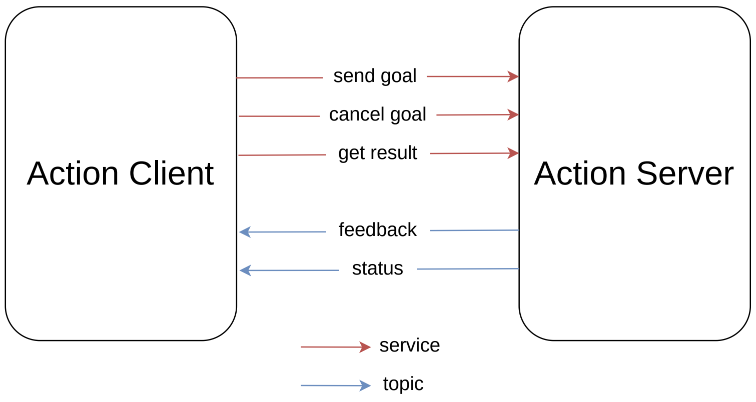


Figure 2.6: ROS 2 actions, topics and services. Taken from [41].

services defined in a package. For example, in ROS 1, after compilation we can have the following names:

- `my_package/MyMessage`
- `my_package/MyService`

In ROS 2, a namespace is inserted before the interface's actual name, as shown below:

- `my_package/msg/MyMessage`
- `my_package/srv/MyService`

ROS 2 client libraries

Client libraries are application programming interfaces (APIs) that allow developers to write their ROS code. With client libraries, developers have access to the fundamental pieces of the ROS Graph like nodes, topics, and services.

In ROS 1, the officially supported languages are C++ and Python. For the Python version, the supported versions are Python3 as of ROS Noetic, and Python2 for ROS Melodic and previous distributions.

ROS 2 uses more recent C++ standards, C++11 and C++14. As for Python support, ROS 2 only supports versions greater or equal to Python 3.5. The C++ ROS 2 client library is named *rclcpp* and the Python client library is named *rclpy*. There are other supported languages for ROS 2 maintained by the community like: Java (*rcljava*), Objective C (*rclobjc*), C# (*rclcs*), Node.js (*rclnodejs*), Ada (*rclada*), and Rust (*rclrs*).

The main difference between ROS 2 and ROS 1 client libraries is that ROS 1 client libraries like *roscpp* and *rospy* are both implemented from the ground up, meaning that they are completely independent, and they do not share any common code. In ROS 2, both *rclpy* and *rclcpp* utilize common functionality from the ROS 2 client library, or *rcl*. Client libraries in ROS 2 are built on top of *rcl* and each client library only needs to wrap the common functionalities of *rcl*, such as logic and behavior of ROS core concepts that are not programming language specific. The *rcl* is written in the C language because it is typically the easiest for other languages to wrap its foreigner functions interfaces.

This use of a common interface enables client libraries to be lighter and with less code to maintain. Another strong point is that, client libraries become more consistent. If any changes are made to the ROS core functionality in the *rcl* API, all client libraries will reflect the changes without having to fix anything that is language specific.

ROS 2 Middleware Interface and DDS

In ROS 1, communication concepts are built on top of custom protocols (e.g. TCPROS and UDPROS). In ROS 2, these communication concepts are built on top of the Data Distribution Service (DDS) [42], an existing and well-established middleware solution. DDS is defined as a middleware protocol and API standard for data-centric connectivity [43]. DDS, much like the ROS middleware, provides a reliable publish-subscribe transport protocol. Using DDS, ROS 2 can benefit from a different discovery system used by DDS, which completely replaces the *ROS Master* of ROS 1 in the ROS 2 computational graph. This makes ROS 2 applications completely distributed and fault tolerant, having no central point of failure. An additional benefit from adopting DDS is that ROS 2 supports various Quality of Service Policies, which provide more options when developing ROS 2 applications over different networks.

Because ROS 2 supports various DDS implementations, it includes the **ROS middleware interface**, which defines the API between the ROS client library and any specific DDS implementation. Below the ROS middleware interface, there is a middleware implementation that is DDS specific, meaning that different DDS vendors have their own middleware implementation layer. For example, eProsima's Fast DDS middleware uses *rmw_fastrtps_cpp* [44], while Eclipse Cyclone DDS uses *rmw_cyclonedds_cpp* [45], and RTI Connext uses *rmw_connext_cpp* [46]. As of ROS 2 Galactic, the default middleware implementation is Eclipse's Cyclone DDS.

Before being sent to the DDS implementation, the middleware layer transforms the ROS messages arriving from the client library into a unique data format, specific to the DDS implementation. The opposite situation is also implemented at the middleware layer, where DDS messages are transformed into ROS data objects and then delivered to the ROS 2 client library.

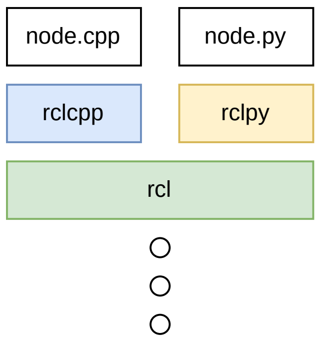


Figure 2.7: ROS 2 client libraries diagram. Adapted from [47].

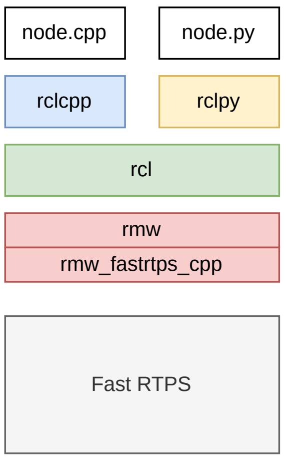


Figure 2.8: ROS 2 architecture using Fast RTPS as the DDS implementation. Adapted from [47].

Quality of Service Policies

Having DDS as the underlying middleware between different nodes enables ROS 2 developers to tune the communication between nodes with different Quality of Service (QoS) policies. With carefully picked QoS policies, developers can have best-effort communication like UDP, or reliable transport like TCP, or every point in between. More information on ROS 2 QoS policies and their compatibility can be found in Appendix A.

ROS 2 CLI tools

The use of command line interface (CLI) tools is extremely useful for developing and running ROS application. All the CLI tools important to ROS 1 migrated to ROS 2, but with some changes. In ROS 1, the commands followed the following structure: **ros[command] [argument]**. In ROS 2, there is a subtle change: **ros2 [command] [argument]**.

Table 2.1: Some examples of ROS CLI commands in ROS 1 and ROS 2.

ROS 1	**ROS 2**	**ROS 1**	**ROS 2**
rosnode	ros2 node	roslaunch	ros2 launch
rostopic	ros2 topic	rosrun	ros2 run
rosmsg	ros2 interface	rosbag	ros2 bag
rosparam	ros2 param	rospkg	ros2 pkg
rosservice	ros2 service	roswtf	ros2 doctor (ros2 wtf)
not implemented	ros2 action	not implemented	ros2 component
not implemented	ros2 lifecycle	not implemented	ros2 daemon
not implemented	ros2 multicast	not implemented	ros2 security

ROS 2 package creation and build system

The current build system used in ROS 1 distributions is *catkin*, which is a combination of CMake macros and Python scripts. ROS 1 developers can use *catkin* to create new packages and to build the catkin workspace. ROS 2 uses a different build tool called: *colcon*. Colcon is considered an universal build tool because it can be used to build and test multiple software packages, including different build systems. The decision behind using *colcon* is that when ROS 2 was released, new features for *catkin* were needed such as, supporting additional package types (pure python packages), supporting Windows or supporting building ROS 1 packages. The ROS 2 equivalent of *catkin_make* is *colcon build*. To create a ROS 2 package, one should use: *ros2 pkg create [package_name]*. When using this command, one can use the **–build_type** argument to specify what build system is going to be utilized for the created package. Currently, ROS 2 supports three build types: *cmake*, which uses standard CMake, *ament_cmake*, which is similar to plain CMake but provides several helping functions to make it easier to build C++ ROS 2 packages, and *ament_python*, that is used to build Python ROS 2 packages.

Table 2.2 summarizes the files needed for C++ and Python packages.

Table 2.2: ROS 2 Python and C++ generated files by ros2 pkg create.

	Python	**C++**	**Description**
package.xml	✓	✓	meta information about the package
setup.py	✓	×	instructions how to install package
setup.cfg	✓	×	python package executables
/<package_name>	✓	×	directory used by ROS 2 tools to find your package
CMakeLists.txt	×	✓	how to build the code in the package

ROS 2 Launch files

Another major change from ROS 1 to ROS 2 is the addition of Python-based launch files. In ROS 1, launch files are exclusively written in XML. In ROS 2, launch files can be written in Python, XML, or YAML. The addition of Python launch files became an improvement in the ROS Graph because it allows embedding more complex logic and routines.

A simple ROS 2 Python launch file is presented in Listing 2.1:

Listing 2.1: Python launch file.

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
  return LaunchDescription([
    Node(
          package='turtlesim',
          executable='turtlesim_node',
          name = 'turtle1',
          namespace = 'my_turtle',
          parameters=[
            {"background_r": 255}
          ],
          remappings=[
            ("/my_turtle/turtle1/cmd_vel", "/cmd_vel"),
          ]
    )
  ])
```

In this Python launch file, we launch a node from the package *turtlesim*. The node executable is *turtlesim_node*, and we change its node name to *turtle1*. We also create a namespace for the node, */my_turtle*, set a parameter *background_r* and remap the */my_turtle/turtle1/cmd_vel* topic to */cmd_vel*.

When developing a ROS 2 package, one can still use XML format launch files, however one should be aware that some XML tags changed [7]. A ROS 2 XML launch file can be view in Listing 2.2:

Listing 2.2: XML launch file.

```xml
<launch>
  <node pkg="turtlesim" exec="turtlesim_node" name="turtle1"
   ↪  namespace="my_turtle">
    <param name="background_r" value="255"/>
    <remap from="/my_turtle/turtle1/cmd_vel" to="/cmd_vel"/>
  </node>
</launch>
```

An example of the same launch file, but written in YAML format, can be view in Listing 2.3:

[7] https://docs.ros.org/en/foxy/How-To-Guides/Launch-files-migration-guide.html#migrating-tags-from-ros-1-to-ros-2

Listing 2.3: YAML launch file.

```yaml
launch:
- node:
    pkg: "turtlesim"
    exec: "turtlesim_node"
    name: "turlte1"
    namespace: "my_turtle"

    param:
    -
      name: "background_r"
      value: 255

    remap:
    -
        from: "/my_turtle/turtle1/cmd_vel"
        to: "/cmd_vel"
```

ROS 2 Bag Files

ROS 2 bag files, changed their file extension from *.bag* to *.db3*, which is associated with *sqlite3*. Because of this, ROS 1 bag files do not work out-of-the-box in a ROS 2 system. There is, however, a way to convert ROS 1 bag files to ROS 2 compatible bag files using the community made *rosbags* pure Python library [8].

Lifecycle nodes

ROS 2 introduces a new concept that does not exist in ROS 1, which is **lifecycle nodes**. Lifecycle nodes work just like a standard ROS node, but with more options on what to execute in each node state. Lifecycle nodes are scoped within a state machine of a finite amount of states (see Figure 2.9). A lifecycle node has 4 states, which are: **Unconfigured**, **Inactive**, **Active**, and **Finalized**. There are also 5 transition states, or intermediate: **configuring**, **activating**, **deactivating**, **cleaningup**, and **shuttingdown**. To change states, 5 state transitions are available: **configure**, **activate**, **deactivate**, **cleanup**, and **shutdown**. These state transitions are available through callback functions of the a *LifecycleNode* class, and in these callbacks the behavior of the node in different state transitions can be changed.

This change gives the ROS developers more options when developing a node. For example, if we want to develop a node for a sensor, we might want to allocate memory for the publishers and subscribers, and start the communication, and only after that, start publishing the data. With the help of *Lifecycle* nodes these stages are clearly separated.

[8] `https://gitlab.com/ternaris/rosbags`

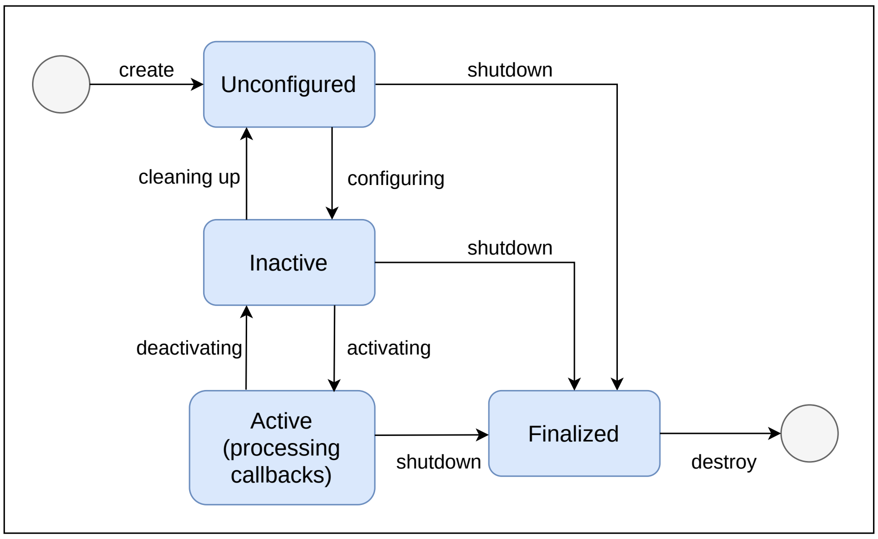


Figure 2.9: ROS2 lifecycle management state machine. Adapted from [48]

2.6 Support for Multi-Robot Systems in ROS 2

As stated before, one of the main changes from ROS 1 to ROS 2 is the absence of the *ROS Master* node invoked by the *roscore* call. This is one of the major changes in ROS 2, and therefore MRS are inherently supported in ROS through seamless distributed communication. With this change, multi-master communication, which was a workaround used in ROS 1 to deploy truly distributed MRS, is no longer needed in ROS 2.

2.7 ROS 2 performance studies

Since the creation of ROS 2, relevant academic work has been done exploring the capabilities of ROS 2 and also comparing it with ROS 1.

In [49], published in 2016, the authors evaluate the performance of ROS 2, focusing mainly on DDS and various DDS vendors. This work reports experiments that have been conducted to measure the latency, throughput, number of threads in use, and memory usage in the connection between a publisher node to a subscriber node. They have run experiments with ROS 1 communication (local and remote), ROS 2 communication (local and remote), and also communication between a ROS 1 node and a ROS 2 node using *ros1_bridge* package[9]. The authors also tested different DDS vendors, namely *Connext*, *OpenSplice* and *FastRTPS*. Authors found out that, when the ROS connection is local, end-to-end latencies vary according to message size. When the message size is lower than 64KB the latency is constant, however when the message size is higher than 64KB the end-to-end latency in-

[9]`https://github.com/ros2/ros1_bridge`

creases, when compared to ROS 1 local scenario. This is due to the message being divided into serveral packets. Also regarding the local case, the authors found that the *OpenSplice* implementation of DDS processes messages much faster than the other implementations, due to the high number of threads it utilizes. Therefore, the authors recommended using *OpenSplice* when working with local connections. For the remote case, the authors recommend using the *Connext* DDS implementation due to the higher throughput. For embedded systems, the recommended DDS implementation is *FastRTPS*, since it outperforms the other implementations in terms of thread and memory use.

In [50], researchers examine the security of ROS 2 and its impact on general performance. Experiments have been conducted on a wired and wireless network, with three different scenarios: using no security, using SROS2[10], which is a package that provides the tools and instructions required to secure DDS communications, and using a VPN. The study found that using a VPN to secure a ROS 2 system is better than using SROS2 in terms of latency and throughput. In both wireless and wired scenarios, it was found that using a VPN affects the latency of the system, but using SROS2 more significantly affects latency, sometimes almost doubling it when compared to the not secured system. However, it is to note that using a VPN is not desirable in most situations as it creates a central point of failure for communications. Therefore, for distributed applications, one can use SROS2 as it still provides secure communication.

In [51], the authors investigate the end-to-end latency of ROS 2 for distributed systems, and have found that when increasing the number of nodes in the system, from 3 to 23, the latency increased linearly. The authors also found that the largest contribution to latency is the delay between message notification and message retrieval by ROS 2.

Some previous works address real applications and use cases in ROS 2, specifically its real-time capabilities. In [52], a new software architecture for autonomous driving based on ROS 2 is presented and discussed. The researchers conclude that appropriate real-time behavior of a ROS 2 system is possible, but only if all special real-time coding requirements are taken into account in the implementation. In [53], real-time requirements are in a ROS 2 multiagent robot system. A comparison between ROS 1 and ROS 2 is made in a multi-node application, and the authors have verified that the nodes met real-time requirements in ROS 2, contrary to ROS 1. To further verify ROS 2 real-time capabilities, the authors implement a MRS composed by two robots, and verified that ROS 2 can meet the real-time constraints.

Despite ROS 2 being of an early age, there are already some relevant works on multi-robot systems using ROS 2. In [54] and [55], ROS 2 is used to develop multi-robot swarm applications. In both works, the authors refer that the main reason for using ROS 2 in swarm

[10]`https://github.com/ros2/sros2`

applications is the replacement of *ROS Master* by DDS, thus removing the need to run *ROS Master* and making the system run without a central point of failure. In [56], researchers developed a toolbox for distributed cooperative robotics based on ROS 2. The authors also refer in this paper that the absence of a *ROS Master* node is the main feature of ROS 2 that made them adopt ROS 2 as the robotics framework for this work.

In [9], the authors enumerate five use cases in which ROS 2 is used in commercial applications. In one of those five works, the authors explain how ROS 2 was used by the OTTO Motors company, a Clearpath Robotics spin-off that sells land research platforms in factory facilities. OTTO Motors reported that when using ROS 1, they were not able to deploy more than 25 robots on the same shared ROS 1 network. When OTTO changed from ROS 1 to ROS 2, they were able to deploy more than 100 robots in a single facility. OTTO claims that this boost in scalability was due to ROS 2 using DDS and therefore having a more scalable network topology.

In 2018 a questionnaire on the needs of the ROS 2 community was conducted by PickNik [57], a robotics company. The report consisted of 25 participants of several robotics companies and were divided into two groups: ROS users and non ROS users. The authors came to the conclusion that although most ROS 2 features were considered to be in a ready state, these have not been exhaustively tested in real scenarios, and that is something that concerned the respondents.

Overall, these recent studies have all found promising results in the performance of ROS 2. However, none of these research papers document the process of migrating a package to ROS 2, nor do they explicitly state the benefits associated with the migration for a ROS 1 package that has gone through the migration process. With this dissertation, we hope to provide the robotics community with more reasons to go through the migration process, by providing them with results from a wider user study about ROS 2 adoption showing what current ROS 2 users and non-users think about ROS 2, and also by describing a real use case of migrating a ROS 1 package to ROS 2, showing the performance improvement that can be expected when a package is migrated to ROS 2.

2.8 Summary

In this chapter, we started by looking into the publish-subscribe design pattern and the client-server model as the foundation for several important aspects regarding ROS, such as ROS Topics and ROS Services. Afterwards, we have revisited general ROS 1 concepts, e.g. nodes, topics, services, messages, the *ROS Master* node and the ROS File system, which are important to grasp before elaborating on ROS 2. A study has also been made regarding the current known methods for supporting MRS in ROS 1, especially regarding multi-master communication.

Finally, the main differences between ROS 1 and ROS 2 (see Table 2.3) were addressed, as well as new features of ROS 2 that are not available in ROS 1. This provides us with the necessary knowledge to proceed with the next chapter, in which we carefully design, implement and evaluate a ROS 2 Adoption User Study and draw useful conclusions from it.

Table 2.3: Summary of key differences between ROS 1 and ROS 2.

	ROS 1	ROS 2
Supported Operating Systems	Linux	Linux, Windows, and macOS
Network transport protocol	TCPROS/UDPROS	DDS (UDP)
Node discovery	*ROS Master*	DDS Dynamic Discovery (Peer-to-peer discovery)
Client libraries	Written independently	Client libraries are built on top of a common C library (rcl)
Launch files	XML	XML, Python, and YAML
Building packages	catkin_make	colcon_build
Parameters	Stored in Parameter Server (*ROS Master*)	Stored and managed by the node
Actions	Implemented with topics	Implemented with topic and services
Non ideal networks	Not possible because of the use of TCP/IP	Possible because of DDS and QoS Policies
Security	None	DDS security standard and SROS2
Manage Lifecycle of a node	None	Lifecycle Nodes

3 ROS 2.0 Adoption User Study

In order to have a broader perspective of the current adoption of ROS 2, a user study has been designed, targeted at ROS users and developers, both with ROS 2 experience and without any ROS 2 experience or knowledge.

This user study aims to understand what the current state of ROS 2 adoption is and what are the biggest challenges that prevent ROS developers from migrating their applications from ROS 1 to ROS 2.

The user study also aims to clarify what most excites developers about the new ROS 2 features and to what extent these novel features impact the development and deployment of MRS based on the ROS middleware.

The user study was made available publicly between the 16th of March and 8th of June 2022 and was hosted on Google Forms. The questions that were present in the User Study are included in Appendix B. The user study was posted on several national and international mailing lists (e.g. robotics-worldwide) and on forum websites such as the ROS Discourse, Reddit (r/ROS), Facebook ROS Developer groups, and ROS Discord server.

In total, 116 responses were collected. However, only 103 were considered valid answers. The criteria for validating an answer were: 1) the respondent having used ROS before; and 2) having correctly answered the screening question 'What is ROS?'.

It is noteworthy that, despite the existence of another user study made in 2018, already mentioned in Section 2.7, this user study [57] presented some limitations. Firstly, the mentioned questionnaire is already considered outdated, with 5 ROS 2 distributions having been released since its creation. So, it is useful to see if the ROS community has shifted its opinion on ROS 2 since then. Secondly, the small number of participants and low diversity of different backgrounds represented in the study are also limitations which our user study attempts to eliminate. All the participants from the mentioned report are from an industrial background, meaning that there was not a representation from the academic background ROS developers, which, as we will see, still represents the majority of the ROS community. Finally, the report from 2018 only slightly addresses the problem of migrating from ROS 1 to ROS 2 giving insight on what is holding back developers, leaving questions, such as, "Who is migrating to ROS 2?", "How difficult is the migration?" and "Why did you decide to migrate to ROS 2?", unanswered. Therefore, we consider our user study to be more broad and inclusive.

3.1 User Study results

In this section, we discuss the results from the user study. With this purpose, we go through each section of the user study, present the results, analyze them and draw conclusions from those results.

3.1.1 User Profile Characterization

In this section, the aim of the questions was to characterize the respondent profile in order to have some geographical information, his/her level of expertise in working with the ROS middleware, and also some information about in which context they use ROS.

The results for the questions from this section are presented in Figures 3.1 - 3.4.

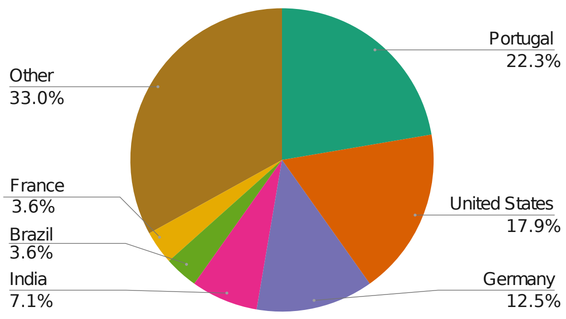


Figure 3.1: Results to question: What country are you from?

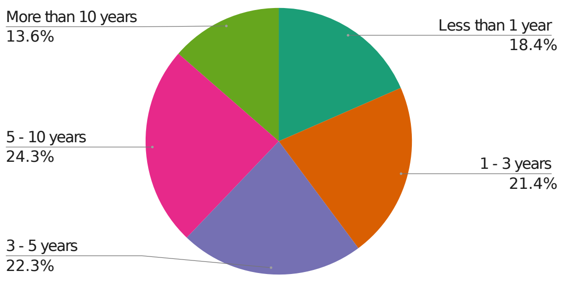


Figure 3.2: Results to question: For how many years have you been using ROS?

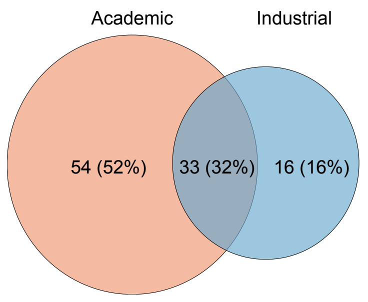


Figure 3.3: Results to question: What is the main context of the previous projects in which you have used ROS?

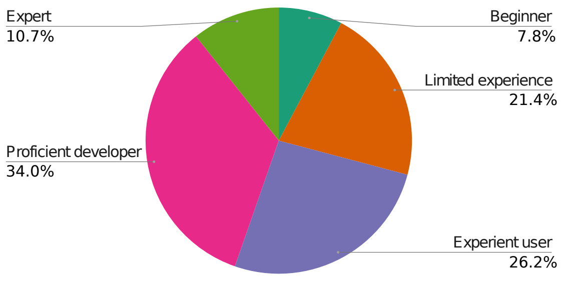


Figure 3.4: Results to question: What do you consider to be your level of proficiency with ROS?

As we can see from Figure 3.1 and Table C.1, the User Study reached worldwide adoption, with respondentes from 6 out of the 7 continents in the world. We can also see that ROS is mostly used in European countries and in the United States of America, which is expected since these countries are considered highly industrialized, thus the interest in robotics is also higher. An observation that has to be made about the geographical distribution of our data is that Portugal attained a high percentage of replies since this work was disseminated from in a Portuguese university, and the User Study was also distributed in internal communication channels.

We can also see from Figures 3.2 and 3.4 that the level of expertise of the respondents is balanced going from "beginner" up to "limited experience", "experient user", "proficient developers" and finally "expert". However, when we look at Figure 3.3, we can see that ROS is mostly used in academic projects with 52% of the respondents working exclusively in robot deployments for academic purposes. It is important to note that the answers to this question are not mutually exclusive, meaning that a respondent could choose both options if he/she had previously worked in both fields. Respondents who have only worked with ROS in an industrial setting are a minority (16%). However, since 48% of respondents have worked or are still working in an industrial setting, we can still consider that the industrial background is well represented in our data.

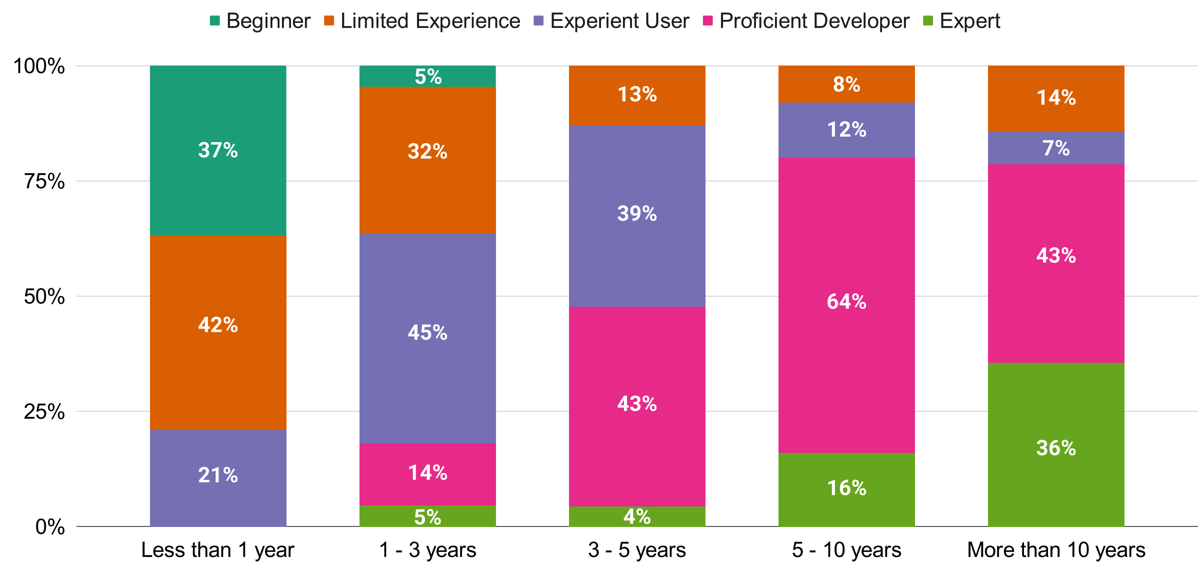


Figure 3.5: For how many years have respondents been using ROS according to their level of expertise

Figure 3.5 shows us for how many years have respondents been using ROS according to their level of expertise. It can be seen that, as expected, beginner users have higher percentages in groups of users who have not used for a long time (less than a year). On the other hand, proficient developers and expert users are more represented in the groups of

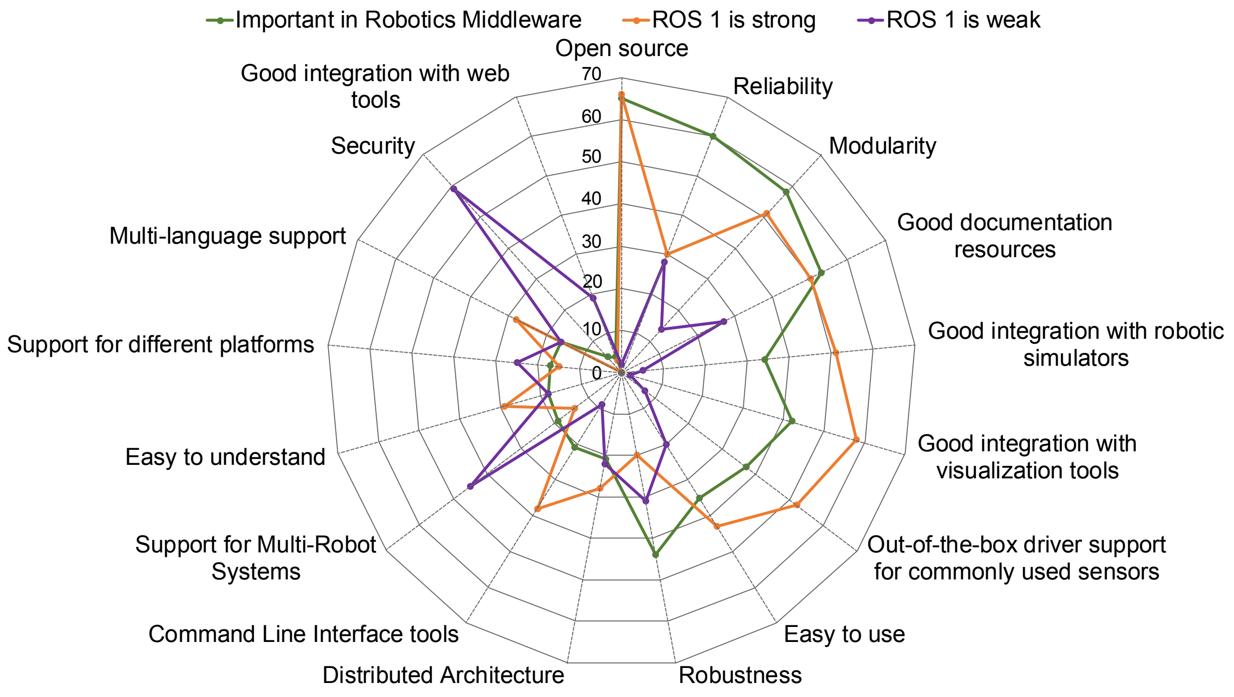


Figure 3.6: Number of answers for each feature to the question: "What are the most important features in a Robotics Middleware?", "What makes ROS 1 a good choice when developing robotics applications?" and "In what areas do you think ROS 1 is not very strong?"

respondent who have been using ROS for more than 5 years.

With these results, we can successfully categorize our respondents according to a geographic, background, and expertise level.

3.1.2 Robotics Middleware and ROS 1 strong/weak points

In this section, the objective is to extract information about what features users find important in a general-purpose Robotics Middleware and also, which of these features does ROS excel and, in contrast, is not very strong. Note that for answering these questions, respondents are assumed to have used ROS in the past.

The results relevant to this section are presented in Figures 3.6 and 3.7.

In Figure 3.6, we can point out some aspects that developers find important in Robotics Middleware that are also really strong in ROS 1, such as, being **open source**, **modular**, having **proper documentation resources**, and **integration with visualization tools**. On the other hand, there are also features that developers find important in Robotics Middleware, that are not very strong in ROS 1, namely, being **reliable** and **robust**.

Another aspect we can observe from Figure 3.6 is in what aspects does ROS 1 excel and that developers do not find that important in a Robotics Middleware. These are: having **support for commonly used sensors** and **integration with robotic simulators**. Both these features are essential when using ROS to deploy robots in real-world applications and in earlier stages of projects based on ROS. Therefore, these features can generally be considered to be in a mature state in ROS 1, while reliability and robustness can be improved on. Interestingly, these aspects are tackled by ROS 2, since it claims to be more robust and reliable than ROS 1 due to the use of the already tested and industry-proven DDS.

Another important observation from Figure 3.6 are the features where ROS 1 is considered weak. These features can be reduced to two, which are **security** and **support for Multi-Robot Systems**. Looking at security, we can see that it ranks surprisingly low on the importance for robotics middleware, perhaps because most of the respondents have an academic background and security is not a priority for academic projects. In addition, none of the respondents indicated that ROS 1 has strong security features. This is foreseeable since ROS 1 has no security implementation in its network stack and is generally not considered secure. This is also something that ROS 2 aims to combat with the help of **DDS Security Specification** and **SROS2**.

Support for multi-robot systems is something that ROS 1 struggles in. As already explained in Section 2.4, this is corroborated by many respondents who point out that support for multi-robot systems is weak. This is also a weakness that ROS 2 claims to fix with the use of DDS and its **peer-to-peer discovery system** for node discovery.

More results regarding this question are available in Table C.6, where we only take into consideration responses from respondents from the 3 highest expertise level (experient user, proficient developers and experts). The reasons behind making this distinction is because these respondents are naturally more capable of answering these questions than users with limited experience or beginners. These results are not much different from the ones presented above where we take all responses into consideration.

If we look at Figure 3.7[1], we can draw deeper conclusions into the state of ROS 1 documentation and tutorial resources. The chart scores each of the 7 aspects from 1 to 5, (1 being not at all and 5 being very much). It can be seen that ROS 1 documentation scored high in aspects such as informative, helpful, and not confusing, which means that **documentation resources are considered useful to ROS 1 developers**.

On the other hand, it did not score that well in the detailed and organized category, which seems to reveal that sometimes ROS developers find that the documentation available is not thorough and organized enough to fully understand some concepts regarding packages or ROS itself.

[1]The scores in Fig. 3.7 are the mean of every respondents' score for each category.

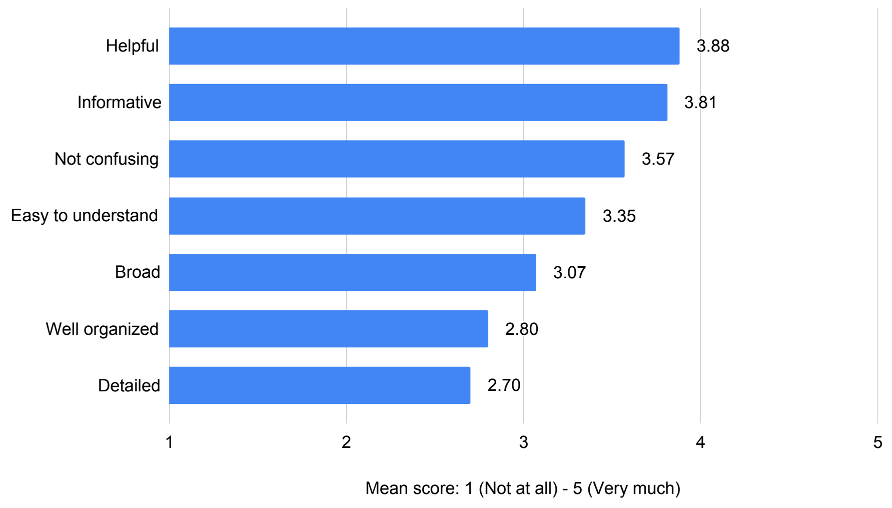


Figure 3.7: Mean scores for the question: How do you feel about ROS 1 Documentation and Tutorials?

3.1.3 ROS 2 Awareness

In this section, we can have a sense of whether ROS 2 is well marketed and what are the first impressions of the respondents about ROS 2. These are important questions, to realize if the apparent lack of ROS 2 adoption is due to users not knowing that it even exists or if it is due to other, more specific, problems.

If we look at Figure 3.8, we can see that 95.1% of the respondents have heard about ROS 2. This means that **ROS 2 is well-marketed in the ROS community** and that developers are at least aware about its existence. It should also be noted that of the five respondents who had never heard of ROS 2, only one had a level of ROS expertise higher than "Limited experience".

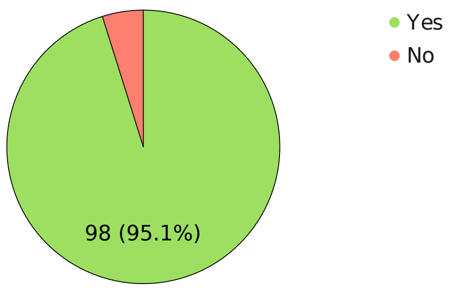


Figure 3.8: Results to question: Have you heard about ROS 2?

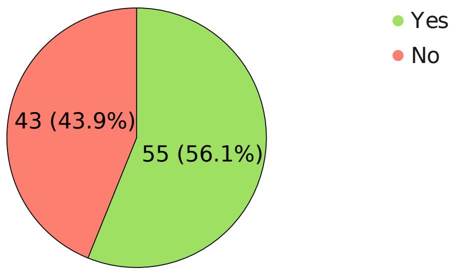


Figure 3.9: Results to question: Have you heard about any negative aspects of ROS 2?

From Figure 3.10 we can observe which ROS 2 features are the most well known. From the features that we found fitting to be part of the list, the ones that stand out from being the most popular are **use of DDS** and **absence of ROS Master**, both of which were selected in 78% of the responses. This is actually a strong point in favor of ROS 2, since, as we have discussed, one of the biggest flaws of the ROS 1 architecture is the fact that it relies on a central node (the ROS Master). Since ROS 2 does not have this issue, for developers, it seems to be a strong point in favor of its adoption.

In Figure 3.9, the results of the question "Have you heard about any negative aspects of ROS 2?" are presented. It can be seen that a large percentage (56.1%) of the respondents answered "yes", which requires an appropriate analysis. A chart with the specific issues developers heard about is presented in Figure 3.11. The most mentioned issue is the **lack of ROS 1 packages migrated to ROS 2**. Unfortunately, we have no way of knowing exactly the number of ROS 1 packages that have already been migrated to ROS 2, so we cannot further assess this issue.

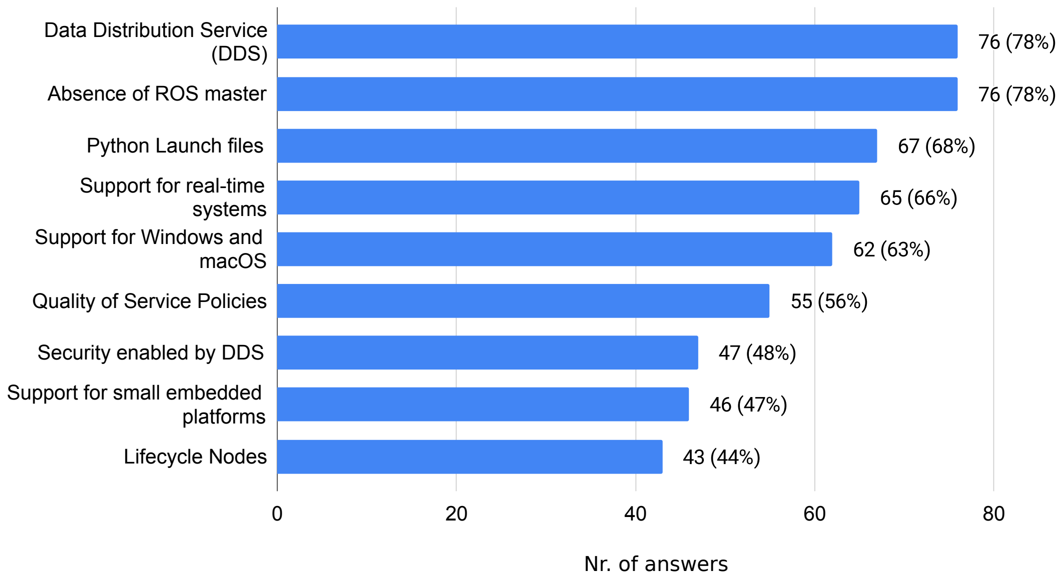


Figure 3.10: Number of answers to the question: Which of these ROS 2 features have you heard about before?.

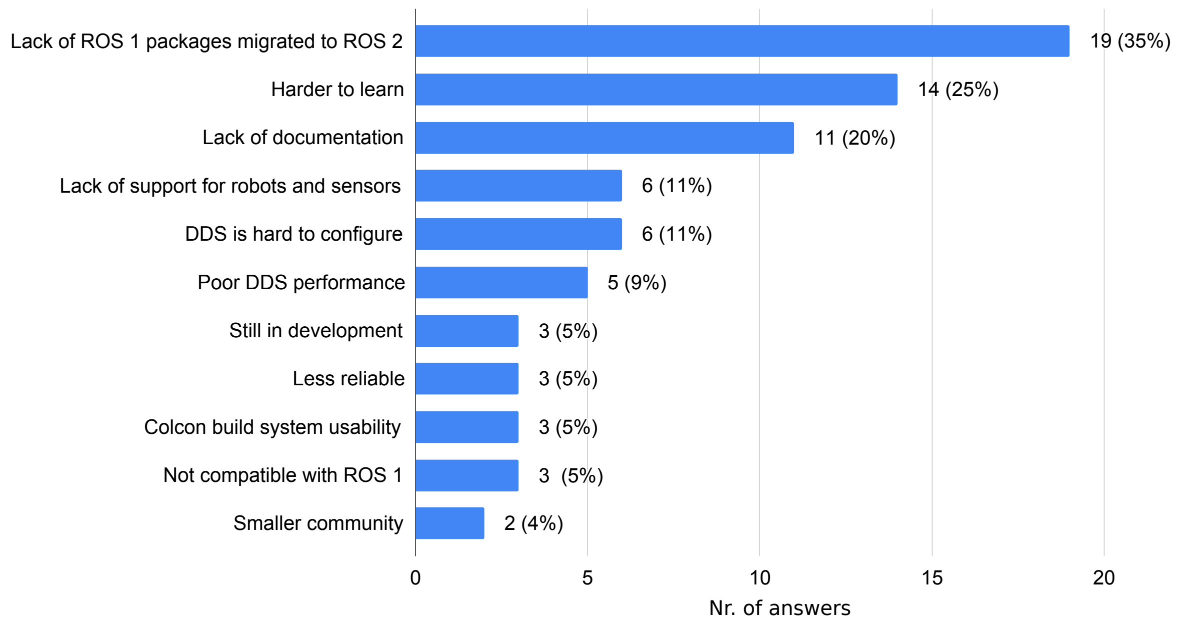


Figure 3.11: Number of answers to the question: Specify the negative aspects you heard about.

The second most mentioned issue is that ROS 2 is **more complex and harder to learn**. This tends to be true for a number of reasons. First, as already mentioned, ROS 2 relies on DDS, which, although more robust and reliable, is also more complex and difficult to configure. **Some respondents mentioned that often the default DDS settings are not suitable** and require significant tuning effort to work as intended for their specific robotics application. The fact that there are several implementations of DDS available is also another layer of complexity that the developer has to consider, since they do not have the same performance. We can conclude that this reduces to a problem that engineers typically face, which is: **one cannot add robustness, performance, and reliability without often adding more complex features and harder-to-understand concepts**.

Another feature that adds to the complexity of ROS 2 are **Python launch files**. Despite enabling more complex logic to launch the nodes of the system, that logic is not necessary in most cases, as mentioned by some respondents. Another fact is that the Python API for launch files is harder to learn than the XML launch files, especially because there is no documentation available for the Python API at the time of writing this dissertation. This is also closely related to the third most mentioned issue, which is **lack of documentation**.

3.1.4 ROS 2 Adoption

In this section, an analysis of the results from the ROS 2 Adoption section of the user study is made, so that we can understand what the current level of ROS 2 adoption is, what are the reasons behind users not using ROS 2, and also, if the user is currently using ROS 2,

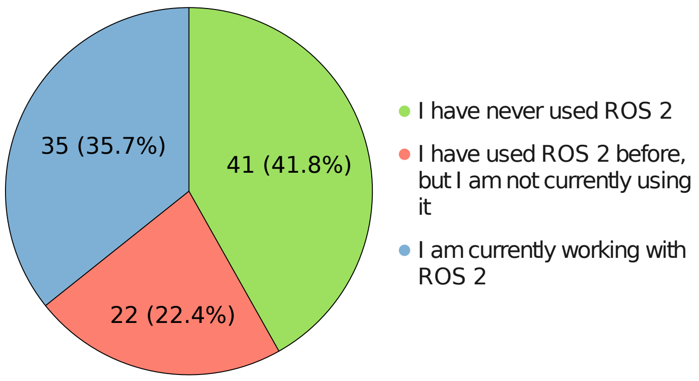


Figure 3.12: Results to the question: Have you ever used ROS 2?

and what the reasons are behind that option as well.

In analyzing these questions, only data from respondents who had heard of ROS 2 were considered. If we look into Figure 3.12, we can see that the majority of the respondents have already used ROS 2 (22.4%) or are currently using it (35.7%). However, there is still a large percentage of developers (41.8%) who have never attempted to use ROS 2.

Looking further into our data (Figure 3.13), we can say that the proficiency levels with the **highest percentage of current ROS 2 users** are **Proficient Developer** (45%) and **Expert** (36%). It should also be noted that, a minimal percentage of Expert users (9%) have never tried ROS 2, which shows interest by these users on what ROS 2 promises to offer. In contrast, developers **who have never used ROS 2** have higher percentages in the proficiency levels of **Beginner** (80%), **Limited Experience** (52%) and **Experient User** (58%). We can also see that expert users tend to be more hesitant to migrate than proficient developers, with 55% of expert users having tried ROS 2, but only 36% currently using it. On the other hand, 26% of proficient developers have tried ROS 2 and 46% are currently using it. This means that the percentage of Proficient developers who have tried ROS 2 and end up using it daily is higher than that of expert users. Based on these results, we tend to infer that ROS 2 is receiving more attention from experienced developers and that less experienced developers are sticking to ROS 1 to gain more experience before eventually deciding to adopt the more complex ROS 2.

In Figure 3.14, we can asses the state of ROS 2 adoption according to different backgrounds (academic and industrial). We can see that the ROS 2 adoption is happening faster in the industry, with 41% of developers with industrial background saying that they are currently using ROS 2. In contrast, the percentage of developers who have never tried ROS 2 is higher in academy, with 47%.

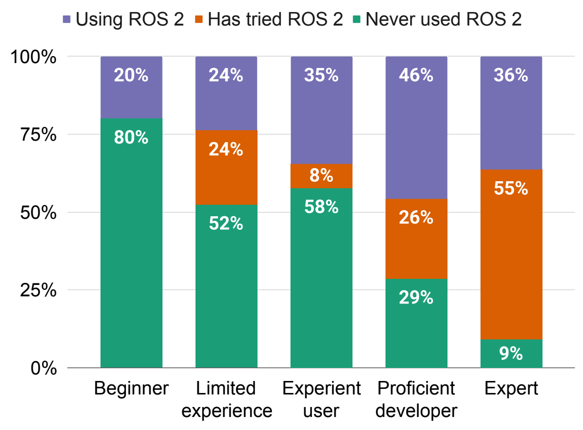


Figure 3.13: ROS 2 adoption related to developers' level of expertise in ROS.

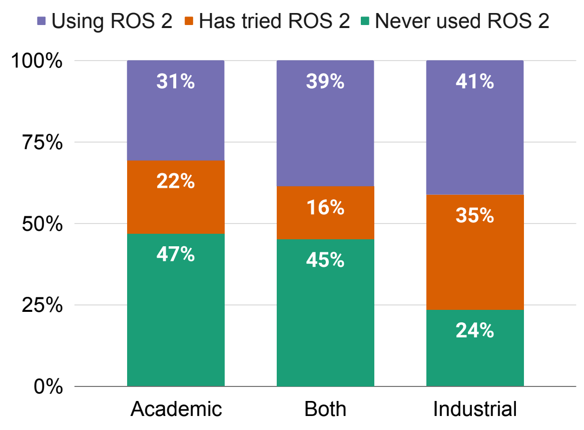


Figure 3.14: ROS 2 adoption related to developers' background.

In Figures 3.15 and 3.18, we can understand what is keeping ROS developers, who are not currently using ROS 2, from fully adopting ROS 2. Figure 3.15 makes a distinction between ROS developers who have never used ROS 2, and developers who have tried ROS 2. For developers who have never used ROS 2, we can see that the main reason for not migrating is that they **do not feel the urgency in migrating**. Interestingly, for developers who have already used ROS 2, the sense of urgency did not reach as high a percentage as in the previous case. For these developers, the main reasons for not fully migrating are related to **dependency on ROS 1 packages** and **lack of ROS 1 packages migrated to ROS 1**. Despite having lower percentages, the availability of ROS 1 packages in ROS 2 is also a reason stated by developers who never used ROS 2. We hypothesize that many of these developers might have considered using ROS 2, more specifically migrating their ROS 1 project to ROS 2, but may have been held back by a package dependency problem of a ROS 1 package that is still not available in ROS 2.

In Figure 3.16 the different reasons for not adopting ROS 2, with respect to different developers' background (academic and industrial) are presented. We can see that, in general, between developers with different background there is not much difference, meaning that the reasons to not adopt ROS 2 are not necessarily related to developers' background. Despite this, there is one result we would like to point out, which is that 0% of users with exclusively industrial backgrounds chose "Waiting for the end of life of ROS 1 Noetic" as a reason for not migrating to ROS 2. This means that in the industrial sector, the fact that support for ROS 1 is ending is not by itself a reason to migrate to ROS 2.

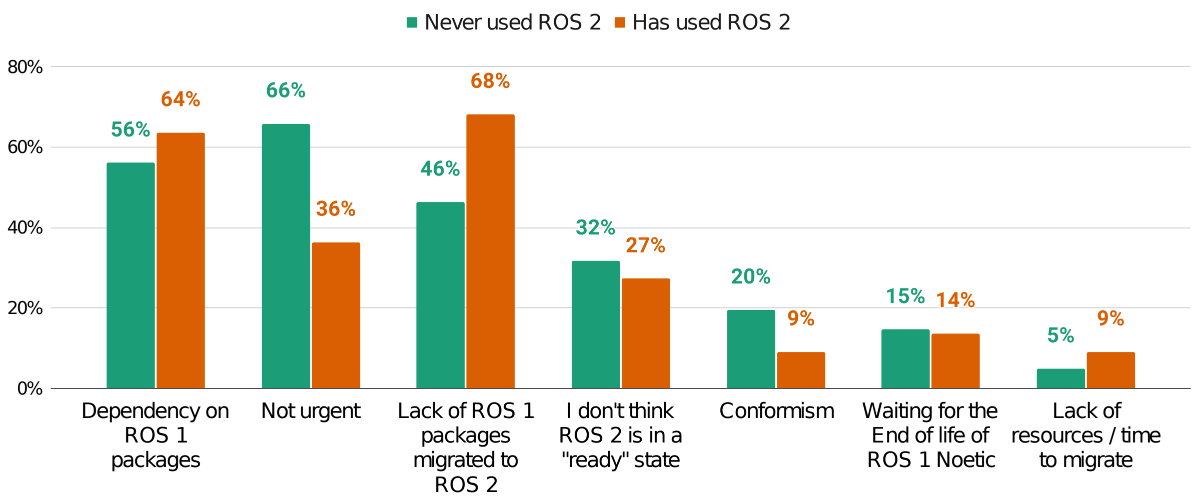


Figure 3.15: Percentage of answers to the question: What is keeping you from adopting ROS 2?

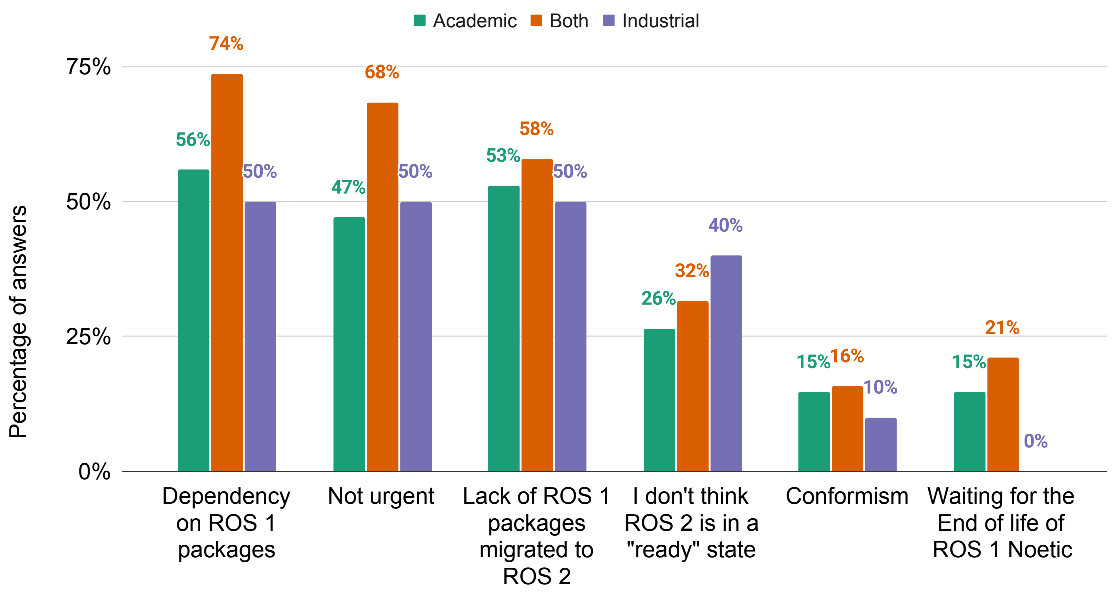


Figure 3.16: Percentage of answers to the question: What is keeping you from adopting ROS 2? (according to developers' background).

In Figure 3.17, we can observe whether developers with different levels of expertise have different reasons for not adopting ROS 2. It can be seen that the level of expertise generally does not have an influence on the reasons for not adopting ROS 2, except for the reason "I don't think ROS 2 is in a ready state", which had a noticeable high percentage of answers by expert developers. This might be because expert developers need concrete evidence to believe that it is worth to adopt ROS 2 before committing to it. Therefore the need to have convincing case studies of how ROS 2 improved a ROS 1 project is of extreme necessity.

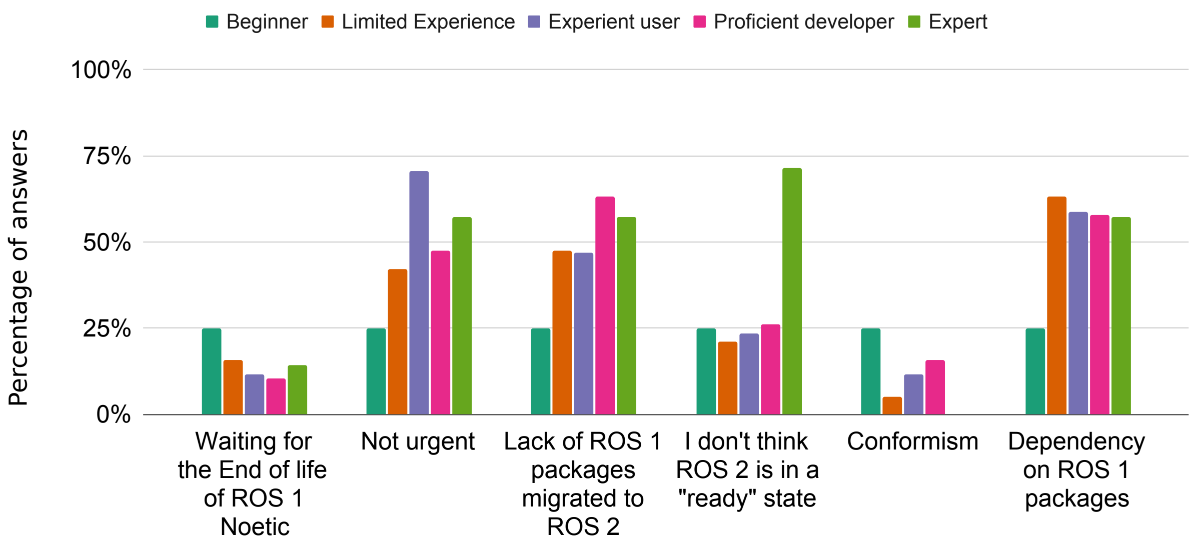


Figure 3.17: Percentage of answers to the question: What is keeping you from adopting ROS 2? (according to developers' expertise).

On the same note, when asked what keeps other developers from adopting ROS 2 (see Figure 3.18), the most common answers are related to **dependency on ROS 1 packages**.

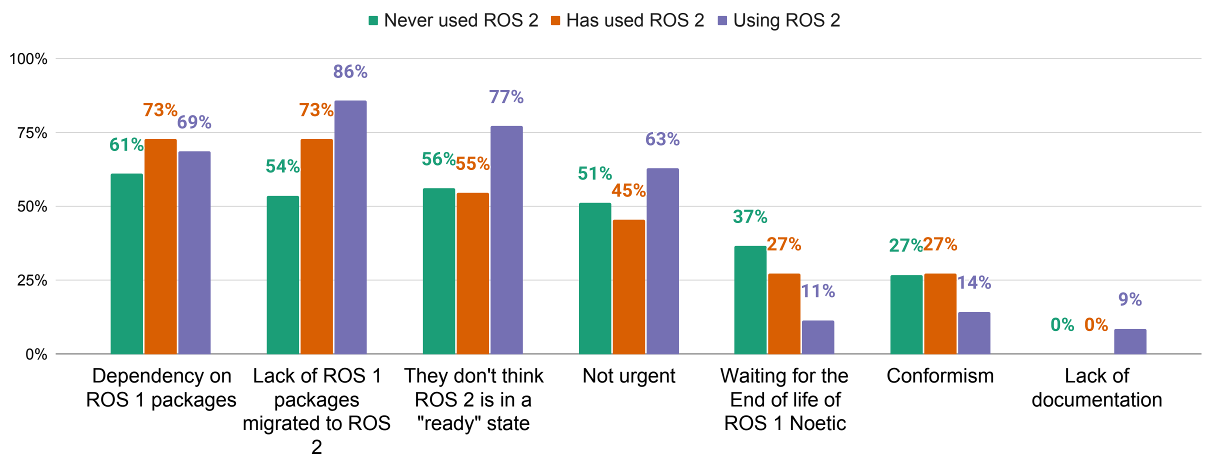


Figure 3.18: Percentage of answers to the question: "What do you think is keeping other developers from adopting ROS 2?"

In Figure 3.19, the main reasons as for why developers have adopted ROS 2 are presented. Two reasons seem to stand out. The first reason is that **ROS 2 is designed for production and has industry stakeholders behind its development**. The design of ROS 2 is more in line with what developers expect from a robotics middleware, as we have seen in Figure 3.6 and Table C.5, with a focus on reliability and robustness. Having large companies, such as Bosch, Amazon, Microsoft, Samsung, Sony, and others, be part of the Technical Steering

Committee (TSC), whose main objective is to steer the direction of ROS 2, are a strong reason to adopt ROS 2.

The second most mentioned reason is **the integration with DDS**. As seen from the answers to other questions, the use of DDS as the middleware of choice for ROS 2 seems to be popular amongst ROS developers and, as seen in Figure 3.19, it is one of the main reasons developers start using ROS 2. Also related to DDS is the third most mentioned reason, which is that **ROS 2 enables fully distributed systems**.

On the down side, there are some reasons that do not seem to impact adoption, such as **support for different Operating Systems** like Windows and macOS, which means that most developers still prefer to use Linux-based operating systems to work with ROS, in this case, ROS 2.

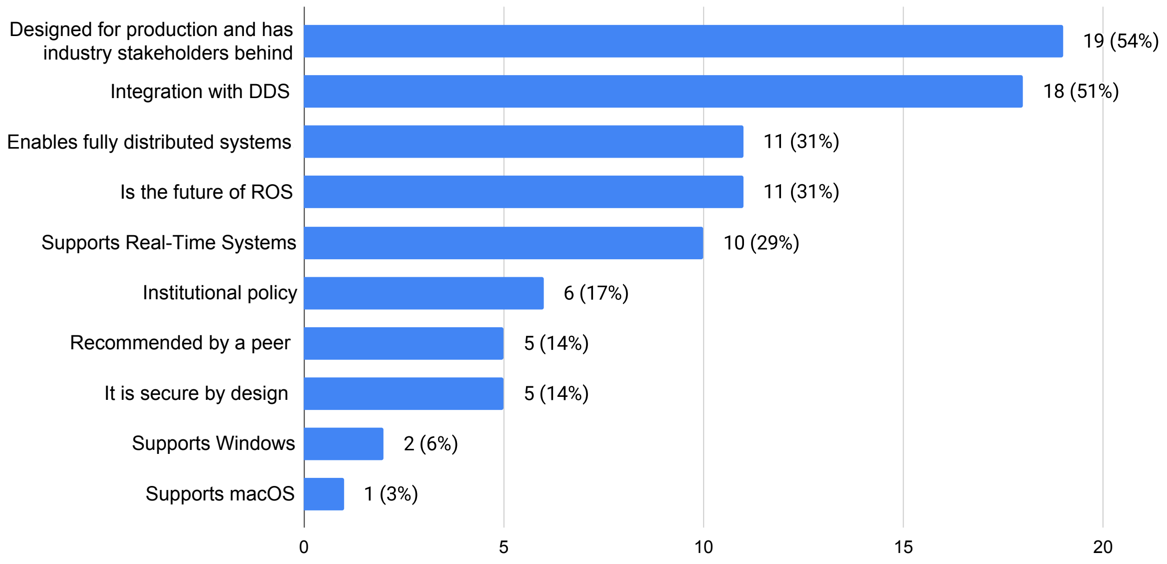


Figure 3.19: Results to question: "Why have you decided to fully adopt ROS 2?"

In Figure 3.20, we can inspect the reasons why developers from different backgrounds are adopting ROS 2. The reason most mentioned in Figure 3.19 holds for both academic and industrial backgrounds. However, we can see great differences between both groups in two reasons. The first is **integration with DDS**, which is mentioned by 62% of the industrial developers and by 40% of the academic developers. Secondly, the fact that **ROS 2 is the future of the ROS middleware** does not make developers from an industrial background particularly adopt ROS 2, only being mentioned by 15% of industrial developers, while 53% of the academic developers mentioned this as a strong reason for using ROS 2.

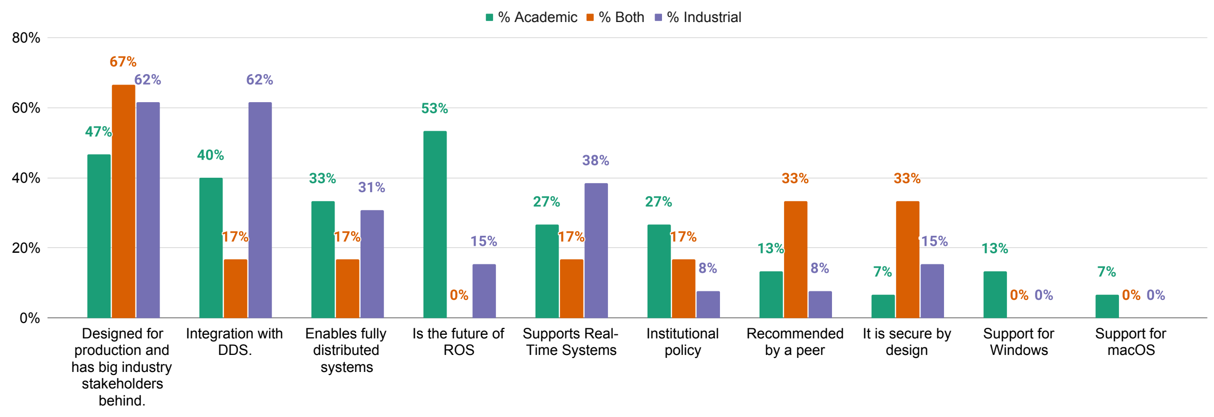


Figure 3.20: Percentage for each reason to adopt ROS 2 according to developers' background.

In Figure 3.21, we look into the current state of the documentation resources of ROS 2 when compared with ROS 1. We can see that, as in the case of ROS 1, the documentation resources are considered poor, achieving a mean score near 3 in almost all assessment criteria.

The documentation resources of ROS 2 received an inferior score when compared with its ROS 1 equivalent, meaning that ROS 1 documentation is considered to be in a more stable state than ROS 2 documentation, especially in categories such as "Not confusing" and "Broad". The lower score in the "Broad" category means that there are still some features of ROS 2, or even ROS 2 packages, that are missing documentation. This can be due to ROS 2 being of early age. The lower score in the "Not confusing" category means that the available documentation is not always clear. This is inherently connected to the fact that ROS 2 is more complex than ROS 1, so it should also be harder to write adequate documentation and tutorials.

Further discussion about usability in ROS 2 compared to ROS 1 can be raised if we look at Figure 3.22. We can see that ROS 1 beats ROS 2 in 4 out of 6 categories. In terms of learning curve and ease of development, by analyzing these results, we can see that ROS 1 has a smother learning curve than ROS 2 and is also easier to develop. This, as stated before can be explained due to ROS 2 being more complex and having a complex underlying middleware (DDS). This result is also in accordance with Figures. 3.7 and 3.21, since ROS 1 obtained superior results in terms of documentation and tutorials. The two categories in which ROS 2 is considered better than ROS 1 are **Features** and **Capabilities for production use**.

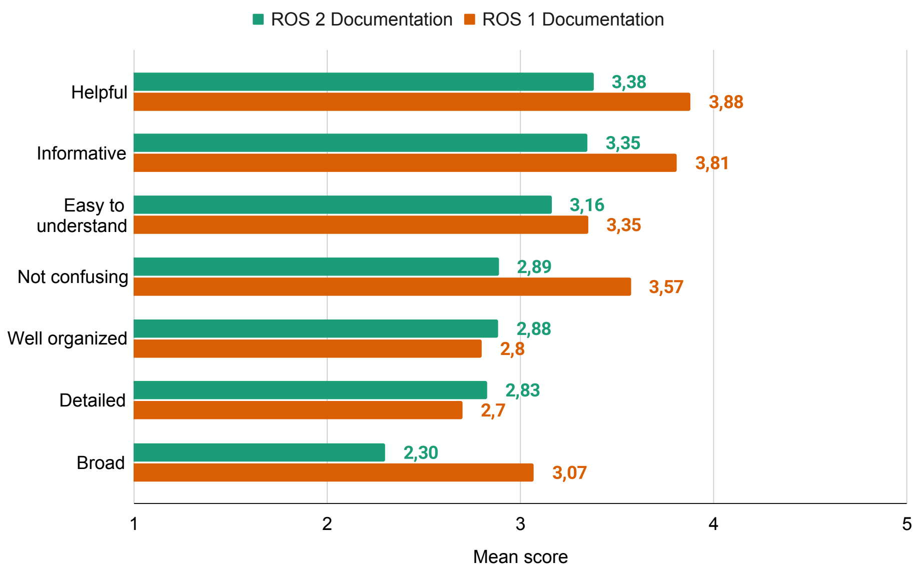


Figure 3.21: Mean scores for the question: "How do you feel about ROS 2 Documentation and Tutorials?" using a Likert scale ranging from 1 (bad) to 5 (good).

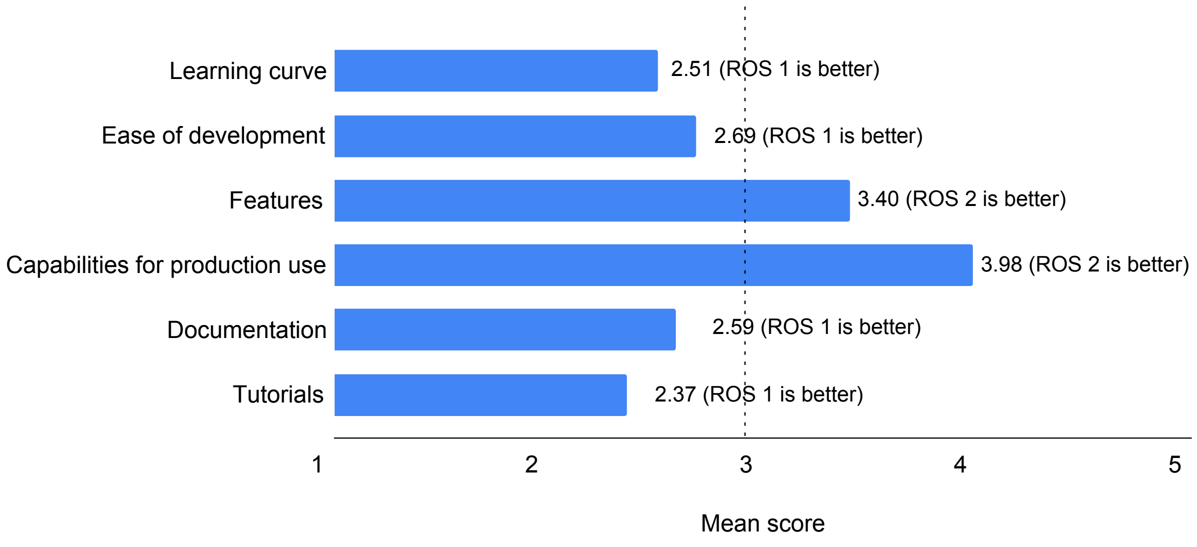


Figure 3.22: Mean scores for the question: How do you compare ROS 1 with ROS 2 in terms of ...?. The scale used for this question was: 1 (ROS 1 is much better), 2 (ROS 1 is slightly better), 3 (ROS 1 and ROS 2 are on the same level), 4 (ROS 2 is slightly better), 5 (ROS 2 is much better).

3.1.5 Migration from ROS 1 to ROS 2

In this section, a closer look is taken at how developers feel about the process of migrating a ROS 1 package to ROS 2. From Figure 3.23, we can see that 52.6% of the developers who have used ROS 2 or are using it at the moment have migrated a ROS 1 package before. From our data, another interesting result is that, out of these developers who have gone through a migration process, there is no developer with an expertise level lower than Experient User,

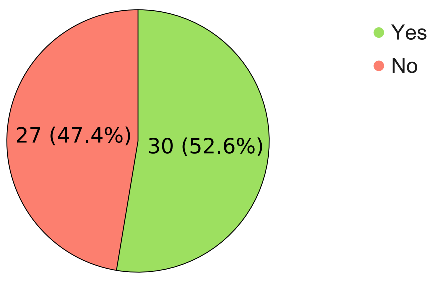


Figure 3.23: Results to question: Have you ever migrated a ROS 1 package to ROS 2?

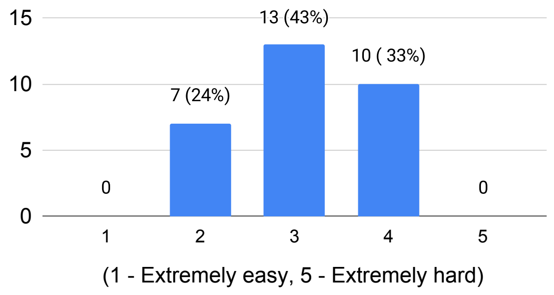


Figure 3.24: Results to question: How difficult did you find the process of migrating a ROS 1 package to ROS 2?

which means that migrating a ROS 1 package is a challenge only being taken by developers with considerable ROS expertise.

Looking at Figure 3.24 we can see that developers do not find the process of migrating a ROS 1 package to be a straightforward task, with 43% saying that the process is not easy nor hard, and one third (33%) saying that the process is hard.

When asked about what difficulties in specific did they face (see Table 3.1) the most mentioned issue was dependency on packages that were not available in ROS 2 (90%). Another issue, already mentioned in the analysis of other questions, is the lack of API documentation for the two main client libraries, *rclcpp* and *rclpy*, and for the Python Launch API as well, when migrating ROS launch files from XML to Python.

The lack of tutorials and adequate documentation on how to migrate a ROS 1 package to ROS 2 is also an issue mentioned by 46% of developers. In fact, on the ROS 2 documentation website, there is only one guide that covers how to migrate to ROS 2 a basic example of a publish-subscribe ROS 1 package [2].

3.1.6 ROS 2 features

In this section of the user study, the intent is to find out how respondents feel about certain ROS 2 features. Each feature was rated in 4 different categories (Essential, Important, Compelling, Useful) using a Likert scale ranging from 1 (Not at all) to 5 (Very Much). This scale allows us to distinguish between essential features, which are considered "must-have" requirements, and important features, which are considered important but not urgent. It also allows us to inquire whether a feature is useful but not important or essential.

The results are presented in Figure 3.25 for each category. From our data we can see

[2] `https://docs.ros.org/en/foxy/The-ROS2-Project/Contributing/Migration-Guide.html`

Table 3.1: Results to question: What difficulties did you face when migrating the package?

	Nr. of answers
Dependency on packages that were not available in ROS 2	27 (90%)
Lack of API documentation for rclcpp or rclpy	19 (63%)
Not enough documentation for the Python Launch API	17 (57%)
There is no standard to follow when migrating a package	14 (46%)
Significant changes in core packages (e.g. ROS1: move_base to ROS2: Navigation2)	13 (43%)
Not enough tutorials to follow that teach you how to migrate a package	12 (40%)
Migrating from roscpp/rospy to rclcpp/rclpy	7 (23%)

that the majority of the features are rated above 3 points (mid point) in every category. The two features that do not follow this trend are **ROS 2 support on Windows** and **ROS 2 support on macOS**, with low mean scores in the Essential and Important categories.

From these features, there are two that stand out from the rest with an average score of over 4 points, namely **ROS 1 bags in a ROS 2 environment** and **ROS 2 support for real-time systems**. The former is particularly important for development teams that have created significant large databases of bag files in ROS 1 and understandably do not want to stop using them, but are considering a move to ROS 2. As explained in Section 2.5.1, ROS 1 bag files can be converted to ROS 2 compatible bag files. The latter is consistent with the need to make ROS a viable product in an industrial environment.

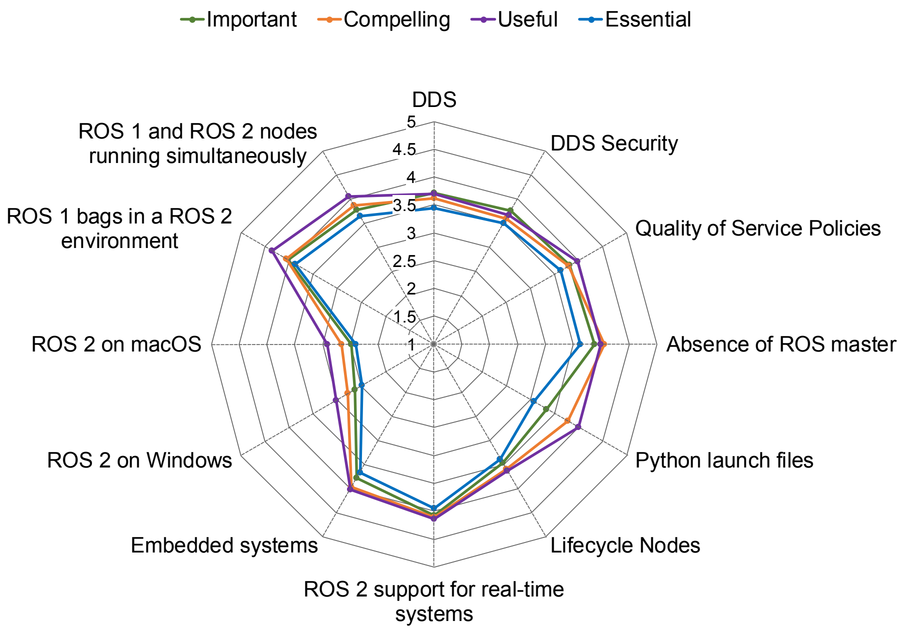


Figure 3.25: Mean scores for the question: How essential/important/compelling/useful do you find these ROS 2 features?

3.1.7 Support for Multi-Robot Systems in ROS 2

The main goal of this section is to evaluate the relevance of the new ROS 2 features for Multi-Robot Systems. First, developers were asked if they were interested in working with Multi-Robot Systems, to which 68% of 103 developers answered yes.

Table 3.2: Results to question: What main difficulties have you found when developing Multi-Robot Systems with ROS1?

	Nr. of answers
Multimaster communication when working with real robots	**44 (63%)**
Managing the complexity of the TF tree with different robots	34 (49%)
Namespace management	33 (47%)
No available standard to develop Multi-Robot Systems	30 (43%)
Not enough debugging tools designed specifically for Multi-Robot Systems	29 (41%)

Next, this subset of respondents was asked what difficulties they encountered in developing Multi-Robot Systems in ROS 1. The intent with this question was to find out if ROS 2 and its new features could solve some of those difficulties. Looking at Table 3.2, we see that the most common difficulty found was **Multimaster communication when working with real robots**. This difficulty is indeed solved by ROS 2 and the use of DDS and its Dynamic Discovery, which is a strong argument for using ROS 2 one wants to work with Multi-Robot Systems. All other difficulties mentioned in Table 3.2 are still difficulties considered to be present in ROS 2.

In Figure 3.26, we can understand how ROS 2 features stand in terms of relevance of Multi-Robot Systems. Not surprisingly, the results are generally not very different from the results of Section 3.1.6. However, there is one feature that stands out in the case of Multi-Robot Systems, which is **the Absence of ROS Master** (4.27). This feature and the **use of DDS** (3.97) is the main reason as to why developers interested in Multi-Robot Systems should consider migrating their applications to ROS 2.

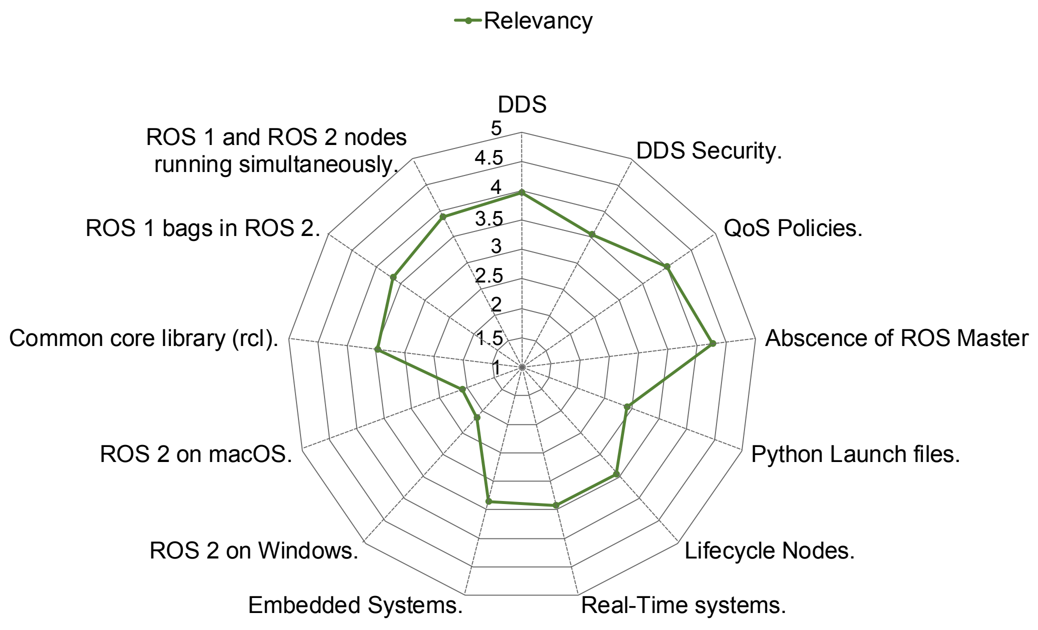


Figure 3.26: Mean scores to the question: How relevant are the these new ROS 2 features to Multi-Robot Systems? (1-Not at all; 5-Very much)

3.2 Summary

The user study presented in this chapter allowed us to learn important aspects related to the adoption of ROS 2, how it is viewed by the ROS community, and what prevents developers from migrating from ROS 1 to ROS 2.

The first conclusion we can take from this user study is that ROS 2 is heading in the right direction in terms of what developers expect from a robotics middleware, and to cover the weaknesses of ROS 1. There are four features that were lacking in ROS 1 that are directly addressed in ROS 2 through the use of DDS. These four features are: **reliability**, **robustness**, **security**, and **support for multi-robot systems**.

Regarding the awareness of ROS 2, we can conclude that ROS 2 and its new features are being well marketed, as the vast majority of respondents knew about the existence of ROS 2 and its new features. We can also conclude that the lack of adoption of ROS 2 is not due to the fact that developers do not know about its existence.

By far, the major problem that developers see in ROS 2 is the lack of ROS 1 packages migrated to ROS 2. The dependency on ROS 1 packages largely discourages developers from doing the migration. This is a difficult problem to solve because the vast majority of the community is waiting for the ROS 1 packages to be migrated, which in turn drastically slows down the speed of adoption. Developers may not start migrating until ROS 1 Noetic reaches its end of life (May 2025), and therefore this problem may persist until then.

Another issue that goes against the migration is the lack of documentation on how to migrate a ROS 1 package, and also the lack of documentation in general. More tutorials are needed, e.g. on how to migrate a service-client package or an action client-server ROS 1 package. Also related, there could also be a tutorial showing how to migrate from *move_base*[3] to the ROS 2 navigation stack, *Navigation2* [58]. Providing a simple example package, such as, a robot traversing several fixed waypoints.

Creating real-world use cases about ROS 1 packages that benefit from migrating to ROS 2 may also be of great help to the ROS community. If there were such evidence showing clear results of improvement, developers would be more convinced that ROS 2 is ready and worth replacing ROS 1. From our results, some developers are concerned that the migration requires too much effort and the benefits are not that great for their use case. Therefore, examples like this one might be convincing for more skeptical users.

One limitation of this user study is that we did not distinguish between ROS 2 users who have never used ROS 1 and users who have used it before. If we had made this distinction, we would be able to know whether there is a difference in viewpoint between these two groups of ROS 2 users. We could, for example, answer the question: "Is there a natural bias of users who have used ROS 1 when they use ROS 2?".

In the remaining chapters of this dissertation, we report on the technical challenges of migrating from ROS 1 to ROS 2. We provide a real example of migrating a ROS 1 package, *patrolling_sim*, to ROS 2 and make a performance comparison between the two versions of the package in terms of network performance (latency) and computational resources usage.

[3]`http://wiki.ros.org/move_base`

4 Migrating from ROS 1 to ROS 2

In this chapter, we go through the process of migrating a complex ROS 1 package to ROS 2. In particular, we go over the changes that must be made to a ROS 1 package to make it compatible with ROS 2, namely package **metadata files**, **source code**, and **launch files**. At the end of the chapter, we address the problems encountered during the migration process.

4.1 Migrating the code base

The package selected for this demonstration is the *patrolling_sim*[1] [13, 59] ROS package. This package consists of a multi-robot patrolling simulation package based on the Stage simulator [60] capable of performing patrolling tasks in a coordinated manner without the need for a central computer. It uses all the 3 communication mechanisms provided by ROS, which are: **topics**, **services** and **actions**, and intensively makes use of the ROS 1 navigation stack. For those reasons, we consider it a useful and comprehensive use case for migrating a ROS 1 package to ROS 2.

4.1.1 Package metadata files

CMakeLists.txt file

The first thing to change in any ROS 1 package that one wants to migrate to ROS 2 is in the `CMakeLists.txt` file, which is the minimum version of CMake required. ROS 2 requires that the minimum version of CMake is 3.5.

```
#cmake_minimum_required(VERSION 2.8.3)
cmake_minimum_required(VERSION 3.5)
```

As mentioned in Chapter 2, ROS 2 uses newer C++ versions such as C++11 and C++14, so it is recommended to add the following code block to enable support for C++14:

[1]`https://github.com/davidbsp/patrolling_sim`

```
set(CMAKE_CXX_STANDARD 14)
```

The next changes that need to be made are related to the migration from *catkin* to *ament_cmake*. Instead of a single call to the `find_package` function with all dependencies, *ament_cmake* requires you to specify these dependencies one by one, as you can see in the next code block.

```
# find_package(catkin REQUIRED COMPONENTS
#   roscpp
#   actionlib
#   ...
#   message_generation
# )

find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(rclcpp_action REQUIRED)
# ...
find_package(rosidl_default_generators REQUIRED)
```

An additional change made to `CMakeLists.txt` in ROS 2 is that every file that needs to be accessed at runtime must be installed, i.e. these files are copied into the share directory of the package, which is `/<ros2_ws>/install/<package_name>/share/<package_name>`. This is done through the function call `install()`, in our case these files are maps, parameter files, launch files and configuration files. This can be done following the guidelines below:

```
install(TARGETS <executables> DESTINATION lib/${PROJECT_NAME})
install(DIRECTORY <directory_name> DESTINATION share/${PROJECT_NAME})
# ...
```

Finally, at the end of the `CMakeLists.txt` file, one must replace `catkin_package()` and its arguments with `ament_package()` to use *ament_cmake* instead of *catkin*.

Package manifest file

In the file `packge.xml` there are not many differences between ROS 1 and ROS 2.

The first change one needs to make to this file is to update the XML format version in the tag `<package format="">`. ROS 2 does not support format version 1, only 2 or higher. One change that results from updating the format version is to replace the `<run_depend>` tag with `<exec_depend>`. Another change that needs to be made is to change the build tool from *catkin* to *ament_cmake*. To do this, you need to change the tag `<buildtool_depend>` from ROS 1, to `<buildtool_depend> ament_cmake </buildtool_depend>` for ROS 2.

In Appendix D, we show a difference file with all the changes made to the `CMakeLists.txt` and the `package.xml` files.

4.1.2 Conversion of launch files from XML to Python

In ROS 2, one can use XML, YAML, or Python formats to conceive launch files. Since in ROS 1 the launch files are written in XML, it is easier to port these launch files to ROS 2 by keeping the language in XML. To do this, one can follow the official XML migration guide[2].

However, to fully engage with ROS 2 and assess the difficulty of the process, we also look into porting the launch files written in XML to Python. We are not going to analyze the migration to YAML launch files, as they do not add any new logic nor complexity.

Before going further, it is important to clarify two definitions of this new ROS 2 launch system, namely **launch description** and **actions**. A **launch description** is described as an ordered set of actions or a group of actions, and an **action** is an instruction to do something, such as launch a node or include another launch description. One can think of actions as the former ROS 1 XML tags in the XML launch files, and a launch description as the launch file itself.

Several ROS 1 XML tags are commonly used in launch files, such as: `<group>`, `<include>`, `<arg>`, `<rosparam>`, and `<param>`, all of which were used in the context of the *patrolling_sim* ROS package.

The `<group>` tag in ROS 1 is used to group a set of launch instructions, optionally, so that they can all be executed under the same namespace (ns attribute) with `<group ns=''>`. In Python launch files, this behavior is performed by the function `GroupAction()`, which takes as argument a set of actions. In conjunction with `PushRosNamespace()` one can group all actions under the same namespace.

The `<include>` tag is used to include other launch files in a launch file. This behavior is reproduced in Python launch files with the functions `IncludeLaunchDescription()` and `PythonLaunchDescriptionSource()`. With these two functions one can include a Python launch file and use it as a launch description.

The `<arg>` tag is used to pass arguments to the launch file via the command line. In ROS 2, this behavior is achieved by the `DeclareLaunchArgument()`, which is an action that declares a new launch argument. Every launch argument has a name associated with it, and the parameter value can be accessed with the `LaunchConfiguration()` function if the launch argument name is given as an argument to the function call.

To load node parameters, there are two general ways one can do it. In XML launch files, one can use the `<param>` tag to set a parameter value, or the `<rosparam>` tag to load a YAML file with parameters. In ROS 2 Python launch files, to load parameters specific to a node, you can pass an array of parameters (or the path to a YAML file containing the

[2]`https://docs.ros.org/en/humble/How-To-Guides/Launch-files-migration-guide.html`

parameters) to the `Node()` action.

Table 4.1 presents a comparison between XML (ROS1) and Python (ROS2) launch files.

Table 4.1: Conversion of ROS 1 XML to ROS 2 Python launch files

XML	Python
`<node pkg="" type="" name=""/>`	`Node(package='', executable='',` ↪ `name='')`
`<include file=""/>`	`IncludeLaunchDescription(` ↪ `PythonLaunchDescriptionSource(` ↪ `launch_file_path=''))`
`<arg name="" default=""/>`	`DeclareLaunchArgument(name='',` ↪ `default_value='')`
`<group ns="">` `<node pkg="" type="" name=""/>` `<node pkg="" type="" name=""/>` `</group>`	`GroupAction(actions=[` `PushRosNamespace(namespace=''),` `Node(...),` `Node(...)` `])`
`<node pkg="" type="" name="">` `<param name="" value=""/>` `</node>`	`Node(package='', executable='',` ↪ `parameters=[{"name": value}],` ↪ `name='')`

4.1.3 Conversion of source code files from roscpp to rclcpp

Most C++ source code is migrated by replacing the references to *roscpp* with *rclcpp* and then adapting the functions to the newer C++ client library. In this subsection, we provide additional details using the *patrolling_sim* package.

The source code of the *patrolling_sim* package consists mainly of creating subscriber/publishers, service clients and interfacing with the navigation stack, for that reason, we will focus on giving examples on how to migrate those ROS communication behaviors/patterns.

We begin with the creation of a node. The way a node is created differs from ROS 1 to ROS 2. In ROS 1, we initialize the node by passing the name of the node to the library's initialization call and then create a node handle. In ROS 2, we also initialize the node, but then pass the name of the node to the constructor of the node object. Another aspect worth highlighting is the use of C++ smart pointers, namely `shared_ptr`, to handle most of ROS 2's objects, such as publishers, subscribers, node handles, service servers, and clients. This is a significant change from ROS 1, where objects are stored by copy. When using the ROS

2 C++ client library, ownership of an object is managed by a smart pointer and its lifecycle and memory usage are managed automatically, i.e. when the smart pointer is no longer used, the memory it points to is freed.

```cpp
// Creating a node in ROS 1
// ros::init(argc, argv, "patrol_agent");
// ros::NodeHandle nh;

//Creating a node in ROS 2
rclcpp::init(argc, argv);
std::shared_ptr<rclcpp::Node> n_ptr =
 ↪  std::make_shared<rclcpp::Node>("patrol_agent");
```

Moreover, in ROS 2, parameters work differently, as mentioned in Section 2.5.1. Since in ROS 2 parameters exist only in the scope of a node, this affects how parameters are accessed and how they are set in the code. In ROS 2, parameters must be declared in the code before they are used. Optionally, they can also have a default value that is used if the parameter cannot be read.

```cpp
// Declaring ang retrieving a parameter in ROS 1
/*if (! ros::param::get("/goal_reached_wait", goal_reached_wait)) {
  //goal_reached_wait = 0.0;
  ROS_WARN("Cannot read parameter /goal_reached_wait. Using default
 ↪  value!");
  //ros::param::set("/goal_reached_wait", goal_reached_wait);
}*/

// Declaring ang retrieving a parameter in ROS 2
n_ptr->declare_parameter<double>("goal_reached_wait",0.0);
if (!n_ptr->get_parameter("goal_reached_wait", goal_reached_wait)) {
    goal_reached_wait = 0.0;
    RCLCPP_WARN(n_ptr->get_logger(),"Cannot read parameter!");
}else{
    RCLCPP_INFO(n_ptr->get_logger(),"Parameter set succesfully!");
}
```

In the example above, one can see the changes in console logging functions. Most of the functions work the same way, the only changes are the name of the function and that these new functions get a logger as an argument. Each node has a logger associated with it that automatically contains the name and namespace of the node.

Another change that needs to be addressed is the included headers. As mentioned in Section 2.5.1, the header files of the included interfaces have an additional namespace corresponding to the type of interface. An example of the required changes can be found below.

```cpp
// Including headers in ROS 1
// #include <nav_msgs/Odometry.h>
// #include <move_base_msgs/MoveBaseAction.h>

// Including headers in ROS 2
#include "nav_msgs/msg/odometry.hpp"
#include "nav2_msgs/action/navigate_to_pose.hpp"
```

The logic behind creating a publisher/subscriber has remained mostly the same. One notable change can be seen in the code for creating a subscriber, as presented below. Due to the transition to a newer version of C++, namely C++11, there is no longer a dependency on the *boost* library, which is a set of libraries that provide support for additional data structures and tasks, such as threads and smart pointers. Therefore, all references to this library should be removed and replaced with the C++ standard library.

```cpp
// Creating a publisher and a subscriber in ROS 1
// positions_pub = nh.advertise<nav_msgs::Odometry>("positions", 1);
// positions_sub = nh.subscribe<nav_msgs::Odometry>("positions", 10,
↪  boost::bind(&PatrolAgent::positionsCB, this, _1));

// Creating a publisher and a subscriber in ROS 2
positions_pub =
↪  n_ptr->create_publisher<nav_msgs::msg::Odometry>("/positions",1);
positions_sub =
↪  n_ptr->create_subscription<nav_msgs::msg::Odometry>("/positions",10,
↪  std::bind(&PatrolAgent::positionsCB,this,std::placeholders::_1));
```

When it comes to creating a service client and making a service request, e.g. to clear the robot's navigation costmap, the logic has changed a bit. In ROS 2 one has to create the service client before making the request to the server, whereas in ROS 1 you simply call the service with the name of the server and the request object.

After creating the service client, in ROS 2, one has to make an asynchronous request to the server and then wait for the server to send the result with `exec.spin_until_future_complete(result)`.

Additionally, in ROS 1, the request and the response were part of the same data structure, while in ROS 2 they are two different classes that share the same namespace, which is the service name. This change can be seen in the code below when we create the request object in the second line.

```cpp
// std_srvs::Empty srv;
auto request =
↪  std::make_shared<nav2_msgs::srv::ClearEntireCostmap::Request>();

```

```cpp
std::string mb_string = "local_costmap/clear_entirely_local_costmap";

// Making a service call in ROS 1
/* if (ros::service::call(mb_string.c_str(), srv)){
    ROS_INFO("Costmap correctly cleared.");
}else{
    ROS_WARN("Was not able to clear costmap");
}*/

// Making a service call in ROS 2
auto clear_client =
↪  n_ptr->create_client<nav2_msgs::srv::ClearEntireCostmap>(
↪  mb_string.c_str());

auto result = clear_client->async_send_request(request);
if (exec.spin_until_future_complete(result) ==
↪  rclcpp::FutureReturnCode::SUCCESS) {
  RCLCPP_INFO(n_ptr->get_logger(), "Costmaps cleared.\n");
} else {
  RCLCPP_ERROR(n_ptr->get_logger(), "Was not able to clear costmap");
}
```

When it comes to the action client, whose job in *patrolling_sim* it is to send navigation goals to the *Navigation2* `NavigateToPose` Action Server, the changes from the *move_base* Action Client are shown below. As one can see, most of the logic for creating an action client is still the same. However, one difference we would like to point out is the use of Time in the ROS 2 implementation, namely the use of `n_ptr-> now()` to retrieve the current clock time instead of `ros::Time::now()`. This is merely a sintax change, it still returns the system time (wall_clock) or the current simulation time (sim_time) depending on the type of clock that is being published to the `/clock` topic.

```cpp
// Creating the navigation goal object in ROS 1
//move_base_msgs::MoveBaseGoal goal;

// Creating the navigation goal object in ROS 2
nav2_msgs::action::NavigateToPose::Goal goal;

//Send the goal to the robot (Global Map)
//geometry_msgs::Quaternion angle_quat =
↪  tf::createQuaternionMsgFromYaw(0.0);
tf2::Quaternion tf2_quat;
tf2_quat.setRPY(0.0,0.0,0.0);
geometry_msgs::msg::Quaternion angle_quat = tf2::toMsg(tf2_quat);
goal.pose.pose.orientation = angle_quat;

goal.pose.header.frame_id = "map";
```

```

//goal.target_pose.header.stamp = ros::Time::now();
goal.pose.header.stamp = n_ptr->now();

goal.pose.pose.position.x = target_x;
goal.pose.pose.position.y = target_y;
goal.pose.pose.orientation = angle_quat;

// Send an action goal to the action server in ROS 1
//ac->sendGoal(goal, boost::bind(&PatrolAgent::goalDoneCallback, this, _1, _2),
//                    boost::bind(&PatrolAgent::goalActiveCallback,this),
//                    boost::bind(&PatrolAgent::goalFeedbackCallback, this,_1));

// Send an action goal to the action server in ROS 2
auto send_goal_options = rclcpp_action::Client<nav2_msgs::action::NavigateToPose>::SendGoalOptions();
send_goal_options.goal_response_callback = std::bind(&PatrolAgent::goalActiveCallback, this, _1);
send_goal_options.feedback_callback = std::bind(&PatrolAgent::goalFeedbackCallback,this,_1,_2);
send_goal_options.result_callback = std::bind(&PatrolAgent::goalDoneCallback,this,_1);

auto goal_handle_future = ac->async_send_goal(goal,send_goal_options);
```

4.2 Final remarks

After the migration process, we had a working version of the *patrolling_sim* in ROS 2[3] using a publicly available version of the Stage simulator for ROS 2 hosted on `https://github.com/woawo1213/stage_ros2`.

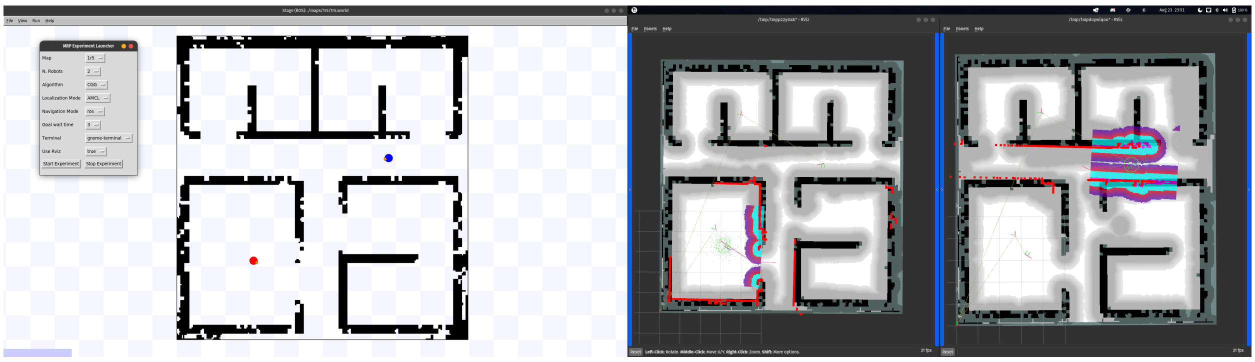
Figure 4.1 shows a screenshot of *patrolling_sim* running with Rviz2[4]. Rviz2 is the ROS 2 version of the ROS visualizer available in ROS 1. It is used to visualize data exchanged in topics, such as laser scans, transformations (tf), navigation costmaps, etc.

A video of the ROS 2 *patrolling_sim* running with a team of 8 robots is also available at `https://youtu.be/B9R6FrFS5OA` (without Rviz2) and `https://youtu.be/fQpPshj2weI` (with Rviz2).

Since Stage is a rather limited simulation environment, i.e. it does not support 3D simulations, we decided to also migrate *patrolling_sim* to support a more realistic and

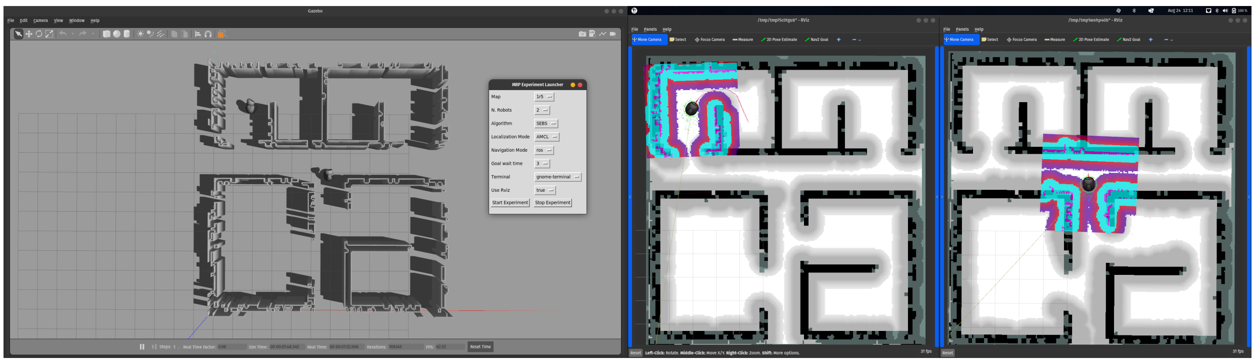
[3]`https://github.com/ccpjboss/patrolling_sim_ros2`

[4]`https://github.com/ros2/rviz`



Map 1r5
N. Robots 2
Algorithm CGG
Localization Mode AMCL
Navigation Mode ros
Goal wait time 3
Terminal gnome-terminal
Use Rviz true

Figure 4.1: Patrolling Sim in the Stage Simulator (left) and RVIZ of two patrol robots (right).



Map 1r5
N. Robots 2
Algorithm SEBS
Localization Mode AMCL
Navigation Mode ros
Goal wait time 3
Terminal gnome-terminal
Use Rviz true

Figure 4.2: Patrolling Sim in the Gazebo Simulator (left) and RVIZ of two patrol robots (right).

powerful 3D simulator, Gazebo [26]. Another reason to go through the effort of having a Gazebo version of *patrolling_sim* is that Gazebo is supported by the same company as ROS, Open Robotics, unlike Stage for ROS 2, which is a community version, offering less long time support. To migrate to Gazebo, we needed several prerequisites: 1) a 3D model of the maps used in Stage; 2) a 3D model and URDF description file of the robots to be used.

The maps used for the Stage simulator are in bitmapped image format (`.pgm`) and Gazebo needs a 3D model of the map for the simulation to work. For this, the 2D `.pgm` images have been extruded into 3D models of the maps in `.stl` format. For the 3D model of the robots, we used the 3D models available for the STOP project [35] and created the URDF file, which consists of an XML format for representing the 3D model of a robot, including its joints and sensors.

Figure 4.2 shows a screenshot of *patrolling_sim* running in a Gazebo simulation environment with 2 robots. A video of the ROS 2 *patrolling_sim* running with Gazebo with a team of 4 robots is also available at `https://youtu.be/8UYlcr0yj0c`.

In the course of working with ROS 2 compatible simulators, we also considered migrating *patrolling_sim* to the CoppeliaSim 3D simulator [27], but the effort required to do so is beyond the goal of this dissertation, as it would not have added any benefit that Gazebo does not have. Nevertheless, the author of this dissertation has contributed to the official repository[5] of the interface CoppeliaSim ROS 2 to make the simulation environment compatible with *ROS 2 Galactic*.

4.3 Summary

Despite the fact that every migration process is specific to the package being migrated, we consider these these migration guidelines to be of essential importance and, by following them, one should be able to achieve the objective of having a working version of a ROS 1 package in ROS 2 and be at the edge of robotics development.

In the next chapter, we compare the ROS 1 and ROS 2 version of the *patrolling_sim* package in terms of network performance and resource usage.

[5]`https://github.com/CoppeliaRobotics/simExtROS2/pull/11`

5 Comparing performance of ROS 1 against ROS 2

In this chapter, we describe the experiments conducted to compare the performance of ROS 1 and ROS 2 using the *patrolling_sim* package. The metrics evaluated were **network latency** (wired and wireless) and **cpu/memory usage**. Firstly, details about the experimental design and metrics used are described. Secondly, an in-depth analysis and discussion of the results are provided.

5.1 Experimental Design

With these experiments, we aim to evaluate the robustness and reliability of a ROS system when multiple roboitc agents with distributed ROS nodes operate in the same network. This way, we can evaluate how the ROS network would behave under load when multiple robots operate in the same network. As such, we will increase the number of robotic agents that are part of the MRS in order to evaluate the scalability of the system.

To evaluate the network latency of the ROS system running *patrolling_sim*, twenty different experiments were conducted in two different scenarios:

- **Scenario 1 - Ethernet:** where the patrol agents are connected to the network via Ethernet.
- **Scenario 2 - WiFi:** where the patrol agents are connected to the network through WiFi.

In these experiments, we had a computer running the Stage simulator and other PCs connected to the same network running the *patrolling_sim* package. The latter are denoted **patrol agents**. This way, we can measure the network latency of messages going from the computer running the simulator to the patrol agents, and vice versa.

In the context of these experiments, we define latency as the interval of time between sending the message (T_{send}) and acknowledging it (T_{ack}), i.e. the round-trip time, divided by

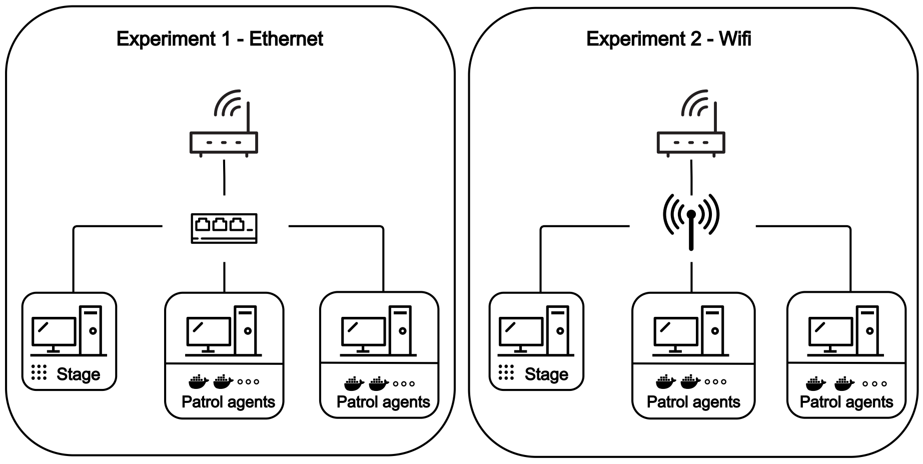


Figure 5.1: Experimental setup with wired and wireless connections for latency measurements.

two (see Equation 5.1). We have used the existing network in the Institute of Systems and Robotics of University of Coimbra and conducted the experiments at a time when activity on the network was at a minimum. Another important aspect to point out is that the ROS 2 experiments made use of the default DDS provider for *ROS 2 Galactic*: the Cyclone DDS from Eclipse Foundation.

$$Latency = \frac{T_{ack} - T_{send}}{2} \tag{5.1}$$

In our experiments, the total number of patrol agents is divided equally by two computers. Thus, if the total number of patrol agents of the experiment is two, one patrol agent is run on each computer.

To achieve results as similar as possible to a real-world scenario where each patrol agent runs on a different machine, we run each patrol agent in an isolated Docker container[1]. Thus, each patrol agent has its own unique MAC address and IP address, and similarly to a real robot, each Docker container uses a *macvlan* network driver to mimic a physical network interface which is directly connected to the physical network [61].

The *Wireshark*[2] network tool is used to measure the network latency in the experiments. With *Wireshark*, we could record all traffic on the network during the experiments and analyze it afterwards. Latency was measured using a Wireshark filter that only displays traffic related to a topic from ROS to which each patrol agent sends a message and the central computer subscribes to that topic and receives that message.

To evaluate the resource utilization (CPU and RAM) of the two versions of the *patrolling_sim* package (ROS 1 and ROS 2), a **new scenario was created**.

- **Scenario 3 - Resource usage:** the Stage simulator and the patrol agents are all run

[1]`https://www.docker.com/resources/what-container/`

[2]`https://www.wireshark.org/`

in the same machine. CPU and RAM usage is measured.

The experiments conducted in this scenario were run on a single machine and, as in the case of scenarios 1 and 2, with an increasing number of robotic agents. The specifications of the machine used for this third experimental scenario can be found in Table 5.1. Resource usage was logged to a *csv* file using a Python script and the *psutil*[3] library.

Table 5.1: Specifications of the machine used for resource usage measurements.

Operating System	Ubuntu 20.04
CPU Model	Intel i7-7700 HQ (8 cores)
CPU Speed	3.8 GHz
RAM	15885 MiB

One aspect that affects the usage of CPU is how we parameterize the navigation stack. During the migration, we made sure that all parameters remained the same by comparing the documentation pages of both navigation stacks. To our favor, most parameters kept the same name and also the same functionality. Since *patrolling_sim* has multiple patrolling algorithms [59], this was also taken into account when performing the experiments, so we have used the same method — the cyclic algorithm — in all trials to guarantee a fair comparison.

It is important to note that, for the scope of this dissertation, there was no benefit on running these experiments with real robots on a real scenario, since our focus is on the network performance of the middleware and the usage of computational resources.

5.2 Results and Discussion

In this section, we present and analyze the results obtained, drawing relevant observations and conclusions.

Figure 5.1 shows that ROS 2 performs better than ROS 1 in terms of network latency. In the Ethernet experiments, ROS 1 achieves similar results to ROS 2 only when there are 2 robotic agents in the network. When there are more than two, ROS 2 performs significantly better than ROS 1, keeping the maximum latency below 0.57 ms, while ROS 1 achieves higher maximum latencies (up to 5.09 ms). Overall, these results show that the communication in ROS 1 is more unstable compared to ROS 2 when using an Ethernet connection.

When the patrol agents are connected via WiFi, ROS 2 significantly outperforms ROS 1. For the case when 10 robotic agents were part of the system, the performance gained

[3]`https://pypi.org/project/psutil/`

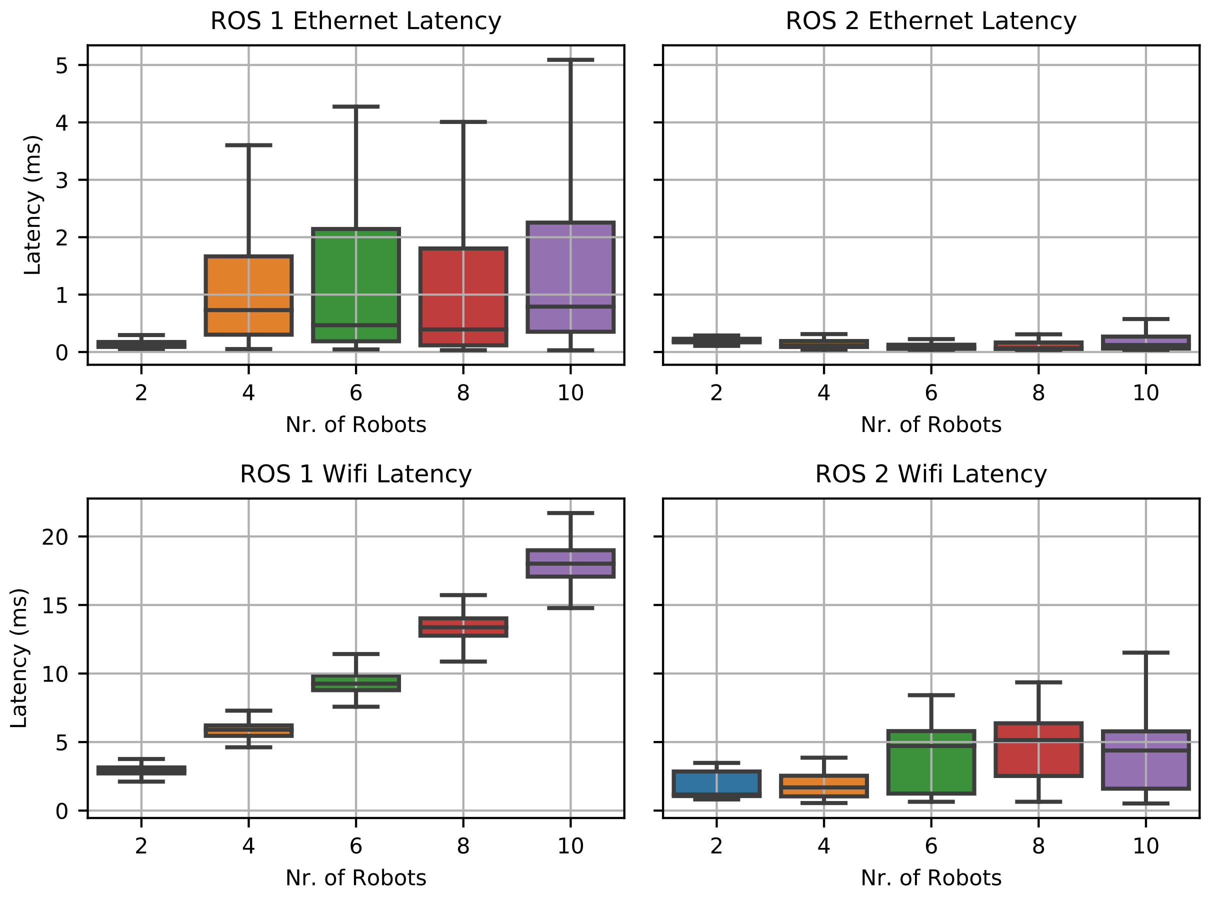


Figure 5.2: Network latency with different ROS versions while running the *patrolling_sim* package.

was around 317.74% (regarding mean values). We can also see that the median latencies in ROS 2 (1.18 - 5.14 ms) are much lower than the median latencies in the ROS 1 network (2.89 - 18.01 ms). This result confirms the selling point of ROS 2, proving that it has higher performance and robustness than ROS 1 in less reliable networks like WiFi, and scales better to more robotic agents.

The performance gains in ROS 2, both when using Ethernet and WiFi, can be explained by the change in the communication middleware between ROS 1 and ROS 2, namely the replacement of TCPROS/UDPROS with the industry-proven DDS. As already noted by other researchers, cf. Section 2.7, and now by these results, ROS 2 has gained in reliability, robustness and scalability by using DDS, compared to ROS 1.

Analyzing now Figure 5.3, it can be seen that the resource usage of ROS 2 is higher than that of ROS 1, both in the case of CPU and RAM. When simulating 10 robots, CPU reaches a median usage of 70% with ROS 2, while ROS 1 only uses about 40% of CPU (median value). For memory usage, we can see that ROS 1 keeps an almost constant memory usage (median value), while ROS 2 does not. This confirms the evidence reported in [49] , which also mentions higher memory usage for ROS 2 when compared to ROS 1.

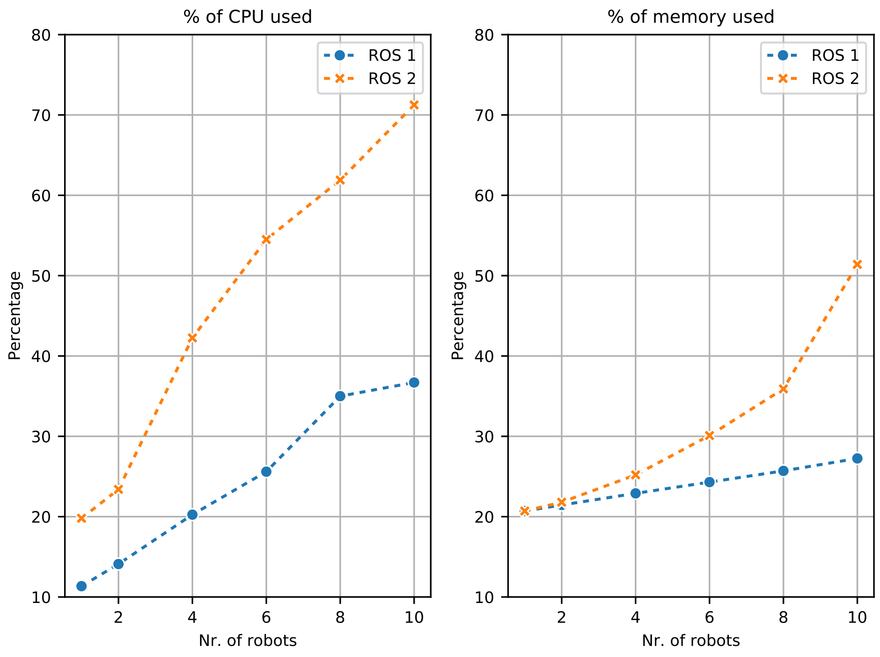


Figure 5.3: CPU and RAM median values with different ROS versions while running the *patrolling_sim* package.

Through our experiments, we found that the main reason for the ROS 2 higher CPU usage comes from the **navigation stack**. It contributes to the higher ROS 2 CPU and RAM utilization due to the number of nodes launched by the navigation stack specific to each robot. For our use case, in ROS 1 the nodes launched per robot are less than in ROS 2. This, clearly shows that the ROS 2 navigation stack is more complex and uses more resources. Detailed results about CPU and RAM utilization can be view in Appendix E.

5.3 Summary

The results of our experiments are useful for the development of MRS in ROS 2, as they show the ways to improve a ROS 1 MRS and to achieve better network performance when migrating to ROS 2. The results show that the performance and scalability of ROS 2 in less reliable networks, such as WiFi, is much better than the performance in ROS 1. This is extremely important for the deployment of MRS with a large number of robots. The network performance, as well as the fact that there is no central failure point unlike ROS 1, show that ROS 2 is an interesting robotics middleware to develop MRS.

6 Conclusion

The work covered in this dissertation focuses on the study of ROS 2 and the subsequent comparison of its performance with ROS 1 in a MRS use case.

First and foremost, we start by studying ROS 1 and its design goals so that we can then have a better understating of how and why ROS 2 was created. We also study the conception of ROS 2, its main differences to ROS 1, as well as newly introduced features in the ROS 2 stack and how they improve the ROS robotics middleware.

With this, we were able to create a carefully designed user study that targeted the robotics community. From this user study, we have found out what the community's requirements are for a robotics middleware, and how ROS 2 can closely meet the robotics community's requirements for a robotics middleware. We were also able to investigate the extent to which ROS 2 is being adopted, who is adopting ROS 2, and what issues are keeping ROS 1 developers from switching to ROS 2. Overall, this user study has shown us what issues persist in ROS 2, and which features of ROS 2 are most popular and well adopted in the robotics community.

After documenting the migration process and ensuring that *patrolling_sim* in ROS 2 works as expected, we ran several experiments in Chapter 5 to compare ROS 1 and ROS 2 in terms of network performance (latency) and computational resources utilization (RAM and CPU). The results found are promising, as ROS 2 can achieve lower latencies than ROS 1 for the same number of robotic agents in the network, in both Ethernet and WiFi scenarios.

We hope that the results presented in this dissertation will contribute to the adoption of ROS 2 by showing the performance gains that can be expected when working with teams of multiple robots in ROS 2.

6.1 Future work

The contributions of this dissertation are relevant for the future use of ROS 2 in robotics. However, since ROS 2 introduces some new features that are out of this dissertation's scope there are some points that should receive attention on future research studies in the topic.

Security concerns are becoming more and more important in the current digitalization age (e.g. Internet of Things, Industry 4.0, etc.), and so this also becomes a problem when using MRS in the real world. ROS 2 security functions should be studied and researched to evaluate the impact when a MRS is run in ROS 2 with security features enabled.

Another aspect that could not be covered in this dissertation was comparing ROS 1 (*move_base*) and ROS 2 (*Navigation2*) navigation stacks. Despite looking similar at a first glance and achieving the same goals, their underlying concepts are much different. While *move_base* uses a monolithic state machine, *Navigation2* uses a Behavior Tree architecture, enabling *Navigation2* to be more modular and highly configurable by rearranging tasks of a robot's navigation behavior.

Another future work in the sequel of this dissertation is conducting a real-world experiment with the ROS 2 package *patrolling_sim*. As far as the author is aware, the mobile robots available at the Institute of Systems and Robotics have not yet been ported to ROS2. This would allow to run *patrolling_sim* in a real scenario with multiple robots working as team with better robustness and scalability to larger teams. Moreover, the same metrics used in this dissertation could be measured to make the adequate observations and conclusions.

Bibliography

[1] Stefan Enderle, Hans Utz, Stefan Sablatnög, Steffen Simon, Gerhard Kraetzschmar, and Günther Palm. “Miro: Middleware for Autonomous Mobile Robots”. In: *IFAC Proceedings Volumes* (). DOI: `10.1016/S1474-6670(17)41721-6`. URL: `https://www.sciencedirect.com/science/article/pii/S1474667017417216`.

[2] *CARMEN*. URL: `http://carmen.sourceforge.net/intro.html` (visited on 11/24/2021).

[3] *Orca Robotics*. URL: `http://orca-robotics.sourceforge.net/index.html` (visited on 11/24/2021).

[4] Brian P Gerkey, Richard T Vaughan, and Andrew Howard. “The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems”. In: International Conference on Advanced Robotics (ICAR 2003).

[5] Paul Newman. “MOOS -Mission Orientated Operating Suite”. In: *Mass. Inst. Technol. Tech. Rep.* 2299 (Jan. 2006).

[6] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. “ROS: an open-source Robot Operating System”. In: vol. 3. Jan. 2009.

[7] *ROS: Home*. URL: `https://www.ros.org/`.

[8] *2021 ROS Metrics Report - General*. Feb. 1, 2022. URL: `https://discourse.ros.org/t/2021-ros-metrics-report/24130` (visited on 09/05/2022).

[9] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. “Robot Operating System 2: Design, architecture, and uses in the wild”. In: *Science Robotics* 7.66 (2022). DOI: `10.1126/scirobotics.abm6074`. URL: `https://www.science.org/doi/abs/10.1126/scirobotics.abm6074`.

[10] *Planning future ROS 1 distribution(s) - General*. ROS Discourse. Oct. 19, 2018. URL: `https://discourse.ros.org/t/planning-future-ros-1-distribution-s/6538` (visited on 10/28/2021).

[11] *Distributions - ROS Wiki.* URL: http://wiki.ros.org/Distributions (visited on 10/28/2021).

[12] David Portugal. *patrolling_sim v2.2.4 (Jan. 2022) ROS Package.* original-date: 2014-04-29T11:38:07Z. Jan. 30, 2022. URL: https://github.com/davidbsp/patrolling_sim (visited on 02/15/2022).

[13] David Portugal, Luca Iocchi, and Alessandro Farinelli. "A ROS-based framework for simulation and benchmarking of multi-robot patrolling algorithms". In: *Robot Operating System (ROS).* Springer, 2019, pp. 3–28.

[14] K. Birman and T. Joseph. "Exploiting virtual synchrony in distributed systems". In: *Proceedings of the eleventh ACM Symposium on Operating systems principles.* SOSP '87. New York, NY, USA: Association for Computing Machinery, Nov. 1, 1987, pp. 123–138. ISBN: 978-0-89791-242-6. DOI: 10.1145/41457.37515. URL: https://doi.org/10.1145/41457.37515 (visited on 04/14/2022).

[15] Jay E. Israel, J. G. Mitchell, and Howard E. Sturgis. "Separating data from function in a distributed file system". In: 1978.

[16] Bruce Jay Nelson. "Remote procedure call". In: 1981.

[17] Simon St. Laurent, Joe Johnston, and Edd Dumbill. *Programming Web services with XML-RPC.* 1st ed. OCLC: ocm47278976. Beijing ; Sebastopol, Calif: O'Reilly, 2001. 213 pp. ISBN: 978-0-596-00119-3.

[18] *Nodes - ROS Wiki.* URL: http://wiki.ros.org/Nodes?action=print (visited on 11/07/2021).

[19] *rospy - ROS Wiki.* URL: http://wiki.ros.org/rospy?distro=noetic (visited on 11/08/2021).

[20] *roscpp - ROS Wiki.* URL: http://wiki.ros.org/roscpp (visited on 11/08/2021).

[21] *Topics - ROS Wiki.* URL: https://wiki.ros.org/Topics (visited on 11/08/2021).

[22] *ROS/TCPROS - ROS Wiki.* URL: http://wiki.ros.org/ROS/TCPROS (visited on 10/28/2021).

[23] *ROS/UDPROS - ROS Wiki.* URL: http://wiki.ros.org/ROS/UDPROS (visited on 10/28/2021).

[24] *Services - ROS Wiki.* URL: https://wiki.ros.org/Services (visited on 11/08/2021).

[25] *Bags - ROS Wiki.* URL: http://wiki.ros.org/Bags (visited on 11/16/2021).

[26] *Gazebo.* URL: http://gazebosim.org/ (visited on 11/16/2021).

[27] E. Rohmer, S. P. N. Singh, and M. Freese. “V-REP: A versatile and scalable robot simulation framework”. In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2013, pp. 1321–1326. DOI: 10.1109/IROS.2013.6696520.

[28] *Master - ROS Wiki*. URL: http://wiki.ros.org/Master (visited on 11/11/2021).

[29] *ROS/Technical Overview - ROS Wiki*. URL: http://wiki.ros.org/ROS/Technical%5C%20Overview?action=print (visited on 11/11/2021).

[30] *Packages - ROS Wiki*. URL: http://wiki.ros.org/Packages (visited on 11/16/2021).

[31] *Manifest - ROS Wiki*. URL: http://wiki.ros.org/Manifest (visited on 11/16/2021).

[32] Gonçalo S. Martins, David Portugal, and Rui P. Rocha. “mrgs: A Multi-Robot SLAM Framework for ROS with Efficient Information Sharing”. In: *Robot Operating System (ROS)*. Ed. by Anis Koubaa. Vol. 895. Series Title: Studies in Computational Intelligence. Cham: Springer International Publishing, 2021, pp. 45–75. ISBN: 978-3-030-45955-0 978-3-030-45956-7. DOI: 10.1007/978-3-030-45956-7_3. URL: http://link.springer.com/10.1007/978-3-030-45956-7_3 (visited on 01/03/2022).

[33] Alessando Farinelli, Luca Iocchi, and Daniele Nardi. “Distributed on-line dynamic task assignment for multi-robot patrolling”. In: *Autonomous Robots* 41 (Aug. 2017). DOI: 10.1007/s10514-016-9579-8.

[34] Alexander Tiderko, Frank Hoeller, and Timo Röhling. “The ROS Multimaster Extension for Simplified Deployment of Multi-Robot Systems”. In: Jan. 2016. ISBN: 978-3-319-26052-5. DOI: 10.1007/978-3-319-26054-9.

[35] David Portugal, André G Araújo, and Micael S Couceiro. “Improving the robustness of a service robot for continuous indoor monitoring: An incremental approach”. In: *International Journal of Advanced Robotic Systems* 18.3 (2021). DOI: 10.1177/17298814211012181. URL: https://doi.org/10.1177/17298814211012181.

[36] *Design*. URL: https://design.ros2.org/ (visited on 11/16/2021).

[37] Jarrod McClean, Christopher Stull, Charles Farrar, and David Mascareñas. “A preliminary cyber-physical security assessment of the Robot Operating System (ROS)”. In: SPIE Defense, Security, and Sensing. Ed. by Robert E. Karlsen, Douglas W. Gage, Charles M. Shoemaker, and Grant R. Gerhart. Baltimore, Maryland, USA, May 17, 2013, p. 874110. DOI: 10.1117/12.2016189. URL: http://proceedings.spiedigitallibrary.org/proceeding.aspx?doi=10.1117/12.2016189 (visited on 11/16/2021).

[38] *Enterprise Open Source and Linux*. Ubuntu. URL: https://ubuntu.com/ (visited on 04/14/2022).

[39] craigloewen-msft. *What is Windows Subsystem for Linux*. URL: https://docs.microsoft.com/en-us/windows/wsl/about (visited on 04/14/2022).

[40] *Actionlib - ROS Wiki*. http://wiki.ros.org/actionlib.

[41] *Actions*. http://design.ros2.org/articles/actions.html.

[42] *DDS Portal – Data Distribution Services*. URL: https://www.dds-foundation.org/ (visited on 10/28/2021).

[43] *What is DDS?* URL: https://www.dds-foundation.org/what-is-dds-3/ (visited on 01/28/2022).

[44] *ROS 2 Middleware Implementation for eProsima's Fast DDS*. original-date: 2015-05-13T14:00:18Z. Nov. 19, 2021. URL: https://github.com/ros2/rmw_fastrtps (visited on 11/25/2021).

[45] *ROS 2 RMW for Eclipse Cyclone DDS*. original-date: 2018-07-09T11:19:43Z. Nov. 19, 2021. URL: https://github.com/ros2/rmw_cyclonedds (visited on 11/25/2021).

[46] *ROS 2 Middleware Layer for RTI Connext DDS*. original-date: 2020-10-19T21:28:35Z. Nov. 19, 2021. URL: https://github.com/ros2/rmw_connextdds (visited on 11/25/2021).

[47] William Woodall (OSRF) Deanna Hood. "ROS 2 Update". In: *ROSCon Seoul 2016*. Open Robotics, Oct. 2016. DOI: 10.36288/ROSCon2016-900775. URL: https://doi.org/10.36288/ROSCon2016-900775.

[48] *Managed Nodes*. https://design.ros2.org/articles/node_lifecycle.html.

[49] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. "Exploring the performance of ROS2". In: DOI: 10.1145/2968478.2968502. URL: https://dl.acm.org/doi/10.1145/2968478.2968502.

[50] Jongkil Kim, Jonathon M. Smereka, Calvin Cheung, Surya Nepal, and Marthie Grobler. "Security and Performance Considerations in ROS 2: A Balancing Act". In: *arXiv:1809.09566 [cs]* (Sept. 24, 2018). arXiv: 1809.09566. URL: http://arxiv.org/abs/1809.09566 (visited on 10/28/2021).

[51] Tobias Kronauer, Joshwa Pohlmann, Maximilian Matthe, Till Smejkal, and Gerhard Fettweis. "Latency Analysis of ROS2 Multi-Node Systems". In: *arXiv:2101.02074 [cs]* (June 11, 2021). arXiv: 2101.02074. URL: http://arxiv.org/abs/2101.02074 (visited on 01/09/2022).

[52] Michael Reke, Daniel Peter, Joschua Schulte-Tigges, Stefan Schiffer, Alexander Ferrein, Thomas Walter, and Dominik Matheis. “A Self-Driving Car Architecture in ROS2”. In: IEEE, Jan. 2020. DOI: `10.1109/SAUPEC/RobMech/PRASA48453.2020.9041020`. URL: `https://ieeexplore.ieee.org/document/9041020/`.

[53] Jaeho Park, Raimarius Delgado, and Byoung Wook Choi. “Real-Time Characteristics of ROS 2.0 in Multiagent Robot Systems: An Empirical Study”. In: (). URL: `https://ieeexplore.ieee.org/document/9172073/`.

[54] Agata Barciś, Michał Barciś, and Christian Bettstetter. *Robots that Sync and Swarm: A Proof of Concept in ROS 2*. Sept. 12, 2019. DOI: `10.48550/arXiv.1903.06440`. URL: `http://arxiv.org/abs/1903.06440` (visited on 06/22/2022).

[55] Tanja Katharina Kaiser, Marian Johannes Begemann, Tavia Plattenteich, Lars Schilling, Georg Schildbach, and Heiko Hamann. *ROS2SWARM - A ROS 2 Package for Swarm Robot Behaviors*.

[56] Andrea Testa, Andrea Camisa, and Giuseppe Notarstefano. “ChoiRbot: A ROS 2 Toolbox for Cooperative Robotics”. In: *IEEE Robotics and Automation Letters* 6.2 (Apr. 2021), pp. 2714–2720. ISSN: 2377-3766, 2377-3774. DOI: `10.1109/LRA.2021.3061366`. URL: `http://arxiv.org/abs/2010.13431` (visited on 10/28/2021).

[57] Dave Coleman, Andy McEvoy, Stephen Brawner, and Mike Lautman. *Customer Stories - Report On Needs of the ROS 2 Community*. 2019.

[58] Steve Macenski, Francisco Martín, Ruffin White, and Jonatan Ginés Clavero. “The Marathon 2: A Navigation System”. In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020. URL: `https://github.com/ros-planning/navigation2`.

[59] David Portugal and Rui P Rocha. “Distributed multi-robot patrol: A scalable and fault-tolerant framework”. In: *Robotics and Autonomous Systems* 61.12 (2013), pp. 1572–1587.

[60] Richard Vaughan. “Massively multi-robot simulation in stage”. In: *Swarm Intelligence* (Dec. 1, 2008). DOI: `10.1007/s11721-008-0014-4`. URL: `https://doi.org/10.1007/s11721-008-0014-4`.

[61] *Use macvlan networks*. Docker Documentation. Aug. 30, 2022. URL: `https://docs.docker.com/network/macvlan/` (visited on 08/30/2022).

Appendix A

ROS 2 Quality of Service Policies

- **History**:
  - Keep last: only store up to N samples
  - Keep all: stores all samples
- **Depth**:
  - Set the N samples from the Keep last policy
- **Reliability**:
  - Best effort: Data delivery is not guaranteed
  - Reliable: Data delivery is guaranteed
- **Durability**:
  - Transient local: Keep several samples for late joining subscribers
  - Volatile: no effort is made to keep samples for late joining subscribers
- **Deadline**: maximum time allowed between messages published to a topic;
- **Lifespan**: maximum time allowed between the publishing and the reception of a message. If the time is expired, then the message gets dropped
- **Lease Duration:** maximum period of time, a publisher that it is alive before the system considers it to be dead
- **Liveliness**:
  - Automatic: The system considers all publishers in a node to be alive for another lease time duration when one of the publishers publishes a message to a given topic
  - Manual by topic: The system considers the publisher to be alive for another *lease time* if it manually asserts that it is alive through a call to the publisher API

For all these options, except for durations, there is also the option to keep the default from the underlying middleware "system default". For duration policies, there is an option for unspecified durations, "default".

A set of QoS policies form a QoS profile, and ROS provides a set of predefined profiles for different use cases (e.g., default, sensor data, services, parameters, and system default).

- **Default:**
  - History: keep last queue size of 10
  - Reliability: reliable
  - Durability: volatile
  - Liveliness: system default
  - Deadline, lifespan, lease duration: default
- **Services:**
  - History: keep last queue size of 10
  - Reliability: reliable
  - Durability: volatile
  - Liveliness: system default
  - Deadline, lifespan, lease duration: default
- **Parameters:**
  - History: keep last queue size of 1000
  - Reliability: reliable
  - Durability: volatile
  - Liveliness: system default
  - Deadline, lifespan, lease duration: default
- **Sensor data:**
  - History: keep last queue size of 5
  - Reliability: best effort
  - Durability: volatile
  - Liveliness: system default
  - Deadline, lifespan, lease duration: default
- **System default:**
  - All QoS Polices are set to: system default

There might be a scenario where the two QoS profiles are not compatible. A connection is only made between a publisher and a subscriber if all the QoS policies within two QoS profiles are compatible. The compatibility of the two QoS profiles is evaluated as follows: the subscriber requests a QoS profile that is its minimally accepted quality of policies. It can be seen as the worst case scenario policies in which the communication must operate. In contrast, the publisher offers a QoS profile that is its best case scenario, its maximum quality QoS profile that the publisher is able to provide. A QoS policy is compatible if the request is not stricter than what the publisher offers.

When comparing these features with already implemented features in ROS1, we can clearly observe that ROS2 is more complete than ROS1, regarding giving more options on the behavior of certain nodes and topics.

Table A.1: Comparative table between ROS2 QoS policies and ROS1 equivalent implementation

ROS2	ROS1
History + Depth	Queue size when creating a publisher/subscriber
Reliability - Best effort	UDPROS (only in roscpp)
Reliability - Reliable	TCPROS
Durability - Transient local	Latching publishers[1]
Durability - Volatile	Not implemented
Deadline	Not implemented
Lifespan	Not implemented
Liveliness + Lease duration	Not implemented

[1]When creating a publisher with *advertise* an option to latch the connection is given. When a connection is latched, the last message published is saved and automatically sent to any future subscribers that connect.

Appendix B

Adoption User Study Questions

The questions included in the User Study are divided into 7 sections, as described bellow. The user study can be view in the following link: `https://forms.gle/PrvmUFxbRkhurT8q8`

B.1 User Profile Characterisation

- What country are you from?
- For how many years have you been using ROS?
- What is the main context of the previous projects in which you have used ROS?
- What do you consider to be your level of proficiency with ROS?

B.2 Robotics middleware and ROS 1 strong/weak points

From this point on, only respondents who have already used ROS proceed to the next questions.

- What are the most important features in a Robotics Middleware?
- What makes ROS 1 a good choice when developing robotics applications?
- In what areas do you think ROS 1 is not very strong?
- How do you feel about ROS 1 Documentation and Tutorials?

B.3 ROS 2 Awareness

- Have you heard about ROS 2?
- Which of these ROS 2 features have you heard about before?
- Have you heard about any negative aspects of ROS 2?
    - Please specify the negative aspect(s) you've heard about.

B.4 ROS 2 Adoption

- Have you used ROS 2?
- What is keeping you from fully adopting ROS 2?
- What do you think is keeping developers from fully adopting ROS 2?
- Why have you decided to fully adopt ROS 2? (only asked if the user is using ROS 2)
- How do you feel about ROS 2 Documentation and Tutorials?
- How do you compare ROS 1 with ROS 2, in terms of its . . .
    - Learning curve
    - Ease of development
    - Features
    - Capabilities for commercial and production use
    - Documentation
    - Tutorials

B.5 ROS 1 to ROS 2 migration

This section is only available to respondents who have already used ROS 2. Also note that only respondents who answered "Yes" to the question “Have you ever migrated a ROS 1 package to ROS 2?" have access to the questions about migration difficulties.

- Have you ever migrated a ROS 1 package to ROS 2?
    - How difficult did you find the process of migrating a ROS 1 package to ROS 2?
    - What difficulties did you face when migrating the package?

B.6 ROS 2 features

- How essential/important/compelling/useful do you find these ROS 2 features?
    - ROS 2 and Data Distribution Service (DDS)
    - ROS 2 and DDS Security
    - Using ROS 2 on low-quality of service networks with the use of different Quality of Service Policies
    - The absence of ROS master, therefore enabling fully distributed systems
    - Python launch files instead of launch files written in XML
    - Manage the life cycle of a node with Lifecycle Nodes
    - ROS 2 support for real-time systems
    - ROS 2 support for bare-metal microcontrollers (ESP32, Raspberry Pi Pico, Teensy 3.2, ...) with micro-ROS
    - Native support for ROS 2 on Windows
    - Native support for ROS 2 on macOS
    - Being able to reuse ROS 1 bags in a ROS 2 environment.

    - Being able to deploy a system comprising both ROS 1 and ROS 2 nodes running simultaneously.

B.7 ROS 1/2 and Multi-Robot Systems

- Have you ever worked with or are interested in Multi-Robot Systems?
- What main difficulties have you found when developing Multi-Robot Systems with ROS1?
- Now we ask you to think about how relevant the new features announced for ROS 2 are, when regarding Multi-Robot Systems:
    - ROS 2 and Data Distribution Service (DDS)
    - ROS 2 and DDS Security
    - Using ROS 2 on low-quality of service networks with the use of different Quality of Service Policies
    - The absence of ROS master, therefore enabling fully distributed systems
    - Python launch files instead of launch files written in XML
    - Manage the life cycle of a node with Lifecycle Nodes
    - Having a common core library *rcl*
    - ROS 2 support for real-time systems
    - ROS 2 support for bare-metal microcontrollers (ESP32, Raspberry Pi Pico, Teensy 3.2, ...) with micro-ROS
    - Native support for ROS 2 on Windows
    - Native support for ROS 2 on macOS
    - Being able to reuse ROS 1 bags in a ROS 2 environment.
    - Being able to deploy a system comprising both ROS 1 and ROS 2 nodes running simultaneously.

B.8 Screening question

- What is ROS?

Appendix C

Adoption User Study Detailed Results

Table C.1: What country are you from?

Country	Nr. of answers
Portugal	27 (22%)
United States	20 (18%)
Germany	14 (13%)
India	8 (7%)
Brazil	4 (4%)
France	4 (4%)
Canada	3 (3%)
Italy	3 (3%)
Poland	3 (3%)
United Kingdom	3 (3%)
Australia	2 (2%)
Colombia	2 (2%)
Finland	2 (2%)
Hungary	2 (2%)
Indonesia	2 (2%)
Luxembourg	2 (2%)
Spain	2 (2%)
Switzerland	2 (2%)
Czech Republic	1 (1%)
Denmark	1 (1%)
Japan	1 (1%)
Korea South	1 (1%)
New Zealand	1 (1%)
Singapore	1 (1%)
South Africa	1 (1%)
Sweden	1 (1%)
Tunisia	1 (1%)
Turkey	1 (1%)

Table C.2: For how many years have you been using ROS?

	Nr. of answers
Less than 1 year	19 (17%)
1 - 3 years	22 (20%)
3 - 5 years	23 (20%)
5 - 10 years	25 (22%)
More than 10 years	14 (12%)

Table C.3: What is the main context of the previous projects in which you have used ROS? Multiple answers were accepted.

	Nr. of answers
Academic	**84 (82%)**
Industrial	49 (47%)
Hobbies	3 (3%)
Student competition	2 (2%)

Table C.4: What do you consider to be your level of proficiency with ROS?

	Nr. of answers
Beginner	8 (8%)
Limited experience	22 (21%)
Experient User	27 (26%)
Proficient Developer	35 (34%)
Expert	11 (11%)

Table C.5: "What are the most important features in a Robotics Middleware?", "What makes ROS 1 a good choice when developing robotics applications?" and "In what areas do you think ROS 1 is not very strong?"

	Important in Robotics Middleware	ROS 1 is strong in	ROS 1 is weak in
	Nr. of answers		
Open source	65 (63%)	**66 (64%)**	2 (2%)
Reliability	60 (59%)	30 (29%)	28 (27%)
Modularity	58 (56%)	**51 (49%)**	14 (14%)
(Good) documentation resources	53 (51%)	50 (48%)	27 (26%)
Robustness	44 (43%)	20 (19%)	**31 (30%)**
(Good) integration with visualization tools	42 (41%)	**58 (56%)**	2 (2%)
Out-of-the-box driver support for commonly used sensors	37 (36%)	**52 (50%)**	7 (7%)
Easy to use	35 (34%)	43 (42%)	20 (19%)
(Good) integration with robotic simulators	34 (33%)	**51 (49%)**	5 (5%)
Distributed Architecture	21 (20%)	28 (27%)	22 (21%)
Command Line Interface tools	21 (20%)	38 (37%)	9 (9%)
Support for Multi-Robot Systems	19 (18%)	14 (14%)	**45 (44%)**
Easy to understand	18 (17%)	29 (28%)	18 (17%)
Support for different platforms	17 (16%)	15 (15%)	25 (24%)
Multi-language support	16 (15%)	28 (27%)	16 (15%)
Security	5 (5%)	0	**59 (57%)**
(Good) integration with web tools	4 (4%)	3 (3%)	19 (18%)

Table C.6: "What are the most important features in a Robotics Middleware?", "What makes ROS 1 a good choice when developing robotics applications?" and "In what areas do you think ROS 1 is not very strong?"(only considering the 3 highest levels of expertise).

	Important in Robotics Middleware	ROS 1 is strong in	ROS 1 is weak in
	Nr. of answers		
Open source	45 (62%)	47 (64%)	1 (1%)
Reliability	45 (62%)	22 (30%)	19 (26%)
Modularity	44 (60%)	40 (55%)	8 (11%)
(Good) documentation resources	53 (51%)	38 (52%)	19 (26%)
Robustness	31 (42%)	14 (19%)	23 (32%)
(Good) integration with visualization tools	35 (48%)	43 (59%)	2 (3%)
Out-of-the-box driver support for commonly used sensors	30 (41%)	38 (52%)	6 (8%)
Easy to use	21 (29%)	33 (45%)	11 (15%)
(Good) integration with robotic simulators	22 (30%)	35 (48%)	3 (4%)
Distributed Architecture	16 (22%)	23 (32%)	16 (22%)
Command Line Interface tools	20 (27%)	32 (44%)	5 (7%)
Support for Multi-Robot Systems	11 (15%)	7 (10%)	36 (49%)
Easy to understand	10 (14%)	19 (26%)	10 (14%)
Support for different platforms	12 (16%)	9 (12%)	21 (29%)
Multi-language support	12 (26%)	22 (30%)	10 (14%)
Security	4 (5%)	0	48 (66%)
(Good) integration with web tools	3 (4%)	2 (3%)	15 (21%)

Table C.7: How do you feel about ROS 1 Documentation and Tutorials?

	Mean score[1] (Rating)
Helpful	3.88 (Good)
Informative	3.81 (Good)
Not confusing	3.57 (Good)
Easy to understand	3.35 (Neutral)
Broad	3.07 (Neutral)
Well organized	2.80 (Neutral)
Detailed	2.70 (Neutral)

Table C.8: Which of these ROS 2 features have you heard about before?.

	Nr. of answers
Data Distribution Service (DDS)	**76 (78%)**
Absence of ROS master	**76 (78%)**
Python Launch files	67 (68%)
Support for real-time systems	65 (66%)
Support for Windows and macOS	62 (63%)
Quality of Service Policies	55 (56%)
Security enabled by DDS	47 (48%)
Support for small embedded platforms	46 (47%)
Lifecycle Nodes	43 (44%)
None	2 (2%)

Table C.9: What is keeping you from fully adopting ROS 2?

	Never used ROS 2	Has used ROS 2	Total
Dependency on ROS 1 packages	**23 (56%)**	**14 (64%)**	**37 (59%)**
Not urgent	**27 (66%)**	8 (36%)	**35 (56%)**
Lack of ROS 1 packages migrated to ROS 2	**19 (46%)**	**15 (68%)**	**34 (54%)**
I don't think ROS 2 is in a "ready" state	13 (32%)	6 (27%)	19 (30%)
Conformism	8 (19%)	2 (9%)	10 (16%)
Waiting for the End of life of ROS 1 Noetic	6 (15%)	3 (14%)	9 (14%)
Lack of resources / time to migrate	2 (5%)	2 (9%)	4 (6%)

Table C.11: What do you think is keeping other developers from fully adopting ROS 2?

	Never used ROS 2	Has used ROS 2	Using ROS 2	Total
Dependency on ROS 1 packages	**25 (60%)**	**16 (73%)**	**24 (69%)**	**65 (66%)**
Lack of ROS 1 packages migrated to ROS 2	**22 (54%)**	**16 (73%)**	**30 (86%)**	**68 (69%)**
They don't think ROS 2 is in a "ready" state	**23 (56%)**	**12 (56%)**	**27 (77%)**	**62 (63%)**
Not urgent	**21 (51%)**	10 (45%)	**22 (63%)**	**53 (54%)**
Waiting for the End of life of ROS 1 Noetic	15 (37%)	6 (27%)	4 (11%)	25 (26%)
Conformism	11 (27%)	6 (27%)	5 (14%)	22 (22%)
Lack of documentation	-	-	3 (9%)	3 (3%)

[1] [1 - 1.5] Not at all;]1.5 - 2.5] Not so good;]2.5 - 3.5] Neutral;]3.5 - 4.5] Good;]4.5 - 5] Very good.

Table C.10: Detailed results to What is keeping you from adopting ROS 2? (according to developers' expertise)

	Beginner	Limited Experience	Experient user	Proficient developer	Expert
Waiting for the End of life of ROS 1 Noetic	1 (25%)	3 (16%)	2 (12%)	2 (11%)	1 (14%)
Not urgent	1 (25%)	8 (42%)	12 (71%)	9 (47%)	4 (57%)
Lack of ROS 1 packages migrated to ROS 2	1 (25%)	9 (47%)	8 (47%)	12 (63%)	4 (57%)
I don't think ROS 2 is in a "ready" state	1 (25%)	4 (21%)	4 (24%)	5 (26%)	5 (71%)
Conformism	1 (25%)	1 (5%)	2 (12%)	3 (16%)	0%
Dependency on ROS 1 packages	1 (25%)	12 (63%)	10 (59%)	11 (58%)	4 (57%)

Table C.12: How do you feel about ROS 2 Documentation and Tutorials?

	Mean score[2] (Rating)
Helpful	3.38 (Neutral)
Informative	3.35 (Neutral)
Easy to understand	3.16 (Neutral)
Not Confusing	2.89 (Neutral)
Well organized	2.88 (Neutral)
Detailed	2.83 (Neutral)
Broad	2.30 (Not so good)

Table C.13: How do you compare ROS 1 with ROS 2, in terms of its . . .

	Mean score (Rating)
Capabilities for production use	3.98 (ROS 2 is better)
Features	3.40 (ROS 2 is better)
Ease of development	2.69 (ROS 1 is better)
Documentation	2.59 (ROS 1 is better)
Learning curve	2.51 (ROS 1 is better)
Tutorials	2.37 (ROS 1 is better)

[2] [1 - 1.5] Not at all;]1.5 - 2.5] Not so good;]2.5 - 3.5] Neutral;]3.5 - 4.5] Good;]4.5 - 5] Very good.

Table C.14: Detailed results for the question: How essential/important/compelling/useful do you find these ROS 2 features? (Mean scores 1-Not at all 5-Very much

	Important	Compelling	Useful	Essential
DDS	3.71	3.61	3.70	3.44
DDS Security	3.76	3.59	3.68	3.51
Quality of Service Policies	3.82	3.80	3.97	3.63
Absence of ROS master	3.88	4.06	3.99	3.62
Python launch files	3.34	3.77	3.99	3.07
Lifecycle Nodes	3.47	3.60	3.62	3.39
ROS 2 support for real-time systems	4.07	4.11	4.14	3.94
Embedded systems	3.77	3.96	4.01	3.66
ROS 2 on Windows	2.64	2.78	3.03	2.49
ROS 2 on macOS	2.48	2.66	2.92	2.40
ROS 1 bags in a ROS 2 environment	4.02	4.06	4.36	3.88
ROS 1 and ROS 2 nodes running simultaneously	3.78	3.88	4.06	3.65

Table C.15: Detailed to the question: How relevant are the these new ROS 2 features to Multi-Robot Systems? (Mean scores 1-Not at all 5-Very much)

	Relevancy
DDS	3.97
DDS Security.	3.55
QoS Policies.	4.00
Abscence of ROS Master	4.27
Python Launch files.	2.91
Lifecycle Nodes.	3.43
Real-Time systems.	3.43
Embedded Systems.	3.36
ROS 2 on Windows.	2.16
ROS 2 on macOS.	2.08
Common core library (rcl).	3.48
ROS 1 bags in ROS 2.	3.67
ROS 1 and ROS 2 nodes running simultaneously.	3.89

Appendix D

Difference files from the migration to ROS 2

In this appendix, we present difference files for package metadata files (`CMakeLists.txt` and `package.xml`) that were migrated from ROS 1 to ROS 2. Red colored lines represent code that was part of the ROS 1 version that is no longer used in the ROS 2 version. Green colored lines, contrarily, represent code that was added in the ROS 2 version. Lastly, black lines represent code that was left unchanged.

Listing D.1: CMakeLists.txt difference file

```
- cmake_minimum_required(VERSION 2.8.3)
- project(patrolling_sim)
+ cmake_minimum_required(VERSION 3.5)
+ project(patrolling_sim_ros2)
---
- find_package(catkin REQUIRED COMPONENTS
-   actionlib
-   move_base_msgs
-   nav_msgs
-   roscpp
-   roslib
-   tf
-   message_generation
- )
+ find_package(ament_cmake REQUIRED)
+ find_package(rclcpp REQUIRED)
+ find_package(rclcpp_action REQUIRED)
+ find_package(nav2_msgs REQUIRED)
+ find_package(nav_msgs REQUIRED)
+ find_package(tf2 REQUIRED)
+ find_package(tf2_geometry_msgs REQUIRED)
+ find_package(tf2_ros REQUIRED)
+ find_package(ament_index_cpp REQUIRED)
+ find_package(std_msgs REQUIRED)
+ find_package(rosidl_default_generators REQUIRED)
---
- include_directories(
-   src
-   ${catkin_INCLUDE_DIRS}
- )
+ include_directories(src)

add_library(PatrolAgent
            src/PatrolAgent.cpp
            src/getgraph.cpp
            src/algorithms.cpp
            src/config.cpp)

add_library(SSIPatrolAgent
            src/SSIPatrolAgent.cpp)

+ ament_target_dependencies(PatrolAgent "rclcpp" "ament_index_cpp"
+   "nav2_msgs" "rclcpp_action" "tf2_ros" "nav_msgs"
+   "tf2_geometry_msgs" "tf2" "patrolling_sim_msgs")
+ ament_target_dependencies(SSIPatrolAgent "rclcpp" "nav2_msgs"
+   "patrolling_sim_msgs")
---
add_executable(Conscientious_Reactive
  src/Conscientious_Reactive.cpp)
- target_link_libraries(Conscientious_Reactive
-   PatrolAgent ${catkin_LIBRARIES})
+ ament_target_dependencies(Conscientious_Reactive rclcpp
+   rclcpp_action nav2_msgs tf2_ros nav_msgs)
+ target_link_libraries(Conscientious_Reactive PatrolAgent)

add_executable(Heuristic_Conscientious_Reactive
  src/Heuristic_Conscientious_Reactive.cpp)
- target_link_libraries(Heuristic_Conscientious_Reactive
-   PatrolAgent ${catkin_LIBRARIES})
+ ament_target_dependencies(Heuristic_Conscientious_Reactive rclcpp
+   rclcpp_action nav2_msgs tf2_ros nav_msgs)
+ target_link_libraries(Heuristic_Conscientious_Reactive PatrolAgent)

add_executable(Conscientious_Cognitive
  src/Conscientious_Cognitive.cpp)
- target_link_libraries(Conscientious_Cognitive
-   PatrolAgent ${catkin_LIBRARIES})
+ ament_target_dependencies(Conscientious_Cognitive rclcpp
+   rclcpp_action nav2_msgs tf2_ros nav_msgs)
+ target_link_libraries(Conscientious_Cognitive PatrolAgent)

add_executable(Cyclic
  src/Cyclic.cpp)
- target_link_libraries(Cyclic
-   PatrolAgent ${catkin_LIBRARIES})
+ ament_target_dependencies(Cyclic rclcpp
+   rclcpp_action nav2_msgs tf2_ros nav_msgs)
+ target_link_libraries(Cyclic PatrolAgent)

add_executable(MSP
  src/MSP.cpp)
- target_link_libraries(MSP
-   PatrolAgent ${catkin_LIBRARIES})
+ ament_target_dependencies(MSP rclcpp
+   rclcpp_action nav2_msgs tf2_ros nav_msgs)
+ target_link_libraries(MSP PatrolAgent)

add_executable(GBS
  src/GBS.cpp)
- target_link_libraries(GBS
-   PatrolAgent ${catkin_LIBRARIES})
+ ament_target_dependencies(GBS rclcpp
+   rclcpp_action nav2_msgs tf2_ros nav_msgs)
+ target_link_libraries(GBS PatrolAgent)
```

```

add_executable(SEBS
  src/SEBS.cpp)
- target_link_libraries(SEBS
-   PatrolAgent ${catkin_LIBRARIES})
+ ament_target_dependencies(SEBS rclcpp
+   rclcpp_action nav2_msgs tf2_ros nav_msgs)
+ target_link_libraries(SEBS PatrolAgent)

add_executable(CBLS
  src/CBLS.cpp)
- target_link_libraries(CBLS
-   PatrolAgent ${catkin_LIBRARIES})
+ ament_target_dependencies(CBLS rclcpp
+   rclcpp_action nav2_msgs tf2_ros nav_msgs)
+ target_link_libraries(CBLS PatrolAgent)

add_executable(DTAGreedy
  src/DTAGreedy.cpp)
- target_link_libraries(DTAGreedy
-   PatrolAgent ${catkin_LIBRARIES})
+ ament_target_dependencies(DTAGreedy rclcpp
+   rclcpp_action nav2_msgs tf2_ros nav_msgs)
+ target_link_libraries(DTAGreedy PatrolAgent)

add_executable(DTASSI
  src/DTASSI.cpp)
- target_link_libraries(DTASSI
-   SSIPatrolAgent ${catkin_LIBRARIES})
+ ament_target_dependencies(DTASSI rclcpp
+   rclcpp_action nav2_msgs tf2_ros nav_msgs)
+ target_link_libraries(DTASSI PatrolAgent SSIPatrolAgent)

add_executable(DTASSIPart
  src/DTASSIPart.cpp)
- target_link_libraries(DTASSIPart
-   SSIPatrolAgent ${catkin_LIBRARIES})
+ ament_target_dependencies(DTASSIPart rclcpp
+   rclcpp_action nav2_msgs tf2_ros nav_msgs)
+ target_link_libraries(DTASSIPart PatrolAgent SSIPatrolAgent)
---
- catkin_package(
-   INCLUDE_DIRS src
-   LIBRARIES PatrolAgent SSIPatrolAgent
- )
+ ament_export_dependencies(ament_cmake)
+ ament_export_dependencies(rclcpp)
+ ament_export_dependencies(rosidl_default_runtime)
+ ament_export_dependencies(nav_msgs)
+ ament_export_dependencies(rclcpp_action)
+ ament_export_dependencies(nav2_msgs)
+ ament_export_dependencies(tf2)
+ ament_export_dependencies(tf2_ros)
+ ament_export_dependencies(tf2_geometry_msgs)
+ ament_export_dependencies(std_msgs)

+ install(DIRECTORY launch DESTINATION share/${PROJECT_NAME})
+ install(TARGETS monitor Conscientious_Reactive
+   Heuristic_Conscientious_Reactive Conscientious_Cognitive
+   Cyclic MSP GBS SEBS CBLS Random DTAGreedy DTASSI
+   DTASSIPart GoToStartPos DESTINATION lib/${PROJECT_NAME})
+ install(DIRECTORY params DESTINATION share/${PROJECT_NAME})
+ install(DIRECTORY maps DESTINATION share/${PROJECT_NAME})
+ install(DIRECTORY rviz DESTINATION share/${PROJECT_NAME})
+ install(DIRECTORY worlds DESTINATION share/${PROJECT_NAME})
+ install(DIRECTORY description DESTINATION share/${PROJECT_NAME})
+ install(DIRECTORY MSP DESTINATION share/${PROJECT_NAME})

+ ament_package()
```

Listing D.2: package.xml difference file

```
- <?xml version="1.0"?>
- <package>
-  <name>patrolling_sim</name>
-  <version>2.2.4</version>
-  <description>Multi-Robot Patrolling Stage/ROS
-   Simulation Package.</description>
+ <package format="3">
+   <name>patrolling_sim_ros2</name>
+   <version>2.2.4</version>
+   <description>Multi-Robot Patrolling Stage/ROS2 Simulation
+    Package.</description>
  <license>BSD</license>
- <buildtool_depend>catkin</buildtool_depend>
+ <buildtool_depend>ament_cmake</buildtool_depend>

- <build_depend>actionlib</build_depend>
- <build_depend>move_base_msgs</build_depend>
- <build_depend>nav_msgs</build_depend>
- <build_depend>roscpp</build_depend>
- <build_depend>tf</build_depend>
- <build_depend>stage_ros</build_depend>
- <build_depend>roslib</build_depend>
- <build_depend>message_generation</build_depend>
+ <build_depend>rclcpp_action</build_depend>
+ <build_depend>nav2_msgs</build_depend>
+ <build_depend>nav_msgs</build_depend>
+ <build_depend>rclcpp</build_depend>
+ <build_depend>tf2</build_depend>
+ <build_depend>tf2_geometry_msgs</build_depend>
+ <build_depend>tf2_ros</build_depend>
+ <build_depend>stage_ros</build_depend>

- <run_depend>actionlib</run_depend>
- <run_depend>move_base_msgs</run_depend>
- <run_depend>nav_msgs</run_depend>
- <run_depend>roscpp</run_depend>
- <run_depend>tf</run_depend>
- <run_depend>stage_ros</run_depend>
- <run_depend>roslib</run_depend>
- <run_depend>message_runtime</run_depend>
+ <exec_depend>rclcpp_action</exec_depend>
+ <exec_depend>nav2_msgs</exec_depend>
+ <exec_depend>nav_msgs</exec_depend>
+ <exec_depend>rclcpp</exec_depend>
+ <exec_depend>tf2</exec_depend>
+ <exec_depend>tf2_geometry_msgs</exec_depend>
+ <exec_depend>tf2_ros</exec_depend>
+ <exec_depend>stage_ros</exec_depend>

  <export>
+   <build_type>ament_cmake</build_type>
  </export>
</package>
```

Bellow we present an example of a ROS 1 launch file written in XML, and its equivalent in ROS 2 written with the Python launch API.

Listing D.1: ROS 1 XML launch file to launch the Stage simulator with a custom map.

```
<?xml version="1.0" encoding="UTF-8" ?>
<launch>
  <arg name="map" default="grid" />
  <node name="stageros" pkg="stage_ros" type="stageros" args="$(find
   ↪  patrolling_sim)/maps/$(arg map)/$(arg map).world" output="screen" />
</launch>
```

Listing D.2: ROS 2 Python launch file to launch the Stage simulator with a custom map.

```python
import os
from launch_ros.actions import Node
from launch.actions import DeclareLaunchArgument, OpaqueFunction
from launch.substitutions import LaunchConfiguration
from launch import LaunchDescription
from ament_index_python.packages import get_package_share_directory

def launch_setup(context, *args, **kwargs):
  package_path = get_package_share_directory('patrolling_sim_ros2')
  map_name = LaunchConfiguration('map').perform(context)

  stage_node = Node(package='stage_ros',
    executable='stageros',
    arguments =
     ↪  [os.path.join(package_path,'maps',map_name,map_name+'.world')],
  )

return [
  stage_node,
]

def generate_launch_description():
  return LaunchDescription([
    DeclareLaunchArgument('map',default_value="grid"),
    OpaqueFunction(function=launch_setup)
  ])

generate_launch_description()
```

Appendix E

Detailed results for Chapter 5

Table E.1: Detailed results for the ROS 1 Ethernet latency experiment.

Nr.	Latency (ms)			
robots	Min	$\widetilde{X}$	$\overline{X} \pm \sigma$	Max
2	0.05	0.11	0.13±0.06	**0.30**
4	0.05	0.72	0.99±0.87	**3.60**
6	0.04	0.42	1.09±1.26	**4.27**
8	0.03	0.36	0.82±0.99	**4.01**
10	0.03	0.65	1.10±1.15	**5.09**

Table E.2: Detailed results for the ROS 2 Ethernet experiment.

Nr.	Latency (ms)			
robots	Min	$\widetilde{X}$	$\overline{X} \pm \sigma$	Max
2	0.11	0.19	0.19±0.03	**0.29**
4	0.04	0.10	0.12±0.05	**0.31**
6	0.04	0.07	0.08±0.04	**0.22**
8	0.03	0.06	0.08±0.05	**0.31**
10	0.04	0.12	0.13±0.10	**0.57**

Table E.3: Detailed results for the ROS 1 WiFi experiment.

Nr.	Latency (ms)			
robots	Min	$\widetilde{X}$	$\overline{X} \pm \sigma$	Max
2	2.11	2.89	**2.92±0.31**	3.76
4	4.62	5.87	**5.83±0.55**	7.29
6	7.58	9.25	**9.30±0.74**	11.42
8	10.87	13.35	**13.32±0.92**	15.72
10	14.77	18.01	**18.07±1.36**	21.71

Table E.4: Detailed results for the ROS 2 WiFi experiment.

Nr.	Latency (ms)			
robots	Min	$\widetilde{X}$	$\overline{X} \pm \sigma$	Max
2	0.81	1.18	**1.78±0.96**	3.48
4	0.55	1.23	**1.45±0.57**	3.86
6	0.64	4.50	**3.88±2.44**	8.42
8	0.64	5.14	**4.54±2.46**	9.36
10	0.52	4.39	**4.33±2.68**	11.52

Table E.5: Detailed results for the ROS 1 resource usage experiment (CPU).

Nr. robots	CPU Usage (%)			
	Min	$\widetilde{X}$	$\overline{X} \pm \sigma$	Max
1	3.60	11.35	**12.27±4.99**	26.50
2	7.70	14.10	**15.48±5.25**	32.80
4	5.20	15.98	**22.97±16.55**	98.50
6	9.10	21.10	**27.80±15.08**	96.00
8	22.30	30.35	**35.51±6.61**	58.90
10	25.10	36.70	**37.71±6.65**	61.50

Table E.6: Detailed results for the ROS 2 resource usage experiment (CPU).

Nr. robots	CPU Usage (%)			
	Min	$\widetilde{X}$	$\overline{X} \pm \sigma$	Max
1	10.80	19.80	**22.12±7.42**	47.80
2	16.10	23.40	**25.60±7.73**	57.60
4	22.50	42.25	**44.67±14.61**	98.00
6	46.00	54.50	**55.57±5.19**	74.60
8	52.50	61.90	**63.79±8.31**	100.00
10	60.00	71.25	**71.50±5.35**	83.40

Table E.7: Detailed results for the ROS 1 resource usage experiment (RAM).

Nr. robots	RAM Usage (%)			
	Min	$\widetilde{X}$	$\overline{X} \pm \sigma$	Max
1	20.60	20.70	20.75±0.16	20.90
2	21.30	21.45	21.45±0.09	21.60
4	22.70	22.90	22.88±0.10	23.00
6	24.00	24.30	24.29±0.12	24.50
8	25.60	25.70	25.77±0.11	25.90
10	27.10	27.25	27.26±0.10	27.40

Table E.8: Detailed results for the ROS 2 resource usage experiment (RAM).

Nr. robots	RAM Usage (%)			
	Min	$\widetilde{X}$	$\overline{X} \pm \sigma$	Max
1	20.60	20.70	20.70±0.10	20.90
2	21.70	21.80	21.83±0.09	22.00
4	24.90	25.20	25.17±0.13	25.40
6	29.90	30.10	30.13±0.14	30.40
8	35.70	35.90	35.92±0.15	36.20
10	51.30	51.40	51.44±0.13	51.70