FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Web-based user interface development for multiple robots interaction

**João Cerqueira**

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Pedro Gomes da Costa

July 31, 2023

# Resumo

O desenvolvimento de interfaces de utilizador para robôs tem sido historicamente limitado em termos de usabilidade e acessibilidade. Antes da criação de frameworks que auxiliam os especialistas em robótica a criar aplicações para robôs, o processo de desenvolvimento era complexo e exigia um amplo conhecimento tanto na arquitetura de baixo nível do robô como nos detalhes de alto nível da interface de utilizador. Estas frameworks vieram, de certa forma, resolver estes desafios fornecendo uma camada de abstração acima do hardware e do software de baixo nível do robô, tornando mais fácil para os programadores criarem aplicações. No entanto, ainda era necessário um nível significativo de conhecimento para criar aplicações usando estas *frameworks*. O *rosbridge* e a comunidade de Robot Web Tools têm sido úteis na resolução deste problema, integrando tecnologias da web no campo da robótica.

Esta dissertação explora o desenvolvimento de uma interface *web* para monitorizar e controlar robôs moveis, aproveitando as vantages oferecidas pela integração das tecnologias *web*. Esta investigação explora o estado-da-arte das interfaces para robôs, estuda a integração do *rosbridge* e avalia o impacto das tecnologias web na usabilidade e acessibilidade da interface desenvolvida. Por fim, ilustra o processo de desenvolvimento da aplicação *web* e compara-o com as interfaces previamente desenvolvidas dentro do projeto especifico no qual esta dissertação se insere. Os resultados destacam o potencial das tecnologias *web* para melhorar a interação humano-robô, aproveitando as *frameworks* da *web* para aprimorar o projeto e mitigar as limitações associadas às interfaces de robôs tradicionais.

# Abstract

The development of robot user interfaces has historically been limited in terms of usability and accessibility. Prior to the creation of frameworks that assist roboticists in creating robot applications, the development process was complex and required a wide range of knowledge in both low-level robot architecture and high-level user interface details. These frameworks addressed these challenges by providing an abstraction layer on top of the robot hardware and low-level software, making it easier for developers to create applications. However, a significant level of knowledge was still required to create applications using these frameworks. rosbridge and the Robot Web Tools community have been helpful in addressing this issue by integrating web technologies into the robotics field.

This dissertation delves into the development of a web interface to monitor and control mobile robots, utilizing the advantages offered by the integration of web technologies. This research examines the state-of-the-art in robot interfaces, explores the integration of rosbridge and evaluates the impact of web technologies on the usability and accessibility of the developed interface. Lastly, it illustrates the step-by-step process of the web application development and compares it to the previously developed interfaces within the specific project in which this dissertation is inserted. The results highlight the potential of web technologies to enhance human-robot interaction by leveraging web frameworks to improve the overall project and mitigate the limitations associated with traditional robot interfaces.

# Acknowledgements

I want to take a moment to express my heartfelt gratitude to the incredible individuals who have been instrumental in the completion of this dissertation. Their unwavering support, guidance and encouragement have been invaluable throughout this challenging journey.

First and foremost, I want to express my deep appreciation to my supervisor, Pedro Gomes da Costa. His unwavering belief in my potential has been immeasurable in guiding me through this research endeavour. His invaluable feedback, thought-provoking discussions and tireless commitment to excellence have shaped not only this dissertation but also my growth as a researcher.

I am equally grateful to my company supervisor, Diogo Matos. His practical expertise, trust in my abilities and continuous guidance have provided a strong foundation for the practical application of my research. I would also like to give special recognition to Paulo Rebelo, whose extensive knowledge and dedication have consistently contributed to both the development and testing of the dissertation project, greatly enhancing its quality and success. Additionally, I extend my appreciation to all the remarkable individuals I had the privilege of cooperating with at the company, especially Héber Sobreira, for his invaluable assistance and support throughout the entire project.

In this moment of reflection, I cannot overlook the immeasurable support of my family, friends and my girlfriend Débora. Their unwavering belief in my potential, encouragement and support have kept me motivated and determined to overcome challenges. I am deeply grateful for their presence in my life and I am fortunate to have such a strong support that has stood by me throughout this journey.

In conclusion, I am grateful to everyone who has supported me throughout this process. This work would not have been possible without their contribution.

João Cerqueira

# Contents

**Resumo** **i**

**Abstract** **iii**

**1 Introduction** **1**
1.1 Context . . . 1
1.2 Goals . . . 2
1.3 Structure of the Document . . . 2

**2 Background and Fundamental Aspects** **5**
2.1 ROS . . . 5
2.2 Robot Web Tools / rosbridge . . . 8
2.2.1 Robot Web Tools . . . 8
2.2.2 rosbridge . . . 9
2.2.3 rosjs / roslibjs . . . 9
2.3 Web programming languages . . . 11
2.3.1 HTML . . . 11
2.3.2 CSS . . . 11
2.3.3 JavaScript . . . 12
2.3.4 JavaScript frameworks/libraries comparison . . . 14
2.4 Software Application Architectures . . . 18
2.4.1 Monolithic Architecture . . . 19
2.4.2 Microservices Architecture . . . 19
2.4.3 Client-Server Architecture . . . 19

**3 Bibliography Review** **21**
3.1 Existing Interfaces for Robots . . . 21
3.2 Overview of the Project . . . 26

**4 System Setup and Data Acquisition** **31**
4.1 Robot Connection Setup . . . 31
4.2 Technology Stack and Application Architecture . . . 32
4.3 Application Requirements . . . 34
4.4 Selection of ROS Topics and Services for the Robots' Data Acquisition . . . 35
4.5 REST API . . . 39
4.5.1 Configuration Variables . . . 39
4.5.2 Initialization of the Server . . . 39
4.5.3 Connection to the ROS Master . . . 40
4.5.4 Retrieval and Organization of Data coming from the ROS Master . . . 41

4.5.5 Offline Detection Process . . . . . 42
4.5.6 Endpoints of the REST API . . . . . 43

**5 User Interface** **45**
5.1 Data Retrieval . . . . . 45
5.2 File Organization and Hierarchy . . . . . 47
5.3 Representation of the Data . . . . . 48
5.3.1 Structure of the User Interface . . . . . 48
5.3.2 Initialization of the Interface . . . . . 49
5.3.3 Conversion of Coordinate Systems . . . . . 50
5.3.4 Drawing the Map . . . . . 51
5.3.5 Drawing the Robot . . . . . 52
5.3.6 Displaying the Actions Queue . . . . . 54
5.3.7 Displaying the Obstacle Detection . . . . . 55
5.3.8 Displaying the Robot Status . . . . . 55
5.4 Service Invocation and Robot Interaction . . . . . 57
5.4.1 Trajectory Orders to the Robot . . . . . 58
5.4.2 Cancellation of Orders . . . . . 59
5.4.3 Pause and Resume the Robot Activity . . . . . 60

**6 Results and Discussion** **63**
6.1 Qualitative Analysis . . . . . 63
6.2 Feature Analysis . . . . . 65
6.3 Quantitative Analysis . . . . . 66
6.3.1 Time Analysis for Drawing the Robot . . . . . 66
6.3.2 Time Analysis for Issuing an Order . . . . . 68

**7 Conclusions** **71**

**A Videos** **73**
A.1 Demonstration Video . . . . . 73

**References** **75**

# List of Figures

2.1 Synchronous message passing - services . . . . . 6
2.2 Asynchronous message passing - topics . . . . . 7
2.3 rosbridge: an abstraction layer between the application and middleware/kernel layers. Adapted from [1] . . . . . 9
2.4 Performance scalability on architectures with six logical cores. Reprinted from [2] 13
2.5 Most popular web frameworks and technologies. Reprinted from [3] . . . . . 14

3.1 Home Page of the ASV's GUI. Reprinted from [4] . . . . . 22
3.2 PR2 Remote Lab interface. Reprinted from [5] . . . . . 23
3.3 Drawing a square with robot movements. Reprinted from [6] . . . . . 24
3.4 The three interfaces developed in [7] . . . . . 25
3.5 Software infrastructure of the application. Reprinted from [8] . . . . . 26
3.6 Screenshot of the already existing interface built with RViz . . . . . 27
3.7 Iris Interface . . . . . 28
3.8 Screenshot of the already existing web interface . . . . . 29

4.1 Application's projected architecture . . . . . 33
4.2 List of message types from the Map Data Topics . . . . . 36
4.3 List of message types from the Robot Data Topics . . . . . 37
4.4 List of Request and Response Objects from the Services . . . . . 38
4.5 Process Model of the Connection to the ROS Master . . . . . 40
4.6 Process Model of the Offline Status Detection . . . . . 42

5.1 Process of the retrieval of the data from the REST API . . . . . 46
5.2 Organization of the User Interface Directory . . . . . 47
5.3 User Interface Structure . . . . . 48
5.4 Process model of the interface initialization . . . . . 49
5.5 Transformation from the Map coordination system to the Canvas coordinate system 51
5.6 Representation of the Map and the Robot in the User Interface . . . . . 53
5.7 Representation of the Actions Queue for each robot . . . . . 54
5.8 Obstacle Detection Representation . . . . . 55
5.9 Robots Status Representation . . . . . 56
5.10 Representation of the main interface . . . . . 56
5.11 Representation of the simple(secondary) interface . . . . . 57
5.12 Representation of the menu to issue a trajectory order . . . . . 58
5.13 Trajectory Order to the Robot without cargo . . . . . 58
5.14 Representation of the menu to issue a trajectory order with cargo . . . . . 59
5.15 Trajectory Order to the Robot with cargo . . . . . 59
5.16 Process of Cancelling of One Order . . . . . 60

5.17 Process of Cancelling All Orders . . . . . . . . . . . . . . . . . . . . . . . . . . 60
5.18 Process of Resuming the Robot Activity . . . . . . . . . . . . . . . . . . . . . . . . 60
5.19 Process of Pausing the Robot Activity . . . . . . . . . . . . . . . . . . . . . . . . 61

6.1 Time analysis for drawing the robot in the old interface . . . . . . . . . . . . . . . 67
6.2 Time analysis for drawing the robot in the new interface . . . . . . . . . . . . . . . 67
6.3 Time analysis for issuing an order in the old interface . . . . . . . . . . . . . . . . 69
6.4 Time analysis for issuing an order in the new interface . . . . . . . . . . . . . . . 69

# List of Tables

2.1 Frameworks’ production build sizes. Adapted from [9] . . . . . . . . . . . . . . . 17
2.2 Frameworks loading speeds. Adapted from [9] . . . . . . . . . . . . . . . . . . . . 17

6.1 Interfaces’ set of Features . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65

# Abbreviations and Acronyms

| | |
|---|---|
| AGV | Automated Guided Vehicle |
| API | Application Programming Interface |
| ASV | Autonomous Surface Vehicle |
| CSS | Cascading Style Sheets |
| CRUD | Create Read Update Delete |
| DDS | Data Distribution Service |
| DOM | Document Object Model |
| HTML | Hypertext Markup Language |
| HTTP | Hypertext Transfer Protocol |
| IP | Internet Protocol |
| JSON | JavaScript Object Notation |
| OOP | Object-Oriented Programming |
| OS | Operating System |
| PHP | Hypertext Preprocessor |
| REST | Representational State Transfer |
| ROS | Robot Operating System |
| RPN | Robotic Programming Network |
| RWT | Robot Web Tools |
| SGML | Standard Generalized Markup Language |
| SLU | Spatial Language Understanding |
| SOAP | Simple Object Access Protocol |
| TCP | Transmission Control Protocol |
| URL | Uniform Resource Locator |
| URDF | Unified Robot Description Format |
| VPL | Visual Programming Language |
| XML | Extensible Markup Language |

# Chapter 1

# Introduction

## 1.1 Context

The field of robotics has been rapidly growing in the past years and it has become an important part of our society. As technology continues to evolve, and with the need to increase efficiency and precision in the various types of industry in the modern world, the integration of robots in the industry has become more of a necessity than an extra feature as it was a few years ago.

This fast evolution had a major impact on the complexity of robots and consequently, on the level of expertise in order to develop one. It became unrealistic for a single researcher to have the capacity to create all the software necessary for a robot as this requires a big degree of knowledge, from the lowest level of the robot (hardware and its drivers) to the highest (concerning the user interface).

To solve this problem, several efforts were made to create frameworks that provide the robot developer with an abstract layer to allow the researcher to focus on the high-level software without worrying about all the low-level hardware programming. Within these frameworks, nowadays the Robot Operating System (ROS) [10] is often considered the *"de facto standard middleware"* for developing robot applications [11].

Although this work field has evolved over the years, the learning curve of these frameworks is still relatively high and it still requires deep knowledge of the peculiarities of the device's low-level operation systems. This lack of accessibility, usability and adaptability left the human-to-robot interaction area almost exclusive to specialists well-versed in the complexities of the used framework.

One innovative way of solving these weaknesses was the integration of ROS with web technologies, which is a sector inside the robotics work field that is earning its space really fast.

The rise in popularity of the web in the last years is bringing many developers and even users who would not necessarily call themselves programmers to take a shot in work positions related to this area. Furthermore, as one of its main advantages is its accessibility (due to the fact that a lot of its technologies are available to everyone who owns a personal computer, a tablet, or even a smartphone), this area has gained special attention among other work fields.

The integration of robotics with web technologies made it possible to improve many of the weaknesses of the former and expand the reach of the development of robot systems to many more people besides the already specialized low-level software developers while putting together people's creativity and reasoning and robots' efficiency. Additionally, the connection between these two technologies allowed the decrease of the necessary cognitive load of the worker who interacts with the machine due to the high usability of websites. For employers, there's also an advantage in lowering the costs of purchasing devices that will interact with the machines, since web technologies can be accessed through basically every device used currently in companies.

Within the context of the project in which this dissertation is inserted, certain limitations have been identified, particularly concerning accessibility and usability, as discussed in the following chapters. To overcome these challenges, we will leverage the integration of web technologies, which offer a vast array of possibilities. By incorporating these technologies into our project, we are on the right path to making significant contributions to enhance the overall project and effectively mitigate the limitations it has encountered so thus far.

## 1.2 Goals

In this dissertation, to tackle the problem related to the necessity of having a high level of knowledge to interact with a robot, a human-to-robot interface will be developed to try to bridge the gap between the user, with little to no knowledge about the low-level software of the machine, and its developer.

The developed interface will be able to interact with a multi-robot system operating with ROS, by sending orders and commands (via *rosbridge*, a protocol that lets ROS and the web communicate with each other using *WebSockets*). This interface should satisfy some predefined goals in **configuration** - such as the creation of the occupation map as well as the identification of its workstations and the definition of the possible trajectories - and in **operation and monitorization** - with the monitorization of the AGV's state in 2D with info about its position, battery and the task being executed, the report of the robot's problems via messages to the interface and the creation of tasks for the robots. Furthermore, its **scalability** will be enough to interact with different types of robots, in different scenarios and, with the help of web technologies, be accessible through many devices via a web browser.

Additionally, this interface will aim, beyond doing its tasks efficiently, to be as responsive as possible. This will create a great user experience since the user will interact with the robot with little to no latency, resulting in real-time interaction.

## 1.3 Structure of the Document

The document is structured in a logical sequence to provide a comprehensive understanding of the research made, the methodology applied and the resulting findings, followed by a thorough discussion. The following is an overview of the document's structure:

- **Background and Fundamental Aspects:** This section offers a comprehensive review of the technologies that are already implemented in the ongoing project, as well as potential integration options. It provides an overview of the background and fundamental aspects related to these technologies, including their functionalities, capabilities and relevance to our project.

- **Bibliography Review:** In this chapter, a thorough review of the relevant literature is conducted to establish the research's foundation. It explores existing interfaces developed by others and serves as a basis to understand the technologies utilized in those interfaces. This review helps identify potential improvements that can be made to our project. Furthermore, it provides a detailed context of our project by analyzing the current status of our project, particularly in terms of developed interfaces, and highlighting the limitations of the overall project.

- **System Setup and Data Acquisition:** This chapter focuses on the setup of the necessary software to enable the development of our web application. It provides a detailed explanation of the chosen technologies and architecture for the web application, along with the steps taken to implement this architecture. The chapter provides a step-by-step walkthrough through the entire process, starting from the setup and initialization of the application and leading up to the retrieval of data. These events culminate in the subsequent chapter, where the data representation is presented.

- **User Interface:** This chapter provides an extensive explanation of the development process for the user interface. It offers a comprehensive guide that covers the entire workflow, starting from the data retrieval stage and leading to its representation in a user-friendly manner.

- **Results and Discussion:** This chapter focuses on the outcomes of the interface development for the overall project. It emphasizes the advantages and disadvantages of our web application in comparison to the existing interfaces. The results are analysed, both qualitatively and quantitatively, exploring the strengths and weaknesses of our interface and discussing their implications for the project.

- **Conclusion:** This chapter offers a concise summary and conclusion to the dissertation, highlighting the key elements of the project. It includes the main findings, contributions and implications of our web application into the overall project, providing an overview of the dissertation's core aspects.

This structure aims to provide a logical and sequential flow, enabling users to navigate through each step involved in the development of this interface. By following this structured approach, readers can gain a comprehensive understanding of the entire development process, from the conceptualization phase to the final implementation. This sequential logic ensures that all the essential components and considerations are covered, providing a clear presentation of the interface development journey.

# Chapter 2

# Background and Fundamental Aspects

In this chapter, we expose some theoretical aspects and elemental concepts that are crucial for building a solid foundation of knowledge to understand the context of this dissertation.

## 2.1 ROS

In this section, we introduce the basic concepts of ROS and provide a general overview of their significance and role in the material covered later.

Robot Operating System (ROS), as the name already implies, is an operating system for robots. Just like other OS (like Windows, Linux or macOS), it was developed to manage all the hardware and software by creating an abstract layer between these two that helps to hide almost all the complexities of the low-level hardware of the equipment.

The creation of this tool had its emphasis especially on large-scale integrative robotics research, which made it usable in a wide variety of situations as robotic systems grew even more complex over time [10].

Aside from its main goal, ROS was designed to settle in five basic principles [10]:

- **Peer-to-peer**: a ROS-based system is composed of several processes connected in a peer-to-peer topology, possibly on different hosts.
- **Tools-based**: It is a microkernel design, in which several tools are used to build and run the various ROS components. This gives ROS almost complete modularity, which offers this framework significant advantages in terms of stability and complexity management.
- **Multi-lingual**: ROS was designed to be language-neutral. Despite this, the majority of applications developed on top of ROS use C++ (for efficiency) or Python (because of the relatively low learning curve of the language).
- **Thin**: It was made to encourage developers to leave all complexity in libraries, and only create small executables which expose these library functionalities to ROS.

- **Free and Open-Source**: It is a free and open-source platform which made it easier for debugging purposes since it involves many layers of software on robotic systems.

In terms of implementation, ROS is composed of a number of **nodes** (processes that perform computations), that communicate with each other using **messages**. The latter are data structures made up of typed fields, which can be standard primitive data types (integer, floating point, boolean, char, etc.) or arrays of primitives types. These can even include arbitrarily nested structures and arrays [10].

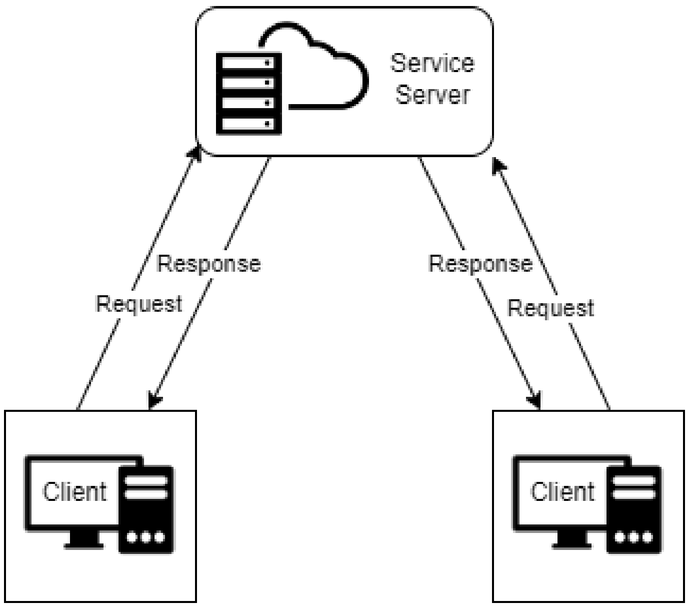


Figure 2.1: Synchronous message passing - services

There are two types of communication for nodes to send messages within the ROS framework:

- **synchronously**, which are called **services**, that work as a client-server request and response (the same way as it happens with web services, which are defined by URLs). Note that there can be many clients using the service server but one can only exist one server for a specific service, as we can see in Figure 2.1.

- **asynchronously** which works with a publish/subscribe network in which nodes publish or subscribe to **topics** depending on their interests [10]. Note that, there may be multiple concurrent publishers and subscribers for a single topic and a single node may publish and/or subscribe to multiple topics as we can see in Figure 2.2

Additionally, ROS provides many more features that have been making the framework more efficient and attractive to developers by improving its usability.

Of all the features added to ROS, one that stands out as the main differentiator from this framework to others is the capability to keep the system running with well-debugged nodes and even with nodes still being debugged [10].

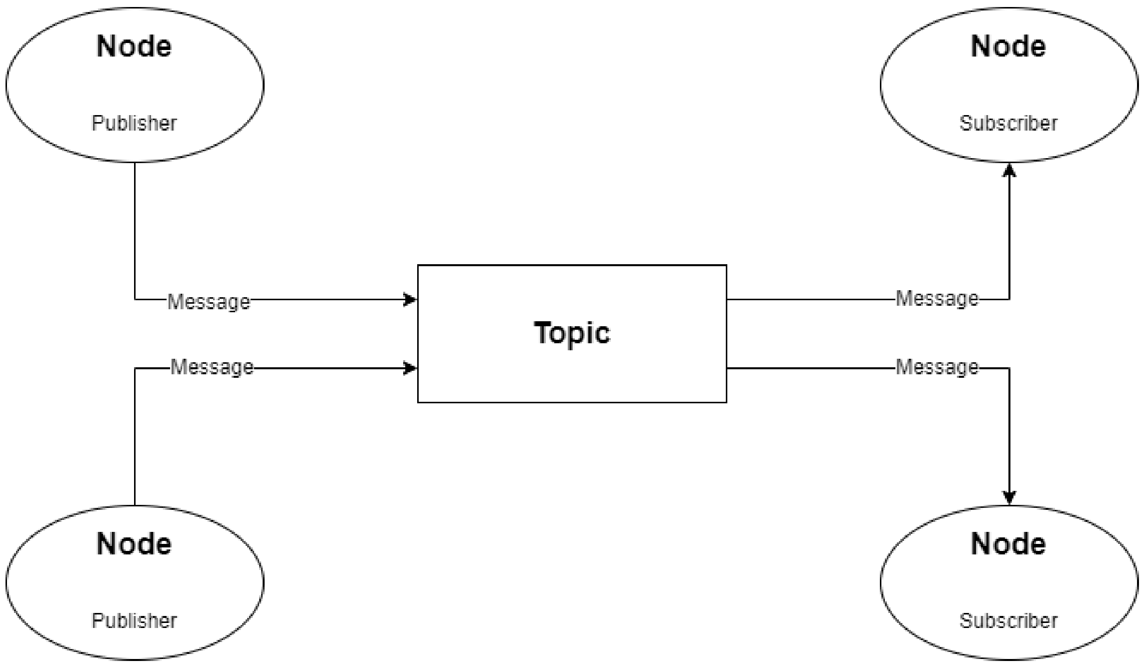


Figure 2.2: Asynchronous message passing - topics

Another peculiar characteristic of ROS is its transformation system, *tf*. This tool creates a transformation tree that states all the relations between the existing frames. This gives the user an automated and systematic update of this information [10].

An additional feature that contributes to the usability of this framework is the packaging system of ROS. This system gives the possibility of separating the software into many small chunks of code - *packages* - which allows the many parts of the code to be treated independently. Furthermore, there are some tools to manage these *packages*, namely *rospack*, used to query these packages by name and find their details, *rosmake*, used to build all the dependencies of the package and *roslaunch* to instantiate the correspondent graph - a set of nodes and their interactions [10].

On top of that, ROS also provides many tools for the visualization and monitoring of data to give some kind of feedback to the user in the form of graphs, plots or even schemas that represent nodes and their dependencies.

Despite achieving all these goals in its first version, ROS 1, its limitations became more apparent as it gained wider adoption in the industry. Because security, network topology and system up-time were not prioritized in ROS 1 [12], the framework was limited in several ways. For example, its only officially supported operating system was Ubuntu, a popular Linux distribution, it struggled to deliver data over lossy links and one of its major flaws in terms of security was having a single point of failure (*rosmaster*)[12]. These are just some examples from many that became visible through its wide utilization across the robotics work field.

The community tried to address these boundaries of the first version of ROS 1, but by changing the already existing framework, due to architectural aspects, it could become unstable which was a major concern, especially for the thousands of developers that were already using this version of ROS.

With this in mind, ROS 2 was developed from scratch to solve many of the issues that plagued the previous version. It is based on the Data Distribution Service (DDS), which allows ROS 2 to have features that were previously lacking, such as better security, embedded and real-time support, multi-robot communication and the ability for robots to thrive in non-ideal networking environments [12].

While ROS has made significant improvements in many areas, including those mentioned earlier (in both versions - ROS 1 and ROS 2), it still faces challenges regarding accessibility and usability. These setbacks are mainly due to the high level of knowledge required in the robot's low-level architecture to work with the ROS environment.

These flaws led researchers to try to find a solution for these problems, who, with the help of the rapidly evolving web community, could put together the complicated and inaccessible robot middleware world with the available to everyone and easy-to-use world of the web.

## 2.2 Robot Web Tools / rosbridge

In this section, we provide a brief description of the creation of Robot Web Tools, including the reasons for its development and the technologies on which it is based.

### 2.2.1 Robot Web Tools

To put together the best of both worlds (robot middleware and web and network technologies), the Robot Web Tools project was created to allow users/developers to have an accessible environment for creating and interacting with human-to-robot interfaces. Moreover, it has allowed not only the industry but also education, to move towards a more accessible way of controlling robots with the creation of remote labs (as it will be explained later) by enabling the access of robots through the internet.

RWT takes advantage of being an open-source, and always developing project to evolve based on network protocols and client libraries [11]. Moreover, this is not a project exclusive to ROS, since it operates over a variety of network transports [11].

This project, unlike ROS, was designed to be a client-server architecture, which takes advantage of having the same topology as the web technologies. This architecture enables clients and servers to speak the same language through messages, which has been the way the web has been evolving throughout these years. Beyond this, this paradigm allows broadcast and stream-oriented services (common to publish-subscribe in ROS) to be supported in this design [11].

### 2.2.2 rosbridge

*"At the heart of Robot Web Tools is rosbridge"* [11], a protocol that works as an *"access point for ROS"* [13]. This is due to the fact that allows non-ROS clients to publish and subscribe to ROS topics and invoke ROS services [13].

This protocol works as an abstraction layer on top of ROS [11]. *rosbridge*, as we can see in Figure 2.3, works as an intermediate between ROS and the application layer, by defining boundaries between the layers in the software development process [11]. It separates the low-level interaction with the robot (via ROS) and the high-level software development for the application. This enables different developers, with different skill sets, to work, collaboratively, for the same purpose. Note that *rosbridge* does not work exclusively with ROS clients or any specific programming language.

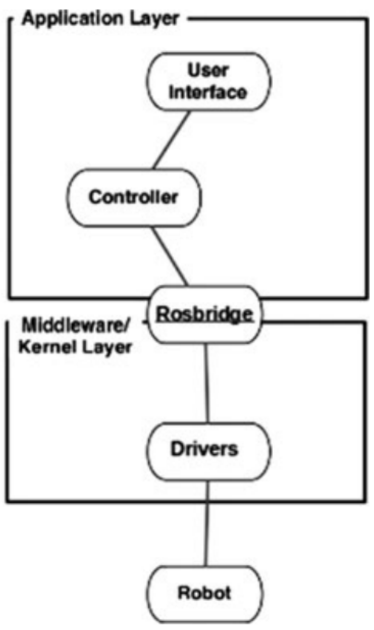


Figure 2.3: rosbridge: an abstraction layer between the application and middleware/kernel layers. Adapted from [1]

One of the reasons that make *rosbridge* so useful for the development of robot applications is its use of WebSockets for communication purposes. This communication is done via TCP sockets where ROS data is transmitted in JSON (JavaScript Object Notation) format. This feature allows *rosbridge* clients to be language-independent as long as they use any programming language that supports WebSockets [13]. It also makes the web browser a perfect tool for building robot applications since WebSockets were initially designed to work in web servers and browsers.

### 2.2.3 rosjs / roslibjs

As a way of achieving one of the main goals of *rosbridge* - expand the development of robot applications to as many users/developers as possible (and not only roboticists) -, a JavaScript library that uses this protocol to communicate with ROS applications was created and given the name ***rosjs*** (nowadays called ***roslibjs***). This library allowed developers to connect browsers to ROS applications (that could be hosted on a remote server) using just JavaScript code.

In order to facilitate its integration with other web applications, *rosjs* was designed to be lightweight. Furthermore, its event-based paradigm makes it possible to emit an event based on the robots' feedback, while being responsive. [13]

This library allows the communication of ROS topics through JSON formatted messages, via WebSockets, as already mentioned before. Additionally, in order to take away some of the developer's workload and make its job easier and more accessible, *rosjs* includes some utilities such as transform clients, URDF (Unified Robot Description Format) parsers, and simple matrix/vector operations [11].

Now, some features of *rosjs* will be explained in great detail in order to get to know all of his strengths in developing robot web applications:

- allows the browser to **expose ROS services and topics**: a *rosjs* object must be created to interact with the robot, which includes not only commands to publish and subscribe to ROS topics but also a function to interact with ROS services. [14]
- it also provides many services (similar to ROS services) that are only accessible within *rosjs* such as: displaying current available topics and services; listing the topics the application is subscribed to; accessing the type of message of the topic or service; accessing objects associated with the type of a requested topic/service; handling key authentication for *rosjs*. [14].
- it provides **protected services/topics** and **key authorization** for the developer. Protected services/topics are utilized to avoid the user to alter some critical aspects of the robot. On the other hand, key authorization works as an authentication feature, which limits access for certain users to the robot application. [14]

Although *rosjs* has all these features, some were left outside the scope of this library, such as visualization tools. This lack of visualization dependencies was intentional since it makes *rosjs* a lightweight library and capable of being integrated with JavaScript applications together with many other libraries.

This limitation of *rosjs* can be complemented not only with already developed general JavaScript libraries (such as *EaselJS* or *three.js*) or with libraries that were developed specifically for ROS interaction (such as *ros2djs*, built on top of *EaselJS* and *ros3djs* built on top of *three.js*). [11] It is also compatible with server-side JavaScript environments such as Node.js (including even a library called *rosNode.js* for that purpose).

Since this library was designed for JavaScript and web technologies, it allows the robot application developer to create an interface in which the end-user will not need to install any third-party software or plugin, as it only requires a browser to interact with it.

## 2.3 Web programming languages

With all the growth of the web community, many things have been changing in the past years in this area. Despite that, the basic languages of web programming - HTML, CSS and JavaScript - still remain the same after all these years of evolution.

In this section, we present a brief description of those languages.

### 2.3.1 HTML

HTML (Hypertext Markup Language) is a markup language used to define the structure of web content [15]. Nowadays this is the standard web programming language to build web pages and applications.

In its launch, HTML was a simple markup language to structure documents. Based on SGML (Standard Generalized Mark-up Language), like the latter it included elements such as titles, paragraphs, headers and unordered and ordered lists. The main differentiator from SGML, at the time, was the incorporation of hypertext links with the *href* attribute [16].

Throughout many years a large number of developers tried improving the standard language while launching new versions of it like HTML 2, HTML 3.2 and HTML 4.01 and adding features still used nowadays like tables, applets, text flow around images, subscripts and superscripts, buttons, frames, objects, classes, ids, etc [16].

The latest specification of this language is HTML5 and its launch represented huge progress in the web community. With many features added to the language (like the e-mail attribute, the audio and video tags, the header and footer tags, the Canvas API, the Web Workers API and many others), it solved limitations of the previous releases and improved the language by making it more efficient and even more sufficient. The Canvas API and the Web Workers API will be studied in greater detail in the sections below because of their many applications in robot web applications.

### 2.3.2 CSS

Cascading Style Sheets (CSS) is a stylesheet language used to style a structured document written in HTML or XML [17].

Just like HTML, CSS is a standardized language across web browsers which makes them the perfect tools to build web pages on top of them. Moreover, since CSS was created in an attempt to separate the already existing styling options of the HTML from the structure of the document, it makes the combination of these two a great combo for building web pages.

CSS allows the developer to style every piece of code written in HTML by using tags, ids or classes to identify each one. Additionally, it helps HTML to position and structure the individual contents of the web page more precisely.

### 2.3.3 JavaScript

JavaScript is the main scripting language for web pages. Currently, all the major browsers support JavaScript and there are also server-side applications like Node.js or Apache CouchDB that use it as well [18].

Just like almost any other object-oriented programming (OOP) language, JavaScript includes the four pillars of OOP: abstraction, inheritance, encapsulation and polymorphism. This allows the language to perform many tasks that other well-known OOP languages (like Java or C++) do, with its advantages and disadvantages.

JavaScript can be used, together with HTML and CSS, to make web pages dynamic by allowing JavaScript functions to interact with the DOM (Document Object Model) of the web page. Furthermore, it can make use of the many libraries already created (like the ones mentioned above for robotics) to reach more fields outside of web development.

#### 2.3.3.1 Canvas API

One of the most popular APIs added in the HTML5 launch was the Canvas API. It allowed users to draw any type of graphics using the HTML <*canvas*> element together with JavaScript [19].

By being focused mainly on 2D graphics, this API provides the developer with many predefined methods for drawing text, lines and shapes, applying styles and colours, using images and applying transformations. With all these functions, this API nails some use cases that were very difficult to do without it before, such as animations, real-time video processing, interactive graphics and other visualizations [19].

Despite the API being capable of doing all these things, it is not always simple to use, which led to the development of many libraries to use this API in order to create canvas-based projects more efficiently [19].

Out of all of the use cases that this API fills out, the web application that is going to be developed suits perfectly in these because of its necessity of having real-time processing and representation of the location and status of the robot, making the API an essential tool for the development of the application.

#### 2.3.3.2 Web Sockets API

This API is the key for communication with the ROS environment as rosbridge only communicates via WebSockets.

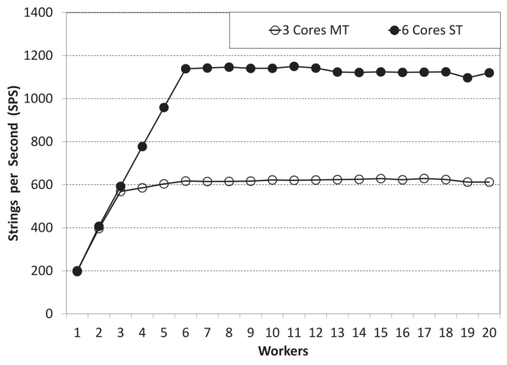
The WebSocket API allows the user to open a two-way interactive communication session between the browser and ROS, in this case [20]. Because of this, both the asynchronous and synchronous methods of message passing of ROS topics are achievable by using this API, which makes it indispensable for the development of the application.

#### 2.3.3.3 Web Workers API

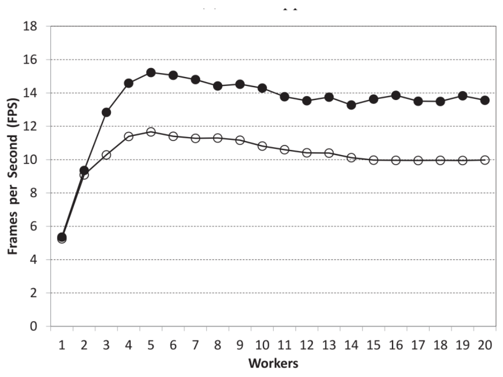
Another main feature that was added to HTML in the HTML5 launch was the Web Workers API. This API was a huge evolution in the web community as it solved a disadvantage of JavaScript to other languages: the impossibility of having multi threads working concurrently.

Web Workers API allows the developer to create workers (threads) that execute a piece of code in the background, separately from the execution of the main thread. This gave developers the possibility of having laborious tasks in the background without interrupting or slowing down the flow of the main thread. Furthermore, in these background workers, it is possible to run almost any JavaScript code that the user already uses in the main thread, with the exception of manipulating the DOM or using some methods of the *window* object [21].

The increase in performance can be seen in Figure 2.4 in which we can see that, whether for asynchronous (Figure 2.4a) or synchronous workers (Figure 2.4b), the performance of the web applications that were tested in this study increases proportionally to the number of workers while the latter is not higher than the number of logical cores. When the number of asynchronous workers running is higher than the number of logical cores, the performance does not degrade. On the other hand, the use of more synchronous workers than logical cores slightly degrades the performance [2]. From this, we can conclude that the ideal number of workers to use depends not only on the number of logical cores but also on the way that these workers will be used.



(a) Performance of HashApp - asynchronous workers. Reprinted from [2]



(b) Performance of RayApp - synchronous workers. Reprinted from [2]

Figure 2.4: Performance scalability on architectures with six logical cores. Reprinted from [2]

Despite all these improvements in the efficiency of web pages, there are still some problems with the usage of the Web Workers API.

The greatest disadvantage of the usage of this API is the impossibility of background workers sending messages directly to each other. They can only communicate with each other indirectly, by using the main thread as an intermediate. Together with this limitation, there is also the problem that workers cannot directly understand replies to messages unless is used a flag or some hard-coded parameter into the message. The third major issue with the background workers is that this API provides no construct for these workers to block while waiting for another message, for example [22].

Despite all these limitations, the use of the Web Workers API will surely have a positive impact on the development of the web application.

### 2.3.4 JavaScript frameworks/libraries comparison

With the increase in complexity of web applications, JavaScript alone is becoming not sufficient as it can make the developer's work very tedious and inefficient.

To solve this problem, many JavaScript libraries and frameworks were developed in the last years. These allowed developers to use and reuse pre-built components or functions to create more complex applications while making it possible to develop applications more efficiently and less time-consuming (avoiding the excess of time used to do commons tasks that can be simplified by the framework/library).

Nowadays, the terms "libraries" and "frameworks" tend to get used interchangeably, which is not correct. The main difference between these two is the fact that a framework provides a structure for how the code should be organized and, on the other hand, a library, by just providing the developers with predefined methods and classes allows for total control on how the code should be structured. In simple words, by using a library, the programmer is able to call functions and classes wherever he wants, while, by using a framework, this gives the developer locations to put in code but only executes this code when needed [23].

According to an annual survey made in May 2022 to the Stack Overflow community, as you can see in Figure 2.5, the most commonly used web framework at the time was Node.js followed by React.js, which was the most used framework in the previous year. This list has also the names of jQuery, Express, Vue and Angular in its top six [3].

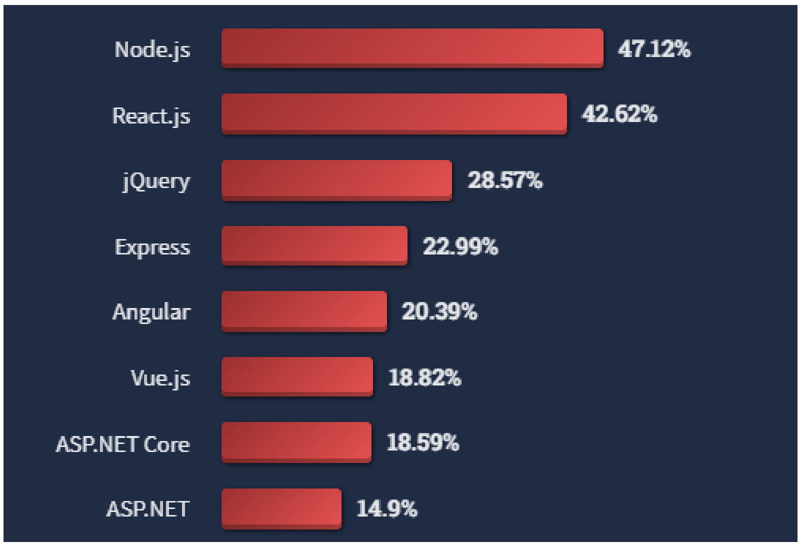


Figure 2.5: Most popular web frameworks and technologies. Reprinted from [3]

Every framework/library listed in this survey was developed to answer a specific set of flaws of the main language. This led each one to be suited to specific use cases.

In the next section, we will firstly present the already mentioned frameworks or libraries that are specifically used for front-end and compare them in order to find if there is a most suitable option for the web application to be built instead of using just plain JavaScript without the use of any framework or library. Then we will present some JavaScript libraries/frameworks that can be used in the back-end to support all the back office structure of the application, such as the communication with the robot operating system (via the *roslibjs* library) and the control of the relevant variables to the interface.

#### 2.3.4.1 Front-End JavaScript Libraries / Frameworks

**React.js**

React (also known as ReactJS or React.js) is a front-end JavaScript library. It is currently maintained by Meta which makes it fully available as a free and open-source tool.

React is a declarative programming language, which allows the developer to design views for each state of the application and renders just the right components when your data changes [24]. This is an advantage for single-page web applications as it makes the pages far more efficient.

The base of this library is its component-based paradigm - it allows the developer to create its own components, which are reusable, and manage their own state. The library takes advantage of the use of components by allowing its logic to be written in JavaScript, which makes it easier to render elements in the DOM (Document Object Model) [24].

In essence, the utilization of this library nowadays brings many advantages to the web development paradigm and this is covered by the fact that it is currently the number one front-end library in use. One of its features that enabled its fastly growing usage in the web development industry was the fact that it can be used with many other technologies, whether in the front or back-end of the applications.

**Angular**

Angular is a free and open-source front-end JavaScript framework developed and maintained by Google. It's known as Angular 2+ or Angular 2 because it was based on AngularJS, a previous framework which was completely rewritten to become what nowadays is called Angular, the second full version of the framework.

Similarly to React.js, it is component-based, in which each component includes a TypeScript class, an HTML template (an element that declares how that component renders) and styles [25].

The fact that Angular was designed by Google to be a single-page application development gives it an advantage over other frameworks in this web application case. It has the advantage that the navigation through web pages does not necessarily require the whole page to be reloaded, by only reloading some specific components.

**Vue.js**

Vue.js, like the other two already mentioned, is an open-source front-end JavaScript framework, developed by Evan You, a former Google employee.

Vue, just like React, follows a declarative programming paradigm in a way that it allows the developers to describe HTML output depending on the JavaScript state. Another main function of this framework is its reactivity in automatically tracking JavaScript state changes to update the DOM when changes occur [26].

According to the authors, Vue is a progressive framework, in a way that despite covering most of the common features needed in front-end development nowadays, it is designed to be flexible and adaptable to follow all advances and things to come in web technologies [26]. This has been of the main reasons for the rise in popularity of Vue in the last few years.

To compare these frameworks, in order to find the most suitable option for the development of the robot web application, we did some research in order to know the various use cases of these frameworks, their performance for specific applications, their learning curve and their pros and cons.

As said before, all of these frameworks were developed to cover any flaw in the plain JavaScript coding and this led the frameworks to have very different use cases. Nowadays, Angular is more used to develop or maintain large-scale, real-time and scalable applications. In contrast to the latter, Vue is commonly employed in lightweight and intuitive applications, whether React is more suited to cross-platform applications or to extend the functionalities of an existing application [27].

One of the most essential topics of comparison, in order to choose the most suited framework, is **performance**. To compare these three frameworks, Elar Saks [9] developed a small test application to create various tables and add and remove rows from the table using each one of the frameworks. Note that all the applications were tested in the same environment and with the same CSS file for styling and the same JavaScript, function to produce data. After all the tests, we could point out some facts:

- in terms of size, the Angular application had an approximately **25 times larger size** than the React and Vue applications as we can see in Table 2.1. This might be explained by the fact that Angular is built for large-scale apps, having an unnecessary overhead for these types of small applications.

- in terms of speed, as we can see in Table 2.2, **Vue performed better than the other two frameworks on all the tasks**, which makes it perfectly suited for small web applications. Between React and Angular, the former outperformed the latter on 6 out of 8 tasks which is not that impressive, since Angular has a far greater size than React.

Another main factor that we need to take into account is the **learning curve** of these frameworks as it could be time-consuming to learn a completely new framework. Out of these three frameworks, the one that has the steepest learning curve is Angular, due to its large scale and more complete code base. Among the other two, Vue is the one that has a smoother learning curve as it uses an easier syntax, focused on pure HTML, CSS and vanilla JavaScript [9].

| | |
|---|---|
| React | 587 KB |
| Vue | 624 KB |
| Angular | 15.7 MB |

Table 2.1: Frameworks' production build sizes. Adapted from [9]

| | **Angular** | **React** | **Vue** |
|---|---|---|---|
| Load Page | 403 (x 1.64) | 269 (x 1.09) | 246 |
| Create table with 1000 rows | 384 (x 1.66) | 420 (x 1.81) | 232 |
| Re-create table with 1000 rows | 331 (x 2.83) | 190 (x 1.62) | 117 |
| Add 1000 rows | 254 (x 1.37) | 370 (x 2.00) | 185 |
| Create table with 10 000 rows | 6361 (x 4.40) | 2866 (x 1.98) | 1447 |
| Re-create table with 10 000 rows | 7427 (x 13.58) | 1176 (x 2.15) | 547 |
| Add 1000 rows | 7623 (x 5.31) | 3854 (x 2.68) | 1436 |
| Remove all rows | 2196 (x 4.35) | 607 (x 1.20) | 505 |

Table 2.2: Frameworks loading speeds. Adapted from [9]

Considering all these factors, we can conclude that for the dimension of the robot application that will be developed, the most suitable frontend framework is Vue. This is due to the fact that this framework has the **smoothest learning curve**, which can save much more time in studying the framework. Moreover, its **high performance** in small applications as seen in Figure 2.2 together with the fact that the use of Vue is commonly used for lightweight applications is an advantage related to the others.

#### 2.3.4.2 Back-End JavaScript Libraries / Frameworks

**Node.js**

Node.js (or Node) is a server-side JavaScript environment which means it provides the user with a way of running JavaScript code without the need for a web browser. For that purpose, it uses Google's runtime implementation - the "V8" engine [28]. This engine is mainly written in C and C++ to ensure high performance and low memory consumption [28].

In contrast to today's modern environments that use a concurrency model in which threads are employed, Node's process relies on an asynchronous I/O eventing model [28]. This model is a perfect match with JavaScript because the latter supports event callbacks.

Furthermore, as we can see in 2.5, a great advantage in relation to other frameworks and programming languages is its popularity. The extensive use of this framework allows the creation of a community of developers that help each other and allows for a single developer to clear any doubts that he may have with all the content available on the internet about this framework.

This framework also allowed web developers to develop full-stack applications by only using JavaScript for both the back and front end of the application. This is a significant development, especially for small applications, because previously it was almost always necessary to have knowledge in more than one programming language to develop a full-stack application, even if the application was relatively small.

Finally, the community created behind Node.js has developed numerous libraries for or compatible with Node.js, such as *node-mysql* or *node-couchdb*, which are essential for establishing database connections, as well as Express, a framework that will be discussed further in the next subsection.

**Express**

Express is a minimal and flexible framework built on top of Node.js. As we can see in 2.5 is one of the most used JavaScript frameworks and the main reason for its extended use is the set of features that it provides the developer for the development of full-stack web applications.

One of its main features is the middleware, a property that deals with requests and responses through the use of middleware functions [29]. These middleware functions will be the foundation to handle events in the applications built on top of Node.

Express also adds a feature to Node.js that allows the web application to create *routes* whether to another web API or to the same one with a specified URL [29]. This allows web applications to have built-in functions for handling incoming HTTP requests such as GET, PUT, POST and DELETE requests. These methods give the app the possibility of handling Create Read Update Delete (CRUD) operations, which are essential when building applications that interact with other applications.

Ultimately, after analyzing all of these frameworks and considering their advantages and disadvantages, we can determine which frameworks are better suited for the development of the web application.

We will be focusing solely on back-end libraries/frameworks for this project, as our priority is to ensure that the application is functional and provides the necessary features to the user. Additionally, as the design is not the first concern of this application, we will not incorporate front-end libraries/frameworks. This will avoid unnecessary burdens during development and maintenance, as it would require learning those frameworks from scratch or already being proficient in the use of those frameworks for future maintenance. Moreover, integrating these frameworks may limit the control over the application by not allowing complete freedom in its customization.

## 2.4 Software Application Architectures

The selection of an architecture for a software application is one of the most fundamental decisions in the conceptual phase of the application. This decision carries significant implications throughout the entire life cycle of the application, including the development and maintenance phases.

Currently, there is a wide range of software application architectures available, each with its own set of advantages and disadvantages. Some examples of these options include monolithic, layered, event-driven, microkernel, microservices, and client-server architectures.

In this section, we will discuss some of these architectures that could be implemented in the development of this application, such as the monolithic, the microservices and the client-server architectures.

### 2.4.1 Monolithic Architecture

Monolithic architecture is the traditional way of building software applications. This architecture follows a paradigm in which the application is structured in a single code base that includes all services, allowing the developer to separate all its functionalities into classes, functions and namespaces [30].

This type of architecture has as its main disadvantage the fact that if a service fails or needs to be changed, the whole application has to be rebuilt and deployed, which makes it a hard task to scale the entire application.

### 2.4.2 Microservices Architecture

This disadvantage of the monolithic architecture led many companies, such as Amazon, Netflix and eBay to change their applications' architectures to a microservices paradigm [31]. This architecture allows the developer to have a distributed and modular application, in which its components work as independent entities communicating with each other using messages [32].

The only constraint imposed by this modularity and message-passing paradigm is the technology used for the communication between services (media, protocols, data encodings, etc.) but, despite that, it allowed the developer to freely choose the optimal resources (languages, frameworks, etc.) for the implementation of each microservice [32]. This led to several improvements in scalability and maintenance, which were the main limitations of the monolithic architecture.

### 2.4.3 Client-Server Architecture

The client-server architecture is widely used in network applications and distributed systems, where the systems consist of two entities: the client and the server, with a well-defined division of responsibility between each other. In this model, clients assume the role of consumers, who make requests to servers whether for services or information. The server, on the other hand, works as the producer by answering the demands made by the client [33]. There are many examples of the use of this architecture, for example, REST (Representational State Transfer) - where the clients interact with the RESTful APIs via HTTP requests - or SOAP (Simple Object Access Protocol), which works with HTTP requests as well.

This architecture allows the developer to have an intermediate scenario between the monolithic and the microservices architecture because it allows having a distributed system, unlike the monolithic approach. On the other hand, for small applications or for applications where the data can be retrieved from a single component of the system, it is better suited when compared with the microservices architecture.

Furthermore, one of its key advantages is that client-server systems can be distributed in different machines, i.e., the server could be hosted in a platform and the client could be on another one [33].

In conclusion, by examining the technology already being used in our project and exploring the various technologies that could bring value to our web application, we have gained a deeper understanding of their strengths, limitations and, most importantly, their suitability for our project.

Overall, the analysis conducted in this chapter has provided a clear understanding of the available technologies. By making informed choices based on the requirements of our application, we are well-positioned to develop an effective web application that enhances the overall project.

# Chapter 3

# Bibliography Review

The field of robotics has come a long way in the last few years, as roboticists have been making great advances in technology, especially with the integration of machine learning and artificial intelligence into robots. As a result of these advances, robots are now capable of doing tasks in almost every existing field of work and are being more and more used in areas like healthcare, education, customer service or even entertainment.

This rapidly growing integration of robots in many types of tasks has been making human-to-robot interaction a more and more discussed issue than ever before. The fact that robots are getting incessantly used in various industries or even in day-to-day tasks is forcing their user interface to be as simple, intuitive and accessible to everyone as ever.

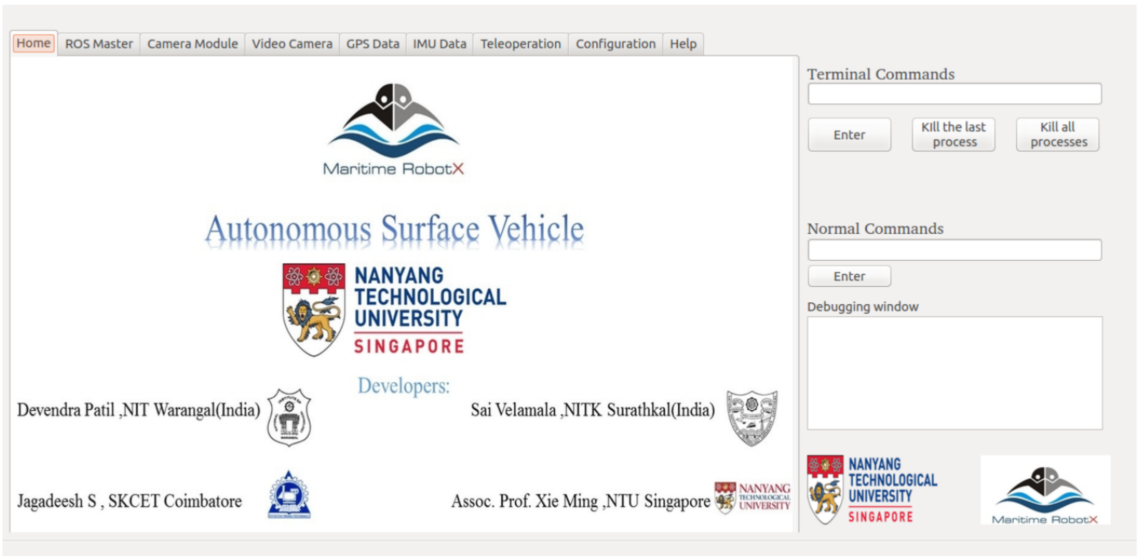
Although there has been huge progress in human-robot interaction through user interfaces evolving from the old physical buttons or switches to the recent digital GUIs (graphical user interfaces), there has still much more work to do in terms of usability and especially accessibility.

## 3.1 Existing Interfaces for Robots

In his section, we will present some interfaces already deployed in robots, whether for educational or industrial purposes, going through the technologies used in the development of these interfaces.

With the establishment of ROS as the standard middleware to build robot applications, an abstraction layer was created to allow developers to create user interfaces without the need to interact with all the low-level architecture of robots.

One of the main tools used along with ROS to develop these interfaces is Qt [34], a cross-platform framework to create graphical user interfaces. The Qt framework allows the developer to write reusable code, by providing a set of intuitive and modularized C++ library classes. The use of C++ makes it a perfect match for ROS as the latter has as his two main languages C++ and Python. [4] presents an interface, that was developed using Qt together with ROS, which enables the end user to interact with an autonomous surface vehicle (having ROS as his framework) with the help of graphical and visual elements. This interface simplifies the interaction between the robot and the end user by creating a simple environment for the latter to control and monitor the robot while hiding some technical aspects under the interface, namely by allowing the interface to have one terminal to execute several commands (as we can see in Figure 3.1 which presents the home page of the ASV's GUI), like the *roslaunch* one that is launched through Qt using the QProcess class (a class that is used to start and communicate with external programs) [4]. Like this interface, there are many others that make use of Qt and C++ in order to improve the human-robot interaction of a robot. On top of that, there are also examples of applications developed in Python, which take advantage of the fast expansion of this programming language together with its compatibility with ROS. One example is TurtleUI [35], a graphical user interface to control a TurtleBot2 that uses PyQt, a framework for the development of GUIs, that makes it possible to attach libraries from Qt to develop the interface. The purpose of this interface was to solve the problem of having to open several terminals to launch many commands by providing an interface with a built-in terminal.



Autonomous Surface Vehicle

Developers:

Devendra Patil ,NIT Warangal(India)

Sai Velamala ,NITK Surathkal(India)

Jagadeesh S , SKCET Coimbatore

Assoc. Prof. Xie Ming ,NTU Singapore

Figure 3.1: Home Page of the ASV's GUI. Reprinted from [4]

As we can see from the examples above, Qt can solve some of the shortcomings that ROS has in terms of accessibility and usability, however, it still doesn't provide the end user with direct access to the robot without a complex setup or the necessity of having some required software installed to run the robot as we see in the examples above - [4] requires the installation of ROS and some software related to Qt and a super complex setup by requiring the ground station PC and the computer onboard of the ASV to be on the same local network in order to establish a connection between the two and [35] also requires the installation of ROS and some software related to Qt as well.

To solve the shortcomings stated in the last paragraph, a new protocol was created to connect ROS and the robots with web technologies - *rosbridge*. Moreover, it became possible to build libraries for web programming languages (like *roslibjs* for JavaScript) in order to take advantage of the rapidly evolving web development work field.

Once developers got to work with these libraries and, consequently, put together ROS with web technologies (which, before, was a very difficult and tedious task), the world of user interfaces for robots saw huge advances in terms of accessibility. Not only allowed the user interfaces to be more accessible and usable to everyone but also allowed people with little to no experience in robotics to be able to develop web applications for robots.

One of the main applications provided by the use of these libraries was the development of specific interfaces for remote laboratories. Although remote access laboratories have been available before the arrival of the *rosbridge* framework - as we can see in [36], with the Telerobot, one of the first web-controlled devices on the internet, which fell short of expectations because of the inability of Java applets of providing a reliable platform for remote clients in a web browser, or the Telelabs, a framework for remote laboratories -, this protocol allowed the development of these laboratories in a much more usable way through web technologies. One example of a remote laboratory that allowed a larger and more diverse group of researchers to take part in the development of robot applications is the PR2 Remote Laboratory [5], which can be seen in Figure 3.2, a remote lab that lets its users interact with a PR2 robot whether by manual control, via a keyboard and a joystick, whether through a more automated control with the use of a point and click interface that allows the user to pick and place an object. Another example is RPN (Robotic Programming Network) [37] that offers the same capabilities in terms of remote access and accessibility as the PR2 Remote Laboratory but has a major difference: RPN scripting is done in Python while the PR2 is done in JavaScript. In short, these remote laboratories represent huge progress, not only for educational purposes, since it improves distance education (which is, nowadays, an even more talked about subject matter after the COVID-19 pandemic), but also for the various industries since it can reduce the cost of running laboratories.

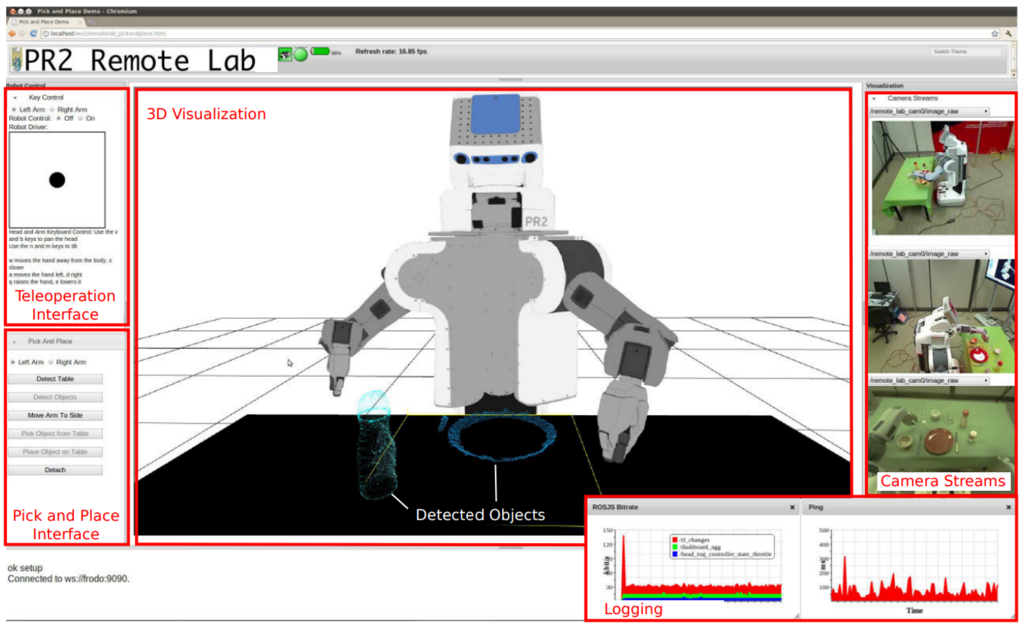


Figure 3.2: PR2 Remote Lab interface. Reprinted from [5]

Another use case explored by the use of the *rosbridge* protocol, which allows developers who are not roboticists to develop robot applications, is the creation of VPLs (Visual Programming Languages) to interact with robots, which provide its user with graphical interfaces for coding without the need of learning a new programming language syntax and semantics. An example of a VPL that uses rosbridge as a protocol is presented in [6], which is a VPL that provides a set of predefined blocks (of basic movements, conditions or even cycles) so that the user can build its own robot application, as we can see in Figure 3.3. The approach of creating building blocks in order to allow the user to create their own applications is not the focus of this dissertation (although it can be implemented in later stages of the development of the application) however, it goes to show that this has a great impact on spreading the development of robot applications to other developers that don't have experience in robotics.

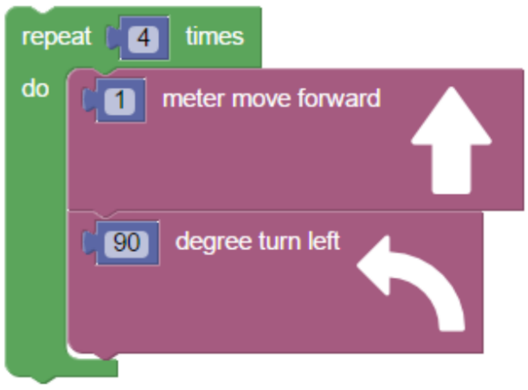
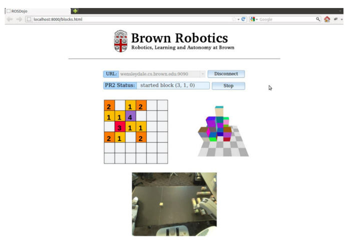
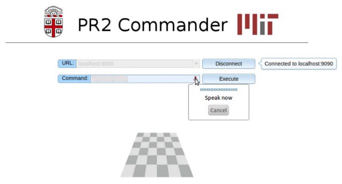


Figure 3.3: Drawing a square with robot movements. Reprinted from [6]

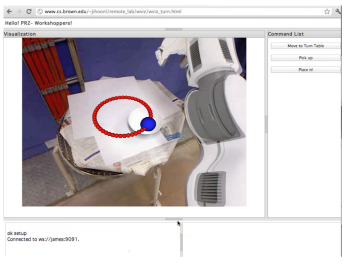
Now, the main use case for the *rosbridge* protocol has been the development of interfaces for controlling and monitoring robots through the web browser, without necessarily being involved in remote labs, but for industrial purposes. Through some research about the topic of web applications for controlling robots that use *rosbridge* as the communication protocol, some more interesting (for the sake of this dissertation) stood out.

One paper that had some examples was [7] which presented three interfaces to control three different PR2 robots: the first one, PR2 Castle Builder (shown in Figure 3.4a), lets the user design a "castle" made of cubes and converts this design in data and, consequently, in JavaScript code in order to communicate with the the Robot Operating System (ROS) through *rosbridge*; the second one, PR2 Commander (shown in Figure 3.4b), allows the user to type in a sentence or tell it through Google speech recognizer, which will be then decoded by MIT's Spatial Language Understanding (SLU) and the robot will try to perform the intended action, like "pick up the red block" (with obvious limitations to what the robot can do, which is picking up and dropping blocks); the last one, PR2 Turntable Manipulator (shown in Figure 3.4c), is a simple interface that lets the user choose when a robot manipulator should pick up a ball that is on top of a rotating table. This paper is very interesting to this dissertation because, although it doesn't present almost any technical issues about the interfaces, it shows that, by using web technologies, there are an infinite number of technologies attached to it (like the Google speech recognizer, the MIT Spatial Language or even the WebGL which is the tool that helps to do the 3D visualizer of the design input by the user in the PR2 Castle Builder) that can be easily accessed and used. This creates an almost unlimited set of options for features to add to the interface that is going to be controlling the robot.


Brown Robotics
Robotics, Learning and Autonomy at Brown
URL:
Disconnect
PR2 Status: started block (3, 1, 0)
Stop
(a) Web interface for PR2 Castle Builder. Reprinted from [7]



(b) Web interface for PR2 Commander. Reprinted from [7]



(c) Web interface for PR2 Turntable Manipulator. Reprinted from [7]

Figure 3.4: The three interfaces developed in [7]

Now that we have knowledge about the various possibilities of features that can be implemented just by the fact of using web technologies, there is a necessity of having some more technical knowledge of what is supporting some applications that use the *rosbridge* protocol. In [8] an interface to interact with a three-wheeled telepresence robot was developed. This interface is divided into two parts that work simultaneously: there is a manual control of the robot in which a joystick is used to move around the robot with the help of live video from a camera attached to the robot and there is an autonomous navigation that uses a point and click strategy - the user points and clicks to a specific location on the map, defines its orientation and the robot moves autonomously to the desired location. This paper, not only talks about the capabilities of the interface but also presents the software infrastructure beneath the interface and its communications with ROS, which are going to be very similar in terms of communications with the interface to be developed in this dissertation. As we can see in Figure 3.5, the robot uses ROS as its framework which is separated in four different modules. Then, ROS communicates with the web application via a server that has the *rosbridge* protocol which makes use of JavaScript's compatibility with WebSockets through a JavaScript library - *rosjs* - and the web_video_server package (which is used to display the live video captures by the camera of the robot). As already mentioned our application will follow a very similar structure but obviously with different ROS modules, different features and the use of a framework which, nowadays, represents an advantage in relation to the use of vanilla JavaScript since it turns the code much more efficient and organized.

Nowadays, there are some robot web applications that use frameworks like React or Angular. One example of an interface that uses React is [38], an interface that uses React as a front-end library and Node.js and Express together on the back-end of the application. Besides being a very complete application with much more features than the others already mentioned, this interface requires a deep knowledge of web technologies, specifically in three different frameworks/libraries which goes beyond the focus of this dissertation. That could even represent a weakness to the

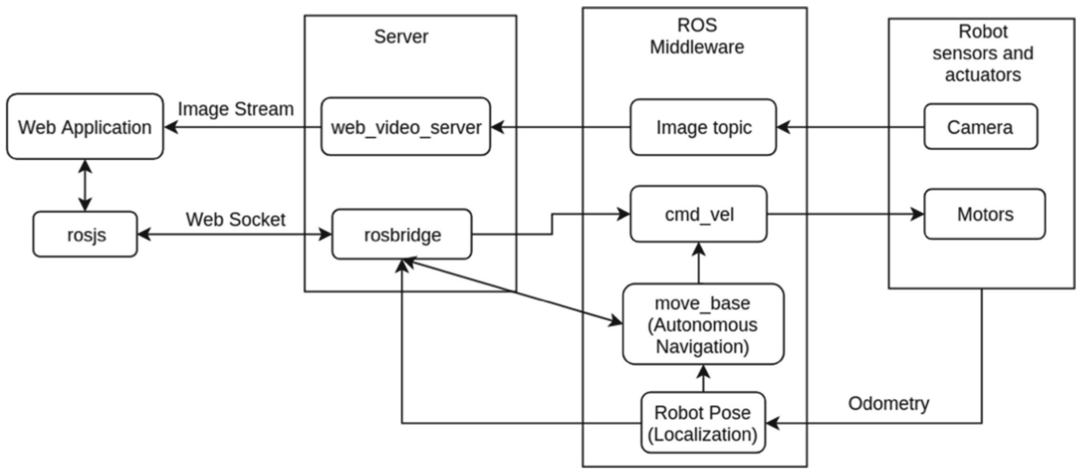


Figure 3.5: Software infrastructure of the application. Reprinted from [8]

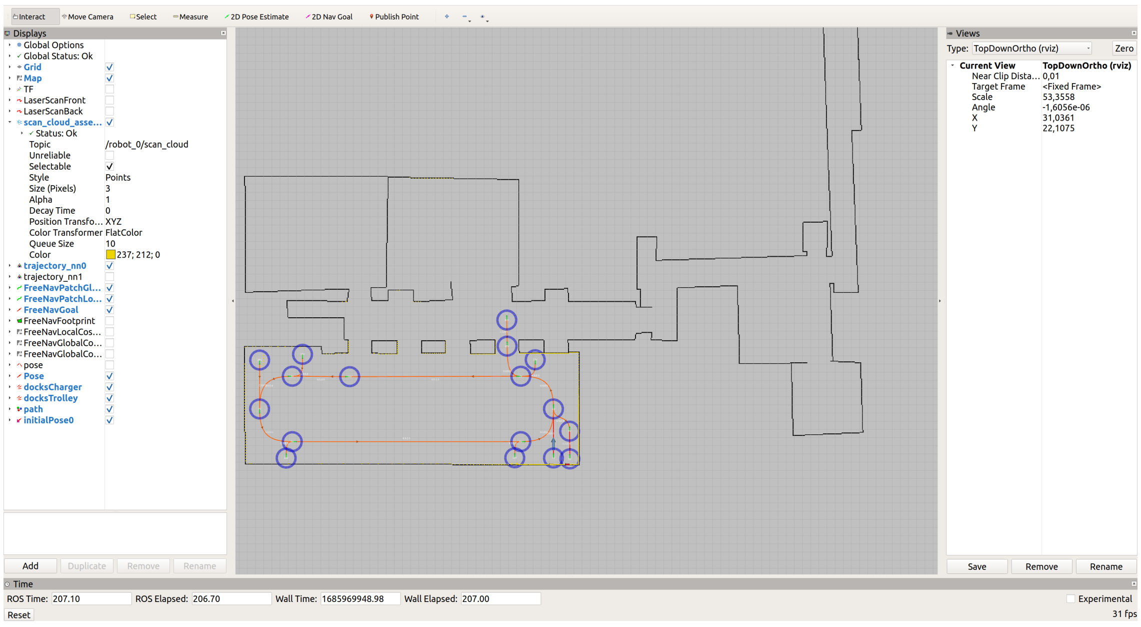
application since it requires a much more expert developer in web technologies in order to maintain the interface. Another interface that is built with React as its front-end library is [39], which although it has some interesting extra features like the login page, doesn't add much to what was presented previously. Despite using a library like React, which is nowadays the most popular front-end framework/library and having an advantage in terms of organization in relation to plain JavaScript, they have also all the disadvantages compared to Vue, namely those that were already stated in the last chapter that are the steeper learning curve and the worst performance. Moreover, as stated earlier, it requires a higher level of expertise in the web development area in order to maintain the applications.

## 3.2 Overview of the Project

In the scope of this project, all the software that allows for the control and monitorization of the robots is already developed, as mentioned before, through the use of ROS 1. On top of the navigation stack of the robots, there are already three interfaces that have provided valuable functionality and control capabilities to the project. Therefore, in this section, we will specify a little bit of the context of this project, the technology underneath the creation of these three interfaces and also the functionalities they bring to the project while pointing out some reasons why our web-based interface could prove to be a valuable asset for the project.

This ongoing project utilizes robots to pick up towers, which we will refer to as cargo, from an initial point and transport them to a final point based on input provided by the user. The primary objective of developing applications for this project revolves around facilitating and managing these actions, including monitoring the status of the action, the status of the robot and other relevant aspects.

The initial approach in developing an interface for the mobile robots in this project involved utilizing ROS, the robots' operating system, specifically RViz, a visualisation tool that is part of ROS. Figure 3.6 illustrates this interface which displays the robot's graph, namely the vertices (the designated stopping points for the robot, barring any obstacles) and the edges (the only paths that the robot could take to walk between the vertices). Additionally, the interface also allows the user to give orders to the robot to move to a certain vertice and to locate the robot on the map, in case the localization system fails. However, it should be noted that this is a very primitive interface offering limited features for the manipulation and monitorization of the robot.



| | |
|---|---|
| Global Options | |
| Global Status: Ok | |
| Grid | ✓ |
| Map | ✓ |
| TF | |
| LaserScanFront | |
| LaserScanBack | |
| scan_cloud_asse... | ✓ |
| Status: Ok | |
| Topic | /robot_0/scan_cloud |
| Unreliable | |
| Selectable | ✓ |
| Style | Points |
| Size (Pixels) | 3 |
| Alpha | 1 |
| Decay Time | 0 |
| Position Transfo... | XYZ |
| Color Transformer | FlatColor |
| Queue Size | 10 |
| Color | 🟨 237; 212; 0 |
| trajectory_nn0 | ✓ |
| trajectory_nn1 | |
| FreeNavPatchGl... | ✓ |
| FreeNavPatchLo... | ✓ |
| FreeNavGoal | ✓ |
| FreeNavFootprint | |
| FreeNavLocalCos... | |
| FreeNavGlobalCo... | |
| FreeNavGlobalCo... | |
| pose | |
| Pose | ✓ |
| docksCharger | ✓ |
| docksTrolley | ✓ |
| path | ✓ |
| initialPose0 | ✓ |

Type: TopDownOrtho (rviz) Zero

| | |
|---|---|
| **Current View** | **TopDownOrtho (rviz)** |
| Near Clip Dista... | 0,01 |
| Target Frame | <Fixed Frame> |
| Scale | 53,3558 |
| Angle | -1,6056e-06 |
| X | 31,0361 |
| Y | 22,1075 |

ROS Time: 207.10 ROS Elapsed: 206.70 Wall Time: 1685969948.98 Wall Elapsed: 207.00 Experimental
31 fps

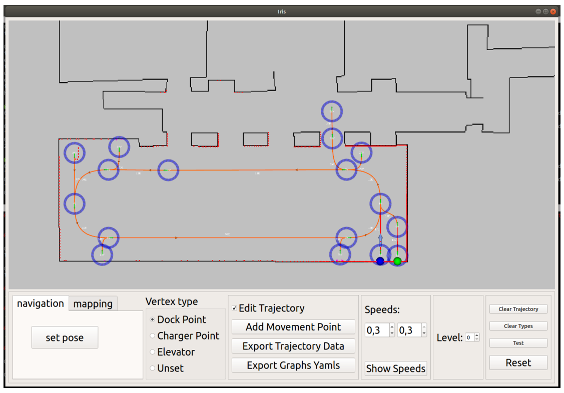
Figure 3.6: Screenshot of the already existing interface built with RViz

To improve the usability of the whole project, with the intention of adding new features and presenting more information to the user, a new interface was created, called Iris. The latter was created on top of RViz, by taking advantage of its open-access nature to incorporate new features alongside the existing ones, already provided by RViz.

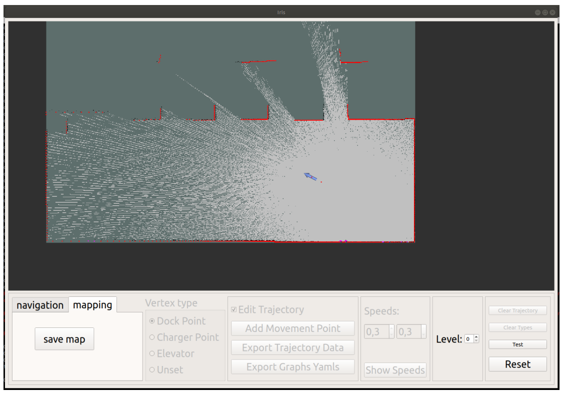
As depicted in Figure 3.7, we can observe that this interface, being built on top of RViz, incorporates all the functionalities included in the first interface while introducing numerous additional features that resulted in a huge improvement in terms of usability for the project. Some of these extra features include:

- the capability of enabling the mapping mode of the robot, allowing the user to observe real-time scanning of the environment with the help of the robot's lasers and sensors.
- the possibility of switching between the navigation and mapping modes.
- the option to switch between maps, allowing, for instance, the robot to transition between different floors.

- the potential of altering the graph data by easily adding or removing vertices and edges, and saving the configuration in a YAML file with the click of a button
- the capability of defining or changing the type of the vertices, allowing for designations such as elevator (for vertices involving floor changes) or docking stations.



(a) Screenshot of the Iris interface working in the Navigation mode



(b) Screenshot of the Iris interface working in the Mapping mode

Figure 3.7: Iris Interface

By analysing these interfaces, we concluded that there were huge flaws in terms of the representation of the robot in the navigation mode. Both the accessibility and the freedom of representation of the data coming from the robot were very limited to the constraints imposed by C++ and RViz. However, by incorporating web technologies into this project, the possibilities of representing the robot's data are greatly expanded. Additionally, as mentioned earlier, this integration brings a new level of accessibility to the project, enabling, for example, the application to be used on different devices than the ones launching the executables to control the robot.

In addition to the RViz-based interfaces discussed earlier, a new interface was developed using web technologies to address the accessibility and usability limitations of the project.

Figure 3.8 showcases this interface, which offers an enhanced user experience compared to the previous interfaces. It provides a visual representation of the robot's movement within the environment map and displays the queue of associated orders. Additionally, it incorporates features such as obstacle detection and the display of the robot's status, providing a comprehensive overview of the robot's status in the environment.

The technology stack employed for this interface includes JavaScript, utilizing the *roslibjs* library, and PHP, a widely used web programming language for backend development. While PHP serves as the primary backend language, incorporating Smarty, a template engine for separating the presentation layer from data manipulation in web applications, JavaScript plays a crucial role in the frontend.

However, one limitation of this application was the organization and separation of functionalities and responsibilities between the backend and the frontend. The division between these two

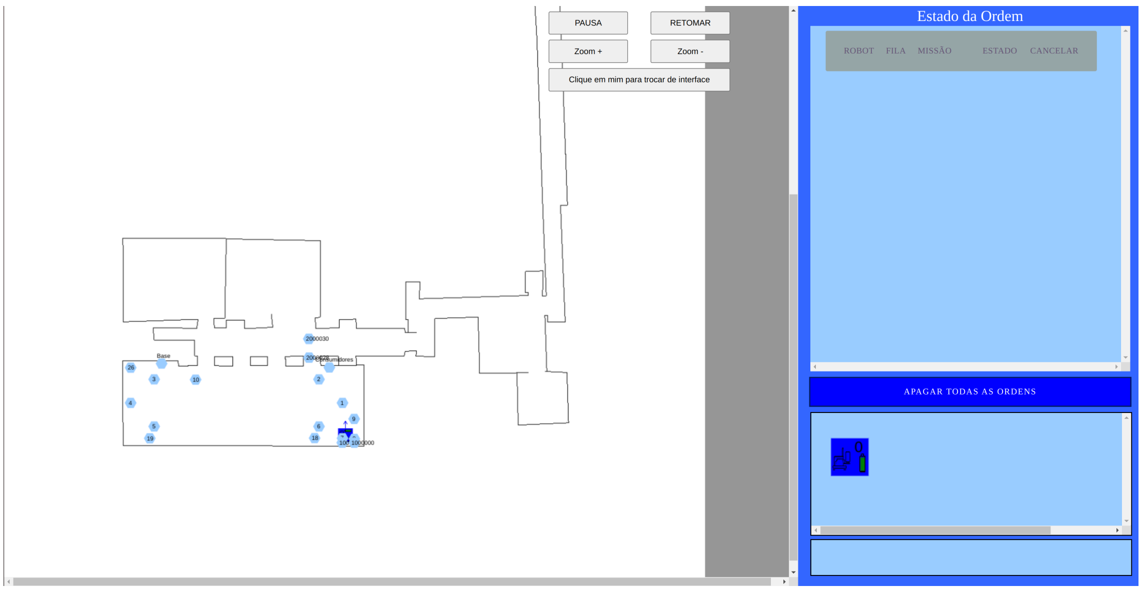


Figure 3.8: Screenshot of the already existing web interface

parts was not clear-cut, as data retrieval was accomplished through the *roslibjs* library, which resulted in a significant portion of the backend responsibility being implemented with JavaScript rather than PHP. In fact, PHP was mainly used for organizing files and separating the presentation layer from the data manipulation files, while JavaScript handled all the processes from the retrieval to the representation of the data in the interface.

Additionally, the inclusion of multiple programming languages introduces complexity and cognitive overload for the developer. The development process becomes more challenging, requiring proficiency in both languages and careful coordination of data exchange between them. For this reason, the maintenance of the application becomes more difficult as well.

Furthermore, a notable drawback of this application is the lack of separation between data retrieval and data representation. The application operates within a single thread functioning as a monolithic application. As a result, the efficiency of the application is compromised, as the main thread remains blocked during data retrieval, manipulation and representation operations.

In conclusion, by making use of some web technologies already used in other interfaces and adding the extra functionalities and efficiency that the back-end libraries provide, the interface that was developed fits perfectly in the current state-of-the-art of robot web applications by allowing the user to do the basic tasks that were possible before with other applications while improving efficiency and bringing new technologies into this work field.

The web interface developed in this dissertation fits perfectly in the current project, as it allows for substantial improvements in usability, whether in terms of accessibility by employing web technologies that bring this characteristic to the project, whether regarding the improvement of many features or the addition of others that were absent in the previous interfaces. Additionally, it effectively addresses the limitations of the previous interfaces and focuses on enhancing the application's architecture and performance to ensure overall effectiveness and maintainability.

# Chapter 4

# System Setup and Data Acquisition

As this dissertation is part of an ongoing project, the development of the application for this study requires the installation of the project's software, which serves as a foundation and support for our application. Furthermore, prior to initiating the application development, an essential phase involved the conceptualization of the application, where its architecture and requirements were carefully defined and outlined. This phase encompasses considerations regarding the chosen technology stack, as well as the overall conceptualization of the application's functionalities and objectives.

Therefore, this chapter begins with the Robot Connection Setup section, which outlines the process of installing all the necessary software to support our application and establish a connection with the robot.

Next, the Technology Stack and Application Architecture section provides an overview of the conceptualization process, where the application architecture was chosen after careful analysis along with the selection of the appropriate technology stack.

The Application Requirements section is also included in this chapter, addressing the specific requirements that the applications must fulfil. These requirements serve as the foundation for determining the necessary data to be retrieved from the robot.

Given the selected requirements, a crucial step was to identify the appropriate ROS topics and services that encompass the relevant information required for the representation. These topics and services serve as the source of data for our application. Finally, the methodology employed to retrieve this data through the REST API is also depicted in the final section of this chapter.

## 4.1 Robot Connection Setup

In order to configure and establish communication with the robot, it was necessary to install the essential robot-side software.

First of all, we installed ROS Melodic, which is the version that was used to develop the navigation stack of all the robots. From then, all the navigation stack, the necessary components and modules related to ROS were placed inside a catkin workspace, which is a directory where ROS packages are built.

After building the catkin workspace and launching its executables, it was already possible to control and monitor the robot through the two interfaces already mentioned above in Chapter 3: the interface made with RViz, represented in Figure 3.7a, and the Iris Interface, shown in Figure 3.7b.

To assure communication with the robot through the web, it was necessary to install the rosbridge version that was compatible with the ROS version installed on the computer and used for the control of the robot. For that, it was necessary to create a catkin workspace, build it and launch its executables, just like for the robot software.

## 4.2 Technology Stack and Application Architecture

The next thing to do, even before the setup of all the necessary software to start building our application, was to choose the stack of technology that would be underneath the whole code base of our interface. After all the research made in Chapter 2, this process was much easier and straightforward to do.

As we saw in Chapter 2, we used the three main web programming languages nowadays: HTML, CSS and JavaScript. The latter is used together with the library *roslibjs* to connect with the robot operating system - ROS.

Moreover, as we had to do the full stack of the application, we had to choose whether to use a frontend or a backend framework/library or even a different programming language on the back office of the interface. The last option was impracticable as we chose to use *roslibjs*, which is a JavaScript framework built on top of the *rosbridge* protocol. After analysing all the benefits and disadvantages that frontend and backend frameworks/libraries could bring to the development of the application, it was clear that having a backend framework in which the application could communicate and get the data from the robot more efficiently and organized was far more important than any potential efficiency gains offered by a frontend application. Furthermore, adding a frontend framework would also bring unnecessary cognitive overload to us, the developers of the application, as we had to learn a whole different framework.

This led to the usage of Node.js, which, as we've mentioned before, is a server-side JavaScript environment in which we can run JavaScript code without the need for a web browser. With this, Express will also be used to allow the application to be split into a distributed system, instead of a monolithic architecture, as we will see below.

In light of Chapter 2, considering all the advantages and disadvantages of all the mentioned architectures, we decided to implement the **client-server architecture** in our project.

It was clear from the beginning that having an application with a monolithic architecture, where we had to fetch data constantly from the robot and draw some representation of that data in the interface at the same time was impracticable. After analysing both the microservices and the client-server architecture, we concluded that, while both allowed having a distributed system, there was no need to have more than an intermediate level between the robot and the drawing of the interface. Moreover, the fact that we could centralize all the data into a single entity while keeping a clear separation of concerns between the server and the clients, together with its ease of integration and scalability were key factors in choosing the client-server architecture.

Therefore, to achieve this architecture, we chose to implement a REST (Representational State Transfer) API. In this implementation, resources, i.e., any information or entity that is meaningful within the context of a web application, should be identified by unique URLs, in a way that they can be manipulated and accessed through HTTP methods like GET, PUT, POST, and DELETE [40]. Furthermore, each request is treated as stateless, which implies that the server does not store any client-specific information between requests, thus, relies only on the information coming from the client on these HTTP methods [40]. In summary, having the HTTP methods as the only interface for accessing and manipulating the data inside the REST API, together with its stateless model, gives the developer ease of scalability, since many clients can be created around the REST API without affecting directly its performance while allowing for simplicity in the retrieval and handling of the data.

Figure 4.1, displays a very simplified representation of the projected architecture for the application. While observing this image, it is evident that there is a clear separation between the backend and the frontend of the application.

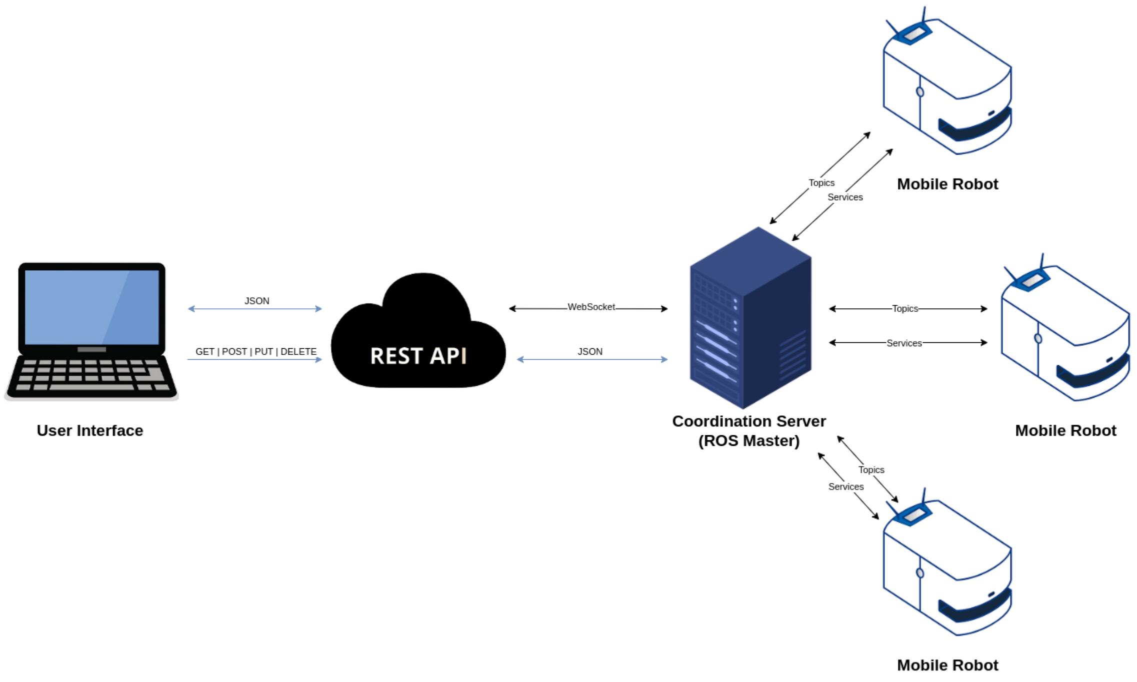


Figure 4.1: Application's projected architecture

On one side, the backend, represented by the REST API, handles all the data that robots send or receive. Therefore, using the *roslibjs* library, this REST API establishes a WebSocket connection with the coordination server - the server that hosts the ROS Master and connects with the robots -, thus creating a background task to handle the incoming messages. Then, the REST API can publish or subscribe to a topic or call a service, while sending and receiving data encapsulated in JSON format.

On the other hand, the front-end side of the application, which interacts directly with the user, does not have direct access to the data coming from the robots. This occurs because the REST API works as an intermediary between the robots (and their common coordination server) and what the user interacts with. It encapsulates the data received or sent by the robots, allowing access to this data only through HTTP requests such as GET, POST PUT or DELETE.

Because it was necessary to host the user interface, installing a local server provider became essential for the development of the application. Therefore, we chose Apache HTTP Server which offers the advantage of being cross-platform, working whether in Linux, Windows or even macOS. This was the decisive factor when choosing this local server provider as it gave us the possibility to host the REST API and the user interface on different devices with different operating systems.

To allow the user interface to send HTTP requests to the REST API, there was no need for the installation of any third-party software since JavaScript already provides the XMLHttpRequest API, a Web API that allows the browser to make HTTP requests to a JavaScript runtime environment. However, to handle the communication with the robots, it became necessary to have a JavaScript runtime environment running in the background. Thus, the installation of Node.js became indispensable. Node.js was utilized to host the REST API and, by installing the necessary packages - Express to allow the API to receive and send data to the user interface and roslibjs to communicate with the ROS Master -, all the necessary software for the development of the web application was now installed.

## 4.3 Application Requirements

In this section, we will outline the key requirements of our application, which will serve as the foundation for every step that follows in its development, from the selection of relevant data from each robot to the representation of this data in the user interface.

Through careful analysis of the project, we have identified the core requirements of the user interface, incorporating some features of the already developed interfaces with new functionalities that enhance and complement the overall project. Presented below are a set of requirements that the application should satisfy:

- Display the map of the robot's environment with its vertices, i.e., the designated stopping points for each robot, barring any obstacles, while excluding the edges of the graph (the only paths that the robot could take to walk between the vertices).

- Represent each robot's location in the map in a 2D scenario from an overhead perspective.

- Allow for interactive map controls, such as the ability to zoom in and out for a closer or wider view of the map and the possibility of scrolling in all directions to navigate and explore the entire map.

- Toggle between zooming in on the robot or the centre of the presented map area.
- Enable the user to give well-defined goals and actions as orders to the robots.
- Display the current queue of each robot's actions.
- Provide information about each order's goal, its position in the queue and its working status (to tell if it's in process, in the queue or even if it was cancelled).
- Allow the cancellation of an individual or all queued orders.
- Indicate the online status and battery percentage of the robots.
- Detect obstacle collision.
- Allow the toggling of maps, such as when a robot changes floors.
- Provide a simplified interface, as an alternative to the interface with the map and the robots, that only allows sending orders to the robots.

While these specifications serve as the main guidelines for developing this web application, it is crucial to emphasize that these requirements will require a process of adaptation as the application scales, particularly regarding the representation of multiple robots in the interface.

## 4.4 Selection of ROS Topics and Services for the Robots' Data Acquisition

To start developing the application, it was detrimental to extract only relevant information from the robots to be able to represent the data correctly and satisfy the requirements above. For that purpose, our main goal was to select suitable topics and services from the navigation stack. These topics and services would, then, enable us to receive meaningful data to be represented in our interface and facilitate the transmission of essential commands to the robots.

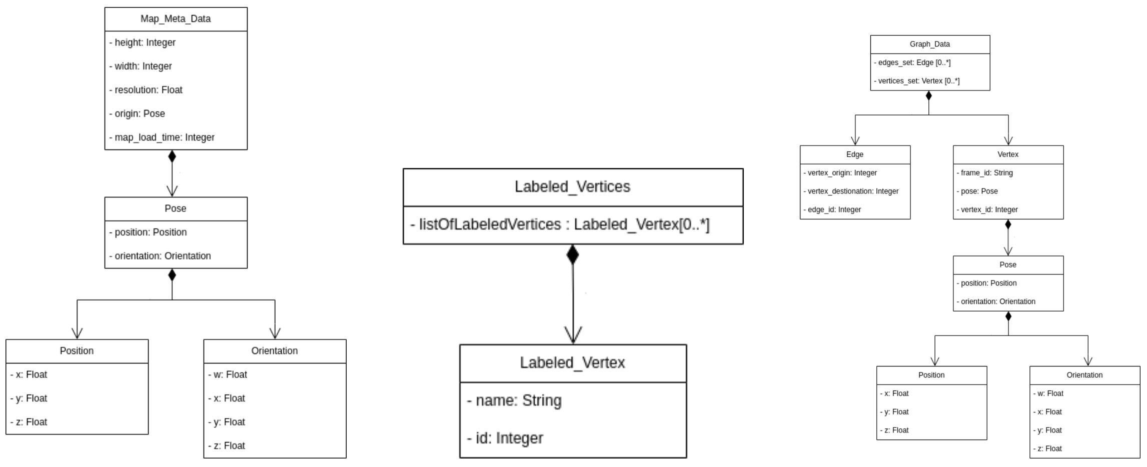
Firstly, we started to select nodes that would provide us with meaningful data to represent. This involved not only the position of the robots on the map but also, the status of the actions' queue and other relevant information. Below, you will find a comprehensive list of the selected topics that facilitated the representation of all the data showcased in the interface:

1. Map Data
    - Map Metadata
    - Labeled Vertices
    - Graph Data
2. Robot Data

- Robot Position
- Robot Cargo
- Robot Battery
- Obstacle Detection
- Robot's Action Queue

These topics are divided into two main groups: the data related to the map and the data coming constantly from the robots. The data derived from the former is obtained during the configuration phase of the robots and is only published once, while the data coming from the latter is published periodically while the robots are online.

Figure 4.2 depicts the structure of data coming from the three topics chosen to retrieve the map data. In Figure 4.2a, we can observe the **Map_Meta_Data** topic, which has a message type that includes crucial data about the map image, including its resolution, height and width. Additionally, it provides information about the image origin and the map loading time. Moving on to Figure 4.2b, the **Labeled_Vertices** topic message type consists of a simple list containing details about the vertices that have labels, such as the label name and vertex id. Finally, as depicted in Figure 4.2c, the **Graph_Data** topic message type consists of two lists: one containing every edge and another containing every vertex that constitutes the graph. Each element of these lists provides additional details about the corresponding vertex or edge.



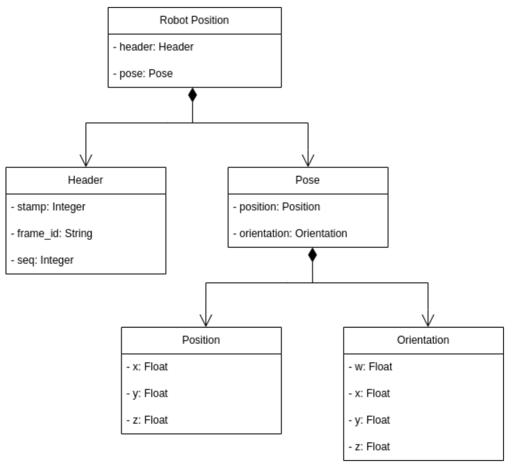
(a) Message type of the Map_Meta_Data Topic

(b) Message type of the Labeled_Vertices Topic

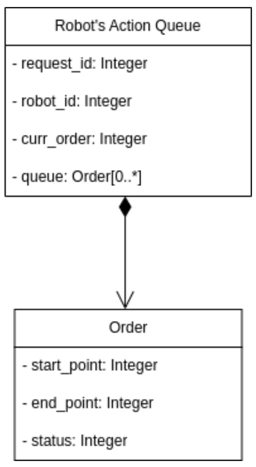
(c) Message type of the Graph_Data Topic

Figure 4.2: List of message types from the Map Data Topics

Moving on to the topics related to the robots' data, the robots' battery, the robots' cargo, and the obstacle detection topics' messages are represented by simple data types, such as a floating point number, a boolean, and a floating point number, respectively. On the other hand, the message types of the other topics are more complex. Figure 4.3 illustrates several diagrams representing how the data is organized in each of these topics related to the robots' data.



(a) Message type of the RobotPosition Topic



(b) Message type of the Robot's Action Queue Topic

Figure 4.3: List of message types from the Robot Data Topics

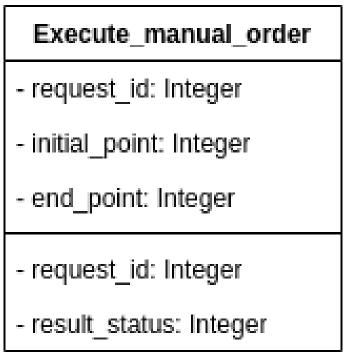
As shown in Figure 4.3a, the message type of the **Robot_Position** topic is divided into two parts: the header and the pose. The header includes overall information such as the current timestamp and the id of the frame, which is a string representing the map and the floor where the robot is located. On the other hand, the pose object comprises the position, given by the (x,y,z) system of coordinates and the orientation given by the quaternion (w,x,y,z).

Proceeding to Figure 4.3b, it depicts the info related to each order sent to each robot. The first three fields are common to all orders from the same robot, which represents the current id of the request, an incremental variable that stores the id of the last order sent to the robot and the id of the robot. The final field represents the id of the order that the robot is currently performing. Additionally, the Order object contains specific information about each order stored in a list of Order objects, including the starting point, ending point and the status of the order.

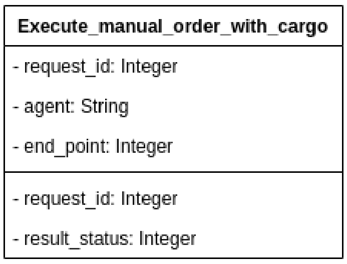
Lastly, we needed to select the appropriate services to send orders to the robots through the server that is running the navigation stack. For that purpose, we chose the following:

- Execute Manual Order
- Execute Manual Order with Cargo
- Cancel Order
- Clear All Orders
- Resume Robot Activity
- Pause Robot Activity

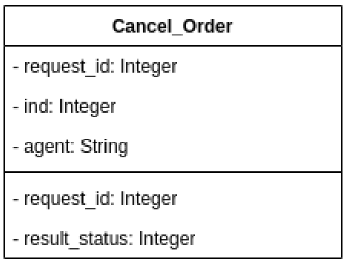
The first service, depicted in Figure 4.4a as the name already implies, sends an order to a robot, specifying the associated request id associated, the initial point and the endpoint, i.e., specifies the



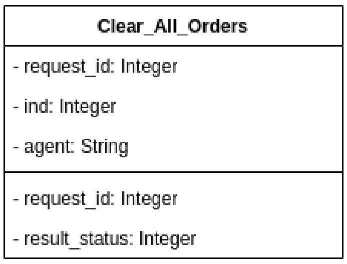
(a) Request and Response Objects of the Execute_Manual_Order Service



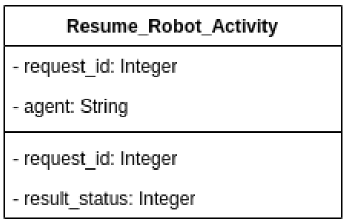
(b) Request and Response Objects of the Execute_Manual_Order_with_cargo Service



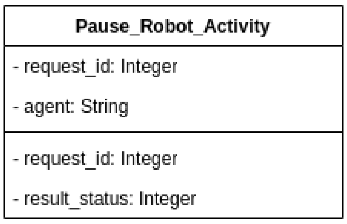
(c) Request and Response Objects of the Cancel_Order Service



(d) Request and Response Objects of the Clear_All_Orders Service



(e) Request and Response Objects of the Resume_Robot_Activity Service



(f) Request and Response Objects of the Pause_Robot_Activity Service

Figure 4.4: List of Request and Response Objects from the Services

initial vertex in which the robot has to pick up the cargo and the final point, which is the vertex where the robot has to drop off the cargo. In response, it includes the request id from the request object and the order status, indicating whether it executed successfully or encountered any errors. Figure 4.4b illustrates the request and response structure for the service to execute a manual order when the robot has cargo. This service is structurally similar to the previously mentioned one, with the only difference being the request object. Instead of an initial point, which can't be assigned as the robot must deliver the cargo to a final vertex, it includes an agent variable, representing the name of the robot performing the task.

Proceeding to Figures 4.4c and 4.4d, which depict the services for cancelling a single order and clearing all orders, respectively. Both services share a similar structure, however, when cancelling a specific order, the *ind* variable must indicate the id of the order to be cancelled. On the other hand, when clearing all orders, the *ind* should be set to 0.

Lastly, the last two services, illustrated in Figures 4.4e and 4.4f, are used to play and pause the robot's activity, allowing for the temporary cessation of any ongoing actions by the robot, with the possibility to resume it later.

It is important to consider that this set of topics and services is associated with a single robot, and some of them are even specific to a particular floor. As the application scales up and allows for more floors and robots, it becomes necessary to retrieve the data from these topics for each respective robot and floor. This introduces a more complex data acquisition method in the REST API, which will be explained in detail below.

Overall, this set of topics and services provides the necessary data to be represented in the interface while allowing communication with the robots, by sending messages with the necessary information to complete specific tasks.

## 4.5 REST API

After choosing the appropriate topics and services and installing the necessary software, the initial step in the application's development phase was the creation of the REST API.

The requirements for the REST API were relatively straightforward. Its main purpose was to retrieve data from the robots, organize it in a simple manner and establish access points to the user interface through HTTP requests.

In this section, we will outline all the essential steps involved in the implementation of the REST API. We will provide insights into the creation of the server that is hosting the application, the establishment of the connection with the ROS Master and an overview of how the data is retrieved and organized.

### 4.5.1 Configuration Variables

To enhance the scalability of our application, we have implemented configuration variables at the start of the application. These variables can be modified by the user before initializing the application.

The inclusion of these configuration variables contributes to the versatility of our application. By enabling users to specify the names of the robots and associate them with corresponding floors mapped by the robots, the REST API can dynamically manipulate this data to retrieve and send information through the topics and services related to each robot and each floor. This process is accomplished by iterating through the arrays of robots and floors provided by the user to dynamically select the relevant topics and services for data transmission.

### 4.5.2 Initialization of the Server

Moving on to the initialization of the server, we utilized the Express library to create an Express instance, which offers a variety of methods from modifying the application settings to handling HTTP requests. With this instance, we were able to start a server that listens for connections on an arbitrarily chosen port. We selected port 3000, but any available port could have been chosen at our discretion, provided that it was not already in use by another application. At this point, we had the server up and running and we could now try to establish a connection to the ROS Master.

### 4.5.3 Connection to the ROS Master

Since one of our main goals in the development of this application was the user's accessibility, we decided to implement both the REST API and the user interface in a way that the user doesn't have to care about whether to start the ROS Master, the REST API server or the user interface first.

Figure 4.5 illustrates the process model for this implementation. Utilizing *roslibjs*, we instantiated a Ros node object named *ros* to establish communication with the *rosbridge* server running in the background. Then, we initially try to establish a connection with the ROS Master, by specifying its IP address and its designated port. By adding an event listener to the connection event to the *ros* object, we could now assure if the connection was successful, closed or if any error occurred during the process.

By handling these events, we could now determine whether the connection was successfully established. In cases where the connection was not established or it was closed, the REST API would try to reconnect to the ROS Master until the connection was successful. These repeated attempts to connect to the ROS Master were made using the *setInterval* method provided by JavaScript - this method allows for the creation of a timer that calls a function at specified intervals.

Once the connection was established, we utilized the *clearInterval* method, also provided by Javascript, to cancel the timer responsible for the connection attempts. This allowed the REST API to proceed with the creation of topics and services and the retrieval of data as further described below. Note that, once the REST API was connected, it also included the two event handlers to detect connection failures (*'close'* and *'error'*). This allowed for the automatic restart of the connection process with the ROS Master as soon as a failure occurred.

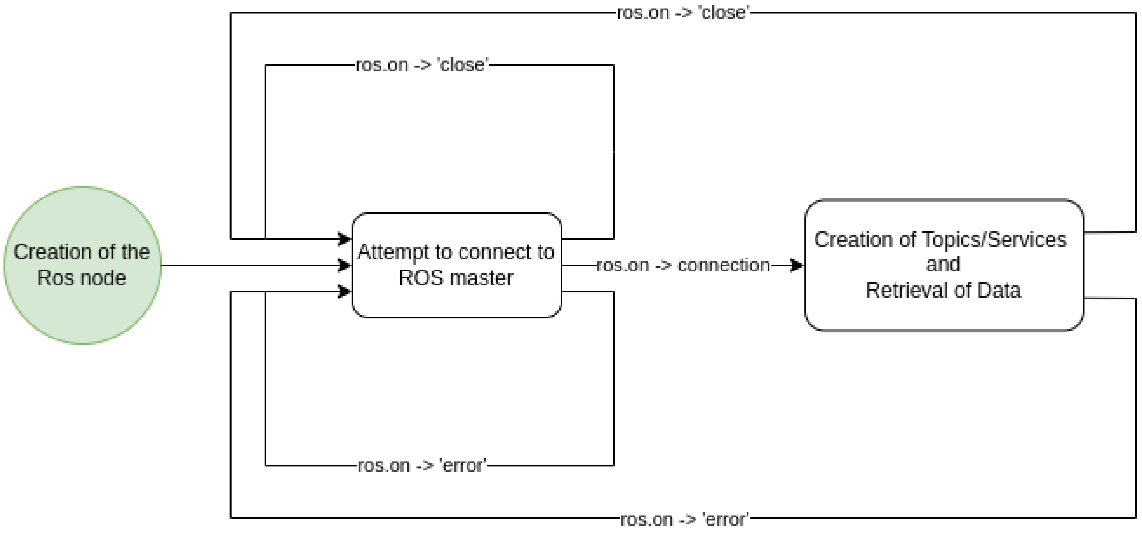


Figure 4.5: Process Model of the Connection to the ROS Master

By implementing this approach, we improve the usability of our application, as the user doesn't have to bother starting the ROS Master, the user interface and the REST API in a specific order, as the REST API will be attempting to connect to the ROS Master periodically. Additionally, with the automated connection restart mechanism working in the background, trying to establish a connection to the ROS Master, the user is spared the burden of managing the initialization process.

### 4.5.4 Retrieval and Organization of Data coming from the ROS Master

After establishing a successful connection, we had to retrieve data from the selected topics that were running in the ROS Master. To accomplish this, we declared a variable associated with each topic - a Topic object -, as Listing 4.1 depicts. This variable had a specific format: it included the variable associated with the ROS Master, in this case, *ros*, followed by the name of the topic running on the ROS Master and finally, the message type for that topic that was predefined in a file located in the catkin workspace where ROS was launched. It is important to acknowledge that the name of the topic associated with each topic is not hard-coded. Instead, it is obtained using the configuration variables specified by the user before initializing containing the robots and floors names.

Listing 4.1: Example of the declaration of a ROS Topic Object

```
mapMeta_topics[i] = new ROSLIB.Topic({
ros : ros,
name : '/' +robots_names[j]+ '/' +map_floors[i]+ '/map_metadata',
messageType : 'nav_msgs/MapMetaData'
});
```

The Topic object created can be utilized for publishing, subscribing, or both. In our case, since we're only interested in retrieving data from the robots through the selected topics, we will focus on the subscription aspect. Therefore, to subscribe to a specific topic, we need to call the *subscribe* function, a method associated with the Topic object which requires a callback function as its argument. Whenever a message is published on the designated topic in ROS, the REST API will receive that data through *rosbridge*, and the callback function associated with the subscribe function will be invoked with the received data.

Hence, in order to retrieve and store the data in the REST API, we made the decision to create global variables for each topic's data. This approach allows us to update the corresponding global variables whenever the callback function (that is inside the *subscribe* method) is invoked. Consequently, the global variables are always up-to-date with the most recent values published by the robots.

When it comes to services, the approach for declaring variables associated with services is quite similar to the one used for topics. As shown in Listing 4.2, we also declare a Service object for each service, following the same structure as the Topic object.

Listing 4.2: Example of the declaration of a ROS Service Object

```
PlayClients[j] = new ROSLIB.Service({
ros : ros,
name : '/' +robots_names[j]+ '/Application_layour/Execute_order',
serviceType : 'OH_ressources/Execute_manual_order'
});
```

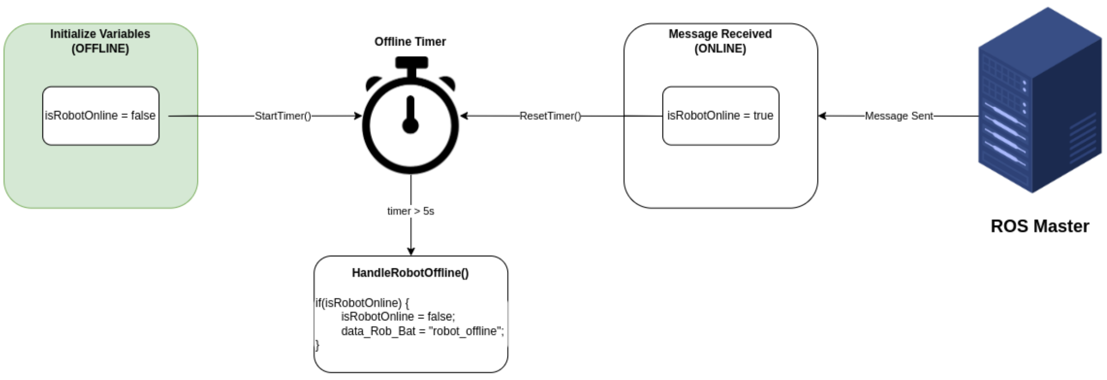
For communicating with the robots using services, we utilize the *callService* function, which is a method associated with the Service object. This function requires two arguments: a ServiceRequest object and a callback function. The ServiceRequest object should have the structure specified for the corresponding request as seen in Figure 4.4. The callback function is invoked when the ROS service responds, providing the developer with the result status information.

### 4.5.5 Offline Detection Process

An additional feature that was missing in the previously developed web interface was the representation of the robots' online status. In the already existing web interface, the online status of the robots was not accurately reflected, as it couldn't detect if a robot disconnected from the ROS Master or was turned off.

In order to overcome this limitation, we implemented a continuous monitoring process to check if the robots are actively sending messages within a 5-second interval. If no messages are received from a robot during this time frame, it is determined that the robot is offline.

Figure 4.6 illustrates the process model for implementing this feature for a single robot. In our application, this process is repeated for each robot, by going through the array of the robots specified in the configuration variables.



```
if(isRobotOnline) {
    isRobotOnline = false;
    data_Rob_Bat = "robot_offline";
}
```

Figure 4.6: Process Model of the Offline Status Detection

The process begins with the initialization of a boolean variable that holds the online status of the robot. Initially, the variable is set to false, indicating that the robot is considered offline. Alongside this, a timer is also initialized.

From this point on, the detection of the robot's online status is managed by this timer. When a message is received from the corresponding robot via the ROS Master, the timer is reset and the robot is considered online. This ensures that as long as the robot continues to send messages within the designated interval, it remains marked as online.

However, if the robot stops sending messages for a duration exceeding 5 seconds, the boolean variable representing its online status is set to false, indicating that the robot is now considered offline. This detection process is done through the *SetTimeout* method provided by Javascript, which allows the invocation of a designated function once the timer reaches the specified interval.

By implementing this functionality, the web interface can accurately determine and reflect the online status of each robot. It ensures that the users have up-to-date information regarding the availability of the robots which is a crucial factor for the perception of the overall status of the robots.

### 4.5.6 Endpoints of the REST API

All the data and methods related to the topics and services of the ROS Master need to be represented on the user interface. To accomplish this, the REST API, not only needs to retrieve and send the data to the robot but also allow the user interface to interact with it.

In our application, we achieve this by utilizing the routing feature of the Express library. This property enables the creation of endpoints (or URLs), to handle client requests. The routing process is carried out using methods associated with the Express instance, which was created earlier to start the server. These methods correspond to HTTP requests, such as the GET(with its corresponding method *get*) and POST(with its corresponding method *post*). They accept the URL of the request as an argument and a callback function that is invoked whenever the REST API receives the corresponding HTTP request.

Additionally, to complement the use of the Express library, we included also the *body-parser* library. This allows our REST API to transmit data to the user interface in JSON format, i.e. in the form of JavaScript objects. This simplifies the process of gathering and manipulating data from the point of view of the user interface.

Throughout the next chapter, we will provide more detailed explanations, supported by relevant figures, regarding the specific endpoints responsible for data transmission between the REST API and the user interface. This will offer a clearer understanding of the data flow and communication processes between these components.

The processes outlined in this chapter, from the software setup to the data retrieval and the creation of endpoints to allow communication between the REST API and the user interface, comprise the application's backend. These processes serve as the foundation for the development of the front-end side of the application, which will be summarized in Chapter 5.

# Chapter 5

# User Interface

For the user interface, we divided it into two primary components: communication and data retrieval from the REST API, and the manipulation and representation of this data.

To implement this structure, we have utilized the Web Workers API. As mentioned before, this API enables the creation of separate threads or workers, which run concurrently with the main thread. By leveraging the workers, the application becomes more efficient and responsive, as it can perform multiple operations simultaneously. Additionally, tasks can run independently and asynchronously without blocking the main thread.

Although, as mentioned earlier, working with threads can introduce some problems or complexities to an application, these concerns do not apply to our specific application. This is because our threads/workers do not share any resources, eliminating the possibility of race conditions and negating the need for thread synchronization.

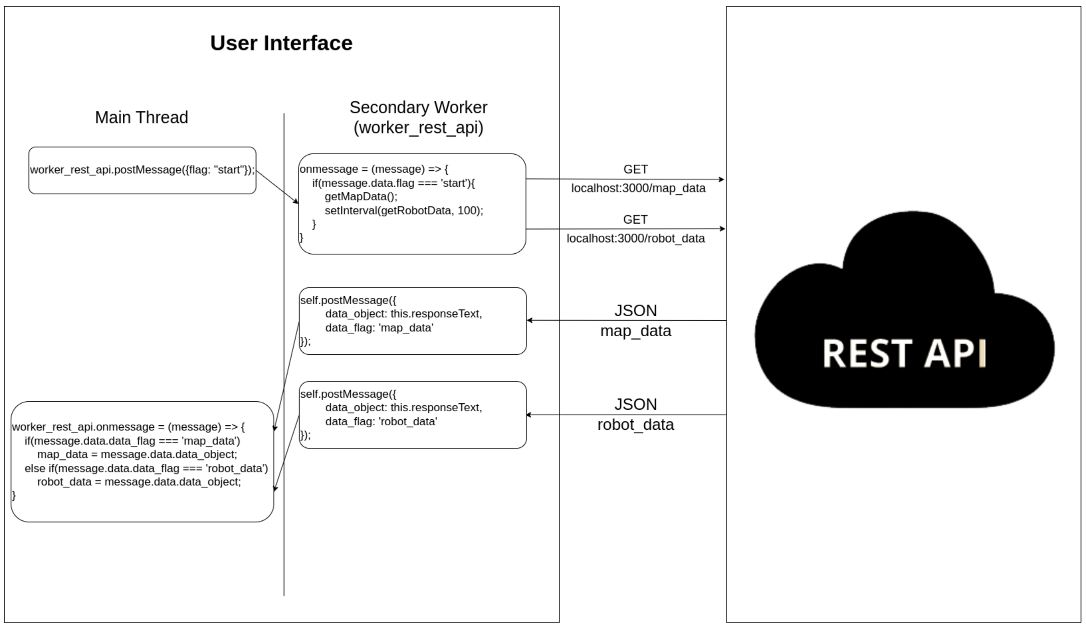
Therefore, by acknowledging these advantages, with the help of this API, we could divide the code base into two different workers/threads, the main one, for the communication with the REST API and the secondary worker, for the representation of the data coming from the secondary worker.

## 5.1 Data Retrieval

To proceed to retrieve data from the REST API, we created two global variables: **map_data** and **robot_data**. These variables store map-related data, including information about all the floors specified in the configuration variables of the REST API, and data specific to each robot, which is continuously updated by each robot, respectively.

The data retrieval process is accomplished by accessing the endpoints defined in the REST API. This communication process between the workers and the REST API is illustrated in Figure 5.1 and will be thoroughly explained in the following lines.

The communication with the REST API is achieved through the secondary worker, responsible for retrieving data from the REST API via HTTP requests and forwarding it to the main worker.


User Interface
Main Thread
Secondary Worker (worker_rest_api)
GET localhost:3000/map_data
GET localhost:3000/robot_data
JSON map_data
JSON robot_data
REST API

```
worker_rest_api.postMessage({flag: "start"});
```

```
onmessage = (message) => {
   if(message.data.flag === 'start'){
      getMapData();
      setInterval(getRobotData, 100);
   }
}
```

```
self.postMessage({
      data_object: this.responseText,
      data_flag: 'map_data'
});
```

```
self.postMessage({
      data_object: this.responseText,
      data_flag: 'robot_data'
});
```

```
worker_rest_api.onmessage = (message) => {
   if(message.data.data_flag === 'map_data')
      map_data = message.data.data_object;
   else if(message.data.data_flag === 'robot_data')
      robot_data = message.data.data_object;
}
```

Figure 5.1: Process of the retrieval of the data from the REST API

These workers communicate with each other using the *postMessage* method provided by the Web Workers API, along with the *onmessage* event handler.

This process begins with the main thread sending a flag to the secondary worker using the *postMessage* method. Upon receiving the message, the secondary worker, which has an *onmessage* event handler waiting for messages from the main worker, processes the message and starts retrieving data from the REST API.

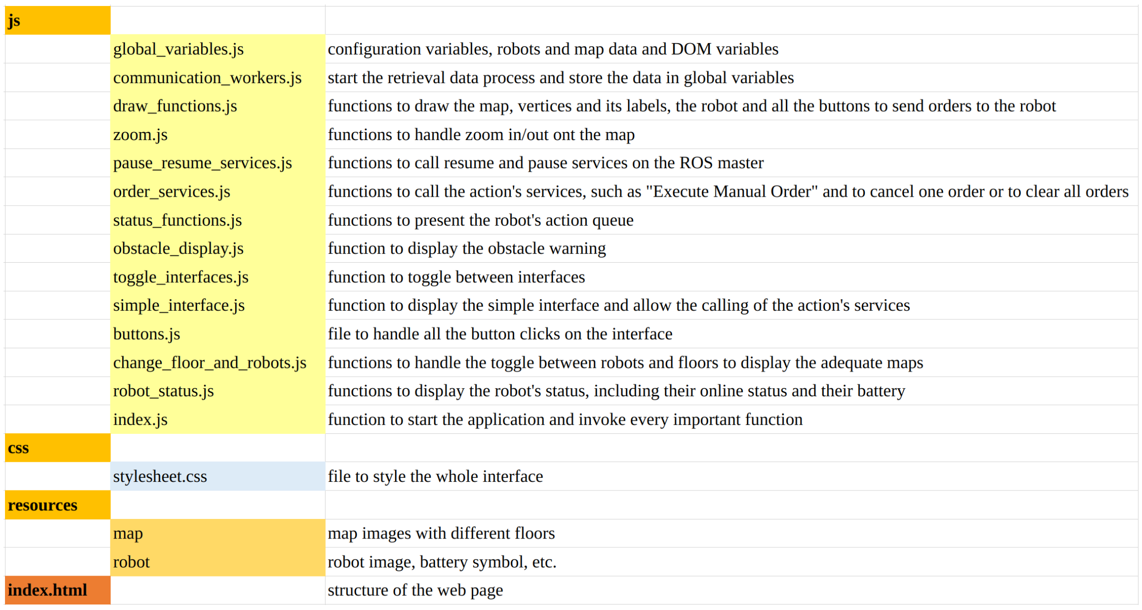
The data retrieval process starts with obtaining the map-related data which remains unchanged throughout the whole lifecycle of the application. Following that, using the *setInterval* method, the robots' data composed of their position, their battery level, their cargo, their obstacle detection and their actions queue is retrieved from the REST API every 100 milliseconds. The communication with the REST API is accomplished, as mentioned before, using HTTP requests but, unlike on the REST API, on the user interface side, making these requests does not require the installation of any third-party packages since JavaScript already includes the XMLHttpRequest object for that purpose.

The map and robots data are accessed and retrieved from the REST API through their respective endpoints (*localhost:3000/map_data* and *localhost:3000/robot_data*) using the GET request. The response data is stored in the *responseText* variable in JSON format. With this data at hand, the secondary worker sends it to the main thread using *postMessage* with a specific flag. This flag allows the main worker to differentiate between different types of data received from the secondary worker with a conditional statement that handles the flag. Afterwards, the main worker stores the received data from the REST API in global variables, which can be accessed by all JavaScript files in the user interface directory.

## 5.2 File Organization and Hierarchy

The organization of code into separate files is generally considered a good practice. In our case, where the application can be used in different projects, this aspect is detrimental. Not only does it improve the readability of the entire code base but it also makes it more maintainable by breaking the application into smaller files with clear names and purposes.

Figure 5.2 provides a visual representation of the organization and hierarchy of the files within our user interface codebase, providing information about the functionalities performed by each code file.



| | | |
|---|---|---|
| **js** | | |
| | global_variables.js | configuration variables, robots and map data and DOM variables |
| | communication_workers.js | start the retrieval data process and store the data in global variables |
| | draw_functions.js | functions to draw the map, vertices and its labels, the robot and all the buttons to send orders to the robot |
| | zoom.js | functions to handle zoom in/out ont the map |
| | pause_resume_services.js | functions to call resume and pause services on the ROS master |
| | order_services.js | functions to call the action's services, such as "Execute Manual Order" and to cancel one order or to clear all orders |
| | status_functions.js | functions to present the robot's action queue |
| | obstacle_display.js | function to display the obstacle warning |
| | toggle_interfaces.js | function to toggle between interfaces |
| | simple_interface.js | function to display the simple interface and allow the calling of the action's services |
| | buttons.js | file to handle all the button clicks on the interface |
| | change_floor_and_robots.js | functions to handle the toggle between robots and floors to display the adequate maps |
| | robot_status.js | functions to display the robot's status, including their online status and their battery |
| | index.js | function to start the application and invoke every important function |
| **css** | | |
| | stylesheet.css | file to style the whole interface |
| **resources** | | |
| | map | map images with different floors |
| | robot | robot image, battery symbol, etc. |
| **index.html** | | structure of the web page |

Figure 5.2: Organization of the User Interface Directory

The primary file for the user interface is *index.html*, an HTML file that defines the structure of the user interface. This file is accompanied by the *stylsheet.css* file, which contains style directives for the entire application. Additionally, the HTML file includes JavaScript files (located in the *js* folder) that provide all the user interface features.

The JavaScript files are organized based on the functionalities they offer to the user. Furthermore, they follow a hierarchical organization, where global variables and methods associated with each file are made available to subsequent files. Because of that, we decided to include the *global_variables.js* first. This file encompasses all global variables and DOM (Document Object Model) objects, which are used throughout the project and managed by the subsequent files. Additionally, it also includes some configuration variables, such as the robots' names, the robots' sizes, the name of the map files, the name of the robots' images, etc.

Additionally, as depicted in Figure 5.2, *index.js* serves as the central file with access to all the variables and functions from the preceding files. This file initializes the interface by invoking all the necessary functions from the files above it. These functions include those related to interface drawing, action queue display, data retrieval and more. Separating these functionalities into distinct files, as shown in Figure 5.2, enhances modularity and simplifies maintenance.

## 5.3 Representation of the Data

This section will illustrate the various processes involved in data representation, starting from the establishment of the user interface structure and encompassing all the steps required to fulfil the previously discussed requirements in Chapter 4.

### 5.3.1 Structure of the User Interface

Before delving into the manipulation of data received from the ROS Master, it is crucial to establish a well-defined structure for the user interface. This step is essential as it enables the allocation of appropriate space for each data representation component and ensures a readable and organized presentation of the data. By structuring the user interface in a thoughtful manner, we can optimize the user experience and facilitate effective data interpretation.

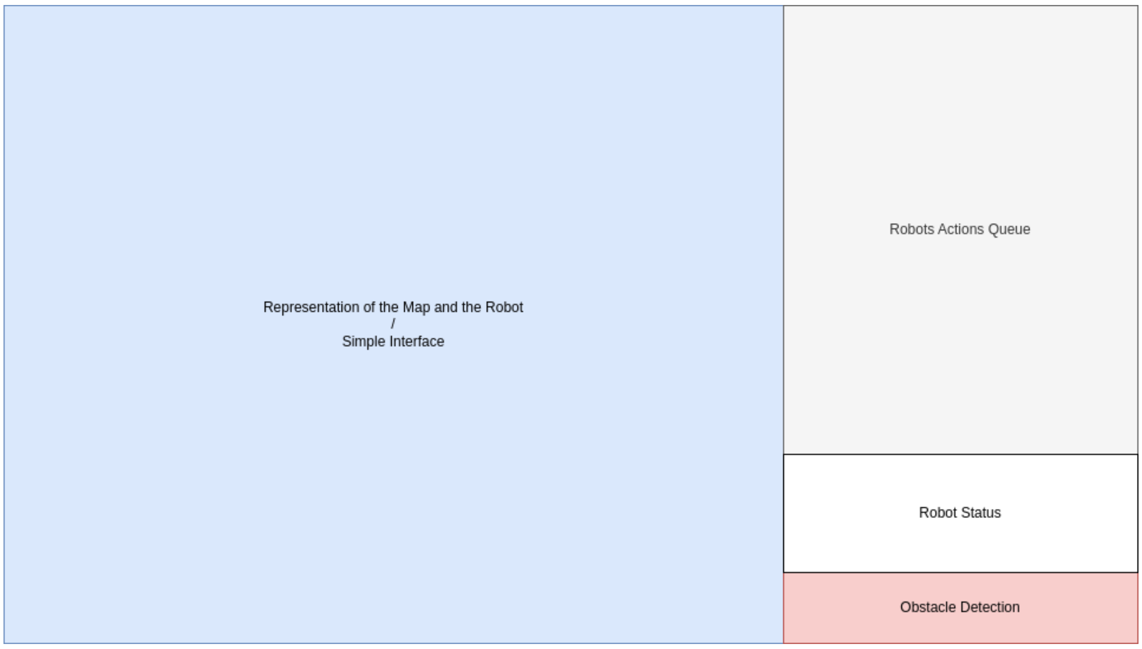


Figure 5.3: User Interface Structure

As shown in Figure 5.3, a significant portion of the web application space is dedicated to the representation of the map and the robot. This representation, using the HTML Canvas API already mentioned in Chapter 2, will have a game-like visualization aspect, where the robots move within the map. Within this designated area, users will have the option to toggle between the previously described interface and a simplified interface. The simplified interface will feature a set of buttons that allow users to issue orders to the robot.

The remaining space within the web application is reserved for secondary data, including the action queue for the robots, the robots' status and the obstacle detection area.

The processes that transform the original data provided by the robot into the representation displayed by the user interface are a crucial component of Chapter 5. Therefore, in the following subsections, we will illustrate all the data manipulation processes involved in creating the final representation of the data on the user interface.

### 5.3.2 Initialization of the Interface

The *index.html* file, as previously mentioned, establishes the fundamental structure for the user interface. Furthermore, it includes all the JavaScript files responsible for representing the data. With the exception of the index.js and *global_variables.js* files, the remaining solely consist of functions and event handlers. Consequently, they need to be invoked by the *index.js* file, which serves as the main file for initializing the interface.

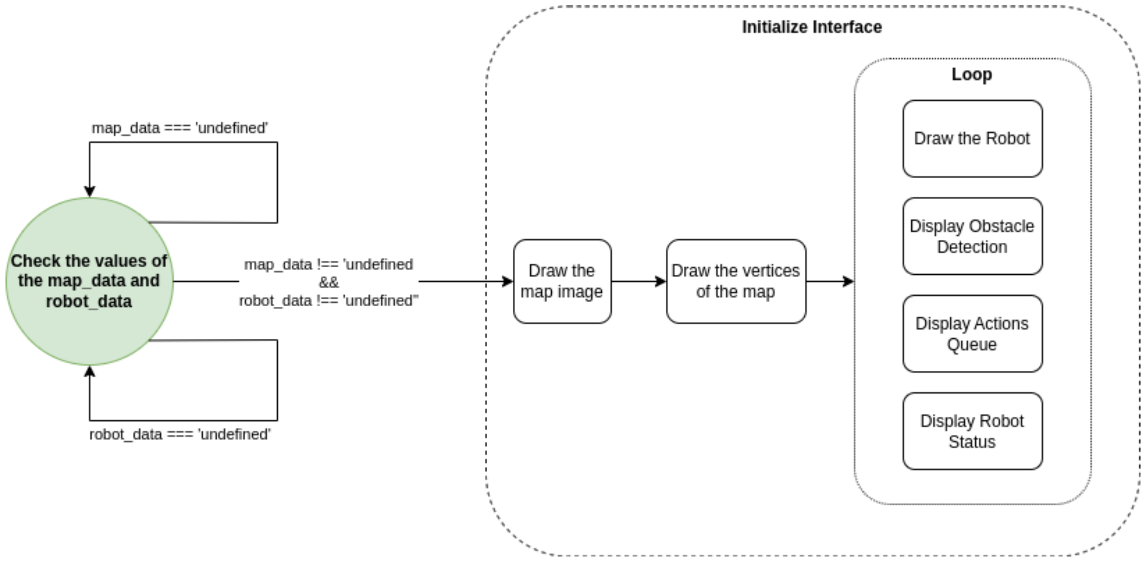

Initialize Interface
Loop
map_data === 'undefined'
Check the values of the map_data and robot_data
map_data !== 'undefined && robot_data !== 'undefined''
robot_data === 'undefined'
Draw the map image
Draw the vertices of the map
Draw the Robot
Display Obstacle Detection
Display Actions Queue
Display Robot Status

Figure 5.4: Process model of the interface initialization

Figure 5.4 illustrates the process of initializing the interface. It begins with an iterative process that continuously checks the values of the *map_data* and *robot_data* variables. This iterative process is done with the help of the *setInterval* method, provided by JavaScript. The iteration continues until both the *robot_data* and *map_data* variables contain data retrieved by the secondary worker.

Once the required data is available, the interface proceeds with its initialization process through a method that encompasses all the necessary steps to represent the data received from the secondary worker. The first step involves drawing the map image, which is assigned based on the associated configuration variable defined in the *global_variables.js* file. Additionally, to provide a complete representation of the map, the vertices are drawn using the data obtained from the secondary worker that is specifically related to the map.

Following the map representation, the data that is directly related to the robot is represented. This representation also involves an iterative process, once again utilizing the *setInterval* method.

This phase of representation of the data is divided into four distinct parts: drawing the robot, displaying obstacle detection, displaying the robots' actions queue and displaying the robots' status. Each of these parts is defined by separate functions that are repeatedly executed at specified intervals. The interval for drawing the robot is set to 100 milliseconds to ensure a smoother display of its movement, while the other three parts have an interval of 500 milliseconds. This frequency is chosen to prevent overloading of the main thread, as a higher frequency is unnecessary for these representations.

In the subsequent subsections, we will delve into these steps in greater detail to provide a comprehensive understanding of the underlying reasoning behind each process.

### 5.3.3 Conversion of Coordinate Systems

One of the challenges we encountered in handling data from the robots was dealing with its coordinate systems.

This issue arose because the robot has two coordinate systems: the map coordinate system and the trajectory coordinate system. The origin of the former is established by the furthest point in the bottom left corner of the map image. On the other hand, the trajectory coordinate system's origin is determined by the first vertex created during the mapping mode. Since we need to display the map in a different coordinate system, specifically the Canvas API coordinate system, we must convert the coordinates from the trajectory coordinate system to the map coordinate system and finally, from the latter to the Canvas API coordinate system.

The relationship between the two coordinate systems associated with the robot is defined by the Map_Meta_Data topic, which provides the origin of the trajectory coordinate system in relation to the map coordinate system. With the coordinates of the vertices and the robots (given by the Graph_Data and the Robot_Position topics, respectively) in the trajectory coordinates system, and the origin of the latter given in relation to the map coordinate system, we can transform both the vertices and the robots coordinates into the map coordinates system by only adding the value of each coordinate of the origin given by the Map_Meta_Data topic.

Subsequently, converting them into the Canvas coordinates system is relatively straightforward, as it only requires the addition of the map image's height to each y and the inversion of the coordinate system in relation to the x-axis, i.e., inverting the y-axis. This adjustment is necessary because, as mentioned before, the origin of the map coordinate system is always at the bottom left, while the origin of the Canvas API coordinate system is always at the top left. An example of this transformation process can be seen in Figure 5.5, with blue circles representing the coordinates of vertices for example and the red circles representing the origin of each coordinate system. Note that all these coordinate system transformations were done in terms of pixels by dividing the x and y coordinates by the resolution, a piece of information also given in the Map_Meta_Data topic.

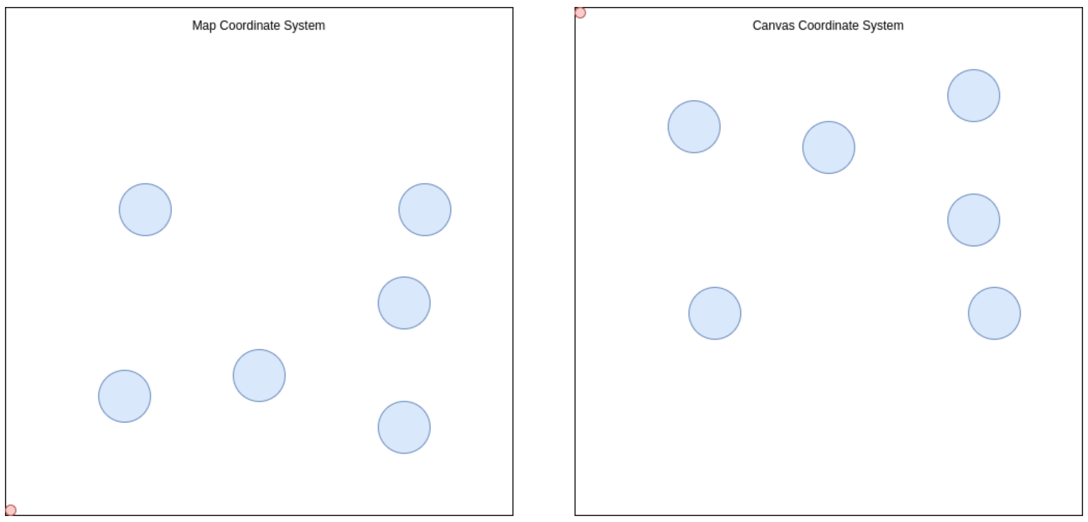


Figure 5.5: Transformation from the Map coordination system to the Canvas coordinate system

### 5.3.4 Drawing the Map

The first crucial element to represent in the interface is the map. This entails the insertion of the map image and the drawing of the vertices associated with it.

To achieve this, we decided to utilize an HTML Canvas object, which allows us to control the position of items on specific coordinates. This choice was driven by the need to accurately place the vertices and the robot at their respective coordinates.

In contrast to the Canvas, which was created with a specific size to cover a significant portion of the user interface, the map image was inserted into the canvas using its original size, which is typically much larger than the interface area. As a result, the complete visualization of the map required the use of the Canvas scrollbar or the implemented Zoom In/Zoom Out buttons as we will discuss further below.

With the map already rendered, the next step involved drawing the vertices. After retrieving the vertices coordinates from the Graph_Data and transforming their coordinates into the Canvas coordinate system, we simply created new Canvas objects with new shapes(in this case circles) using the Canvas API to position them on top of the map image canvas. For those vertices that had labels provided by the Labeled_Vertices topic, we added labels indicating their positions in the environment.

It is important to note that the map representation includes zoom-in and zoom-out buttons, as mentioned earlier. These buttons not only scale the map image up or down but also redraw the entire map, including the position of the vertices and the robot. This is necessary because when zooming in or out, the coordinates system scales accordingly, requiring adjustment of the vertices' and robot's positions in the map image to maintain accurate representation. Furthermore, when zooming in or out, the user is provided with a checkbox that allows him to choose whether to zoom in at the location of the robot or at the centre of the currently displayed portion of the map image.

### 5.3.5 Drawing the Robot

Having the map already drawn, we utilized the data from the Robot_Position topic to represent the movement of the robot within the map image. This topic provides information about the robot's position, with its coordinates represented as (x, y, z), and the orientation, which is given by a quaternion represented as (w, x, y, z), as shown in Figure 4.3a.

Similar to how we handled the vertices, we created a separate canvas on top of the map image canvas to represent the robot. However, directly representing the robot's orientation using the quaternion provided by the Robot_Position topic was not feasible, as we needed an angle to rotate the canvas representing the robot.

The following lines will illustrate the process of converting the quaternion to the rotation angle, required for rotating the robot on the interface.

Starting with the given values, w, x, y and z, the representation that can obtain for the quaternion is:

$$q = w + x\hat{i} + y\hat{j} + z\hat{k} \tag{5.1}$$

Since the given quaternion from the Robot_Position topic is already normalized, we don't need to perform normalization, which would involve dividing its components by its magnitude. Therefore, we can represent the quaternion as indicated in Equation 5.2:

$$q = w + \hat{x} + \hat{y} + \hat{z} \tag{5.2}$$

A unit quaternion can also be represented in the following form:

$$q = cos(\theta/2) + \hat{u}sin(\theta/2) \tag{5.3}$$

This equation represents the rotation of a 3D vector by an angle of $2\theta$ about the 3D axis $\hat{u}$. Thus, the angle of rotation is $\theta$, and the normalized vector is given by $/vu * u = /vu * x + /vu * y + /vu * z$. By equating the real parts, we have:

$$cos(\phi/2) = \hat{w} \tag{5.4}$$

Solving for $\theta$ we obtain the final value for the angle of rotation:

$$\theta = 2\arccos(\hat{w}) \tag{5.5}$$

However, this final representation of the angle, depicted in Equation 5.5, does not account for the orientation of the quaternion along the z-axis. This orientation is crucial as it determines whether the angle of rotation is considering a clockwise rotation (negative z) or a counter-clockwise rotation (positive z) around the z-axis. To address this, we multiplied the angle by the sign of the z-component of the quaternion to accurately represent the direction of rotation.

In addition, to ensure consistency and ease of use when assigning rotation angles to the robot canvas, we normalized the angle to fall within the range of -180º and 180º.

Having the position and orientation in the desired formats ((x, y) coordinates and an angle, respectively), we were able to accurately position the robot on the map with the correct orientation. To visually represent the robot, we created a simple image that serves as the background of the robot canvas. This image depicts a top-down view of the robot, accompanied by an arrow to aid in perceiving its orientation.

Note that the function responsible for positioning the robot on top of the map is invoked every 100 milliseconds to ensure a smooth display of the robot's movement. Additionally, this function not only represents the robot but also takes into account the presence of cargo. The information about the robot's cargo is provided by the Robot_Cargo topic, which returns a boolean value indicating whether the robot is currently carrying cargo or not.

Figure 5.6 provides an illustrative representation of the map in the HTML Canvas, displaying all the vertices along with their corresponding labels. The visualization also includes the representation of the robot within the map, giving a clear visual reference of its position and orientation.

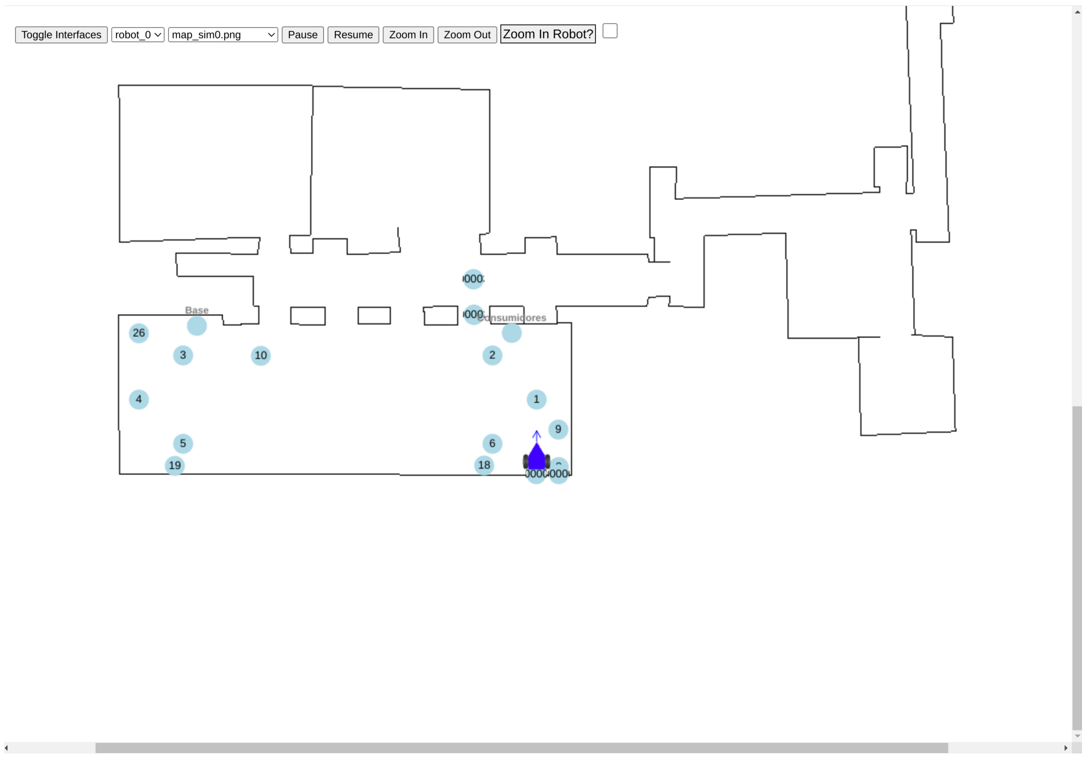


Figure 5.6: Representation of the Map and the Robot in the User Interface

As we can see, the section responsible for displaying the robot on the map contains a collection of buttons and selectors, each serving a specific purpose. Among them, two selectors play a vital role in choosing what robot and what map is being displayed on the interface.

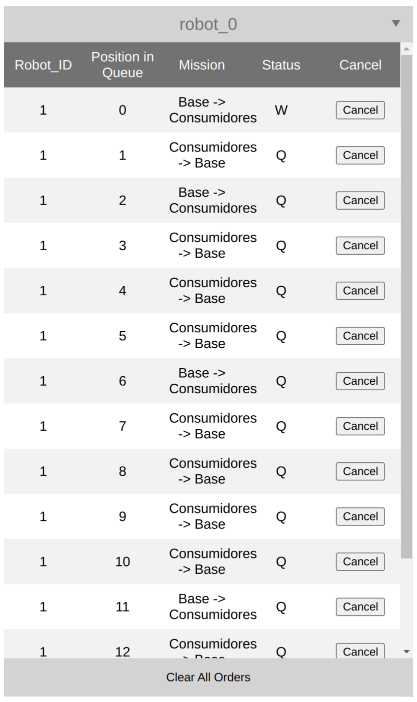
In our interface, only one robot is depicted at a time and this robot is chosen according to the option selected in the robot selector, located at the top section of our interface. Additionally, in the same top section, we have a map selector that enables the user to switch between different floors when the robot operates on multiple levels. This feature allows easy navigation and visualization of the map across different floors.

### 5.3.6 Displaying the Actions Queue

This section aims to provide the user with a simplified representation of the data obtained from the Robots_Actions_Queue topic. This topic contains information about the orders in the queue for each robot, including the current action being performed and any pending actions.

To visually represent this data, we organized it into a table. The table consists of five columns, each representing a specific piece of information for each order: the robot ID, the position of the order in the queue, the associated mission, the mission status, i.e., if the action is in queue or if it is being performed at that moment, and a cancel button that allows the user to cancel the respective order using the Cancel_Order topic, as explained later in this Chapter. Additionally, below the table, there is a "Clear All Orders" button that allows the user to cancel all orders of the robots, using the Clear_All_Orders service.

The orders are displayed with respect to the robot selected in a selector placed above the table, as we can see in Figure 5.7. Furthermore, they are displayed in the order of their position in the queue, with the current action being performed by the robot displayed first, followed by the remaining queued orders. To accommodate a potentially large number of orders, we have set a fixed size for the table and added a scroll bar, enabling the user to scroll through the table to view all the orders. Figure 5.7 provides an illustration of the table, showcasing its ability to accommodate a significant number of orders with the presence of a scroll bar for easy navigation and visualisation of all the orders sent to the robot.



robot_0

| Robot_ID | Position in Queue | Mission | Status | Cancel |
|---|---|---|---|---|
| 1 | 0 | Base -> Consumidores | W | Cancel |
| 1 | 1 | Consumidores -> Base | Q | Cancel |
| 1 | 2 | Base -> Consumidores | Q | Cancel |
| 1 | 3 | Consumidores -> Base | Q | Cancel |
| 1 | 4 | Consumidores -> Base | Q | Cancel |
| 1 | 5 | Consumidores -> Base | Q | Cancel |
| 1 | 6 | Base -> Consumidores | Q | Cancel |
| 1 | 7 | Consumidores -> Base | Q | Cancel |
| 1 | 8 | Consumidores -> Base | Q | Cancel |
| 1 | 9 | Consumidores -> Base | Q | Cancel |
| 1 | 10 | Consumidores -> Base | Q | Cancel |
| 1 | 11 | Base -> Consumidores | Q | Cancel |
| 1 | 12 | Consumidores | Q | Cancel |

Clear All Orders

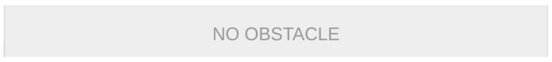
Figure 5.7: Representation of the Actions Queue for each robot

### 5.3.7 Displaying the Obstacle Detection

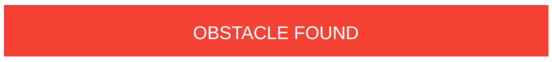
The representation of the obstacle detection section is a straightforward process, as its main purpose is to alert the user about the presence of obstacles that may obstruct the robot's path.

The relevant data for this representation is obtained from the Obstacle_Detection topic, which provides information about the duration, in seconds, that the robot has been in contact with an obstacle. When there are no obstacles detected, the topic publishes a value of 0. However, when an obstacle is detected, the value starts increasing until the obstacle is cleared, at which point the value returns to 0.

To visually represent the obstacle detection warning, this interface section turns red and displays a warning text when the duration provided by the topic exceeds three seconds. This indicates that the robot has either collided with an obstacle or detected one and has remained stationary for a period longer than three seconds. Once the obstacle is removed and the robot resumes its normal movement without detecting any obstacles, the obstacle detection section returns to a neutral colour, such as grey, to minimize attention. Figure 5.8 illustrates these representations, with Figure 5.8a showing the neutral representation and Figure 5.8b showcasing the warning state.



(a) Neutral representation of the obstacle warning when there is no obstacle



(b) Warning representation of the obstacle warning when there is an obstacle

Figure 5.8: Obstacle Detection Representation

### 5.3.8 Displaying the Robot Status

This section of the interface serves to display relevant information about the status of the robots. It uses the same image as shown on the map to visually represent each robot and provides information about their online/offline status. This is determined by checking if the REST API is receiving data from the respective robot, as depicted in Chapter 4. If data is not received for a certain period of time, the robot is considered offline.

Furthermore, the section utilizes data from the Robot_Battery topic to create a virtual representation of the robot's battery. The battery is represented by a simple visual illustration with a battery image with a number inside of it, provided by the topic, that represents the battery percentage of the robot. Additionally, if the battery percentage falls below 20%, the colour representing the current battery percentage is red, while if it is above 20%, the colour of the battery is green.

Figure 5.9 showcases both these cases, with Figure 5.9a displaying the battery in green on the first robot and the battery in red on the second robot. Furthermore, Figure 5.9b illustrates the case where a robot is considered offline, after not sending messages for a certain period of time.

By incorporating these representations, users can easily comprehend the status of the various robots displayed on the interface. Additionally, the battery information also serves to notify users

(a) Representation of the battery of the robots

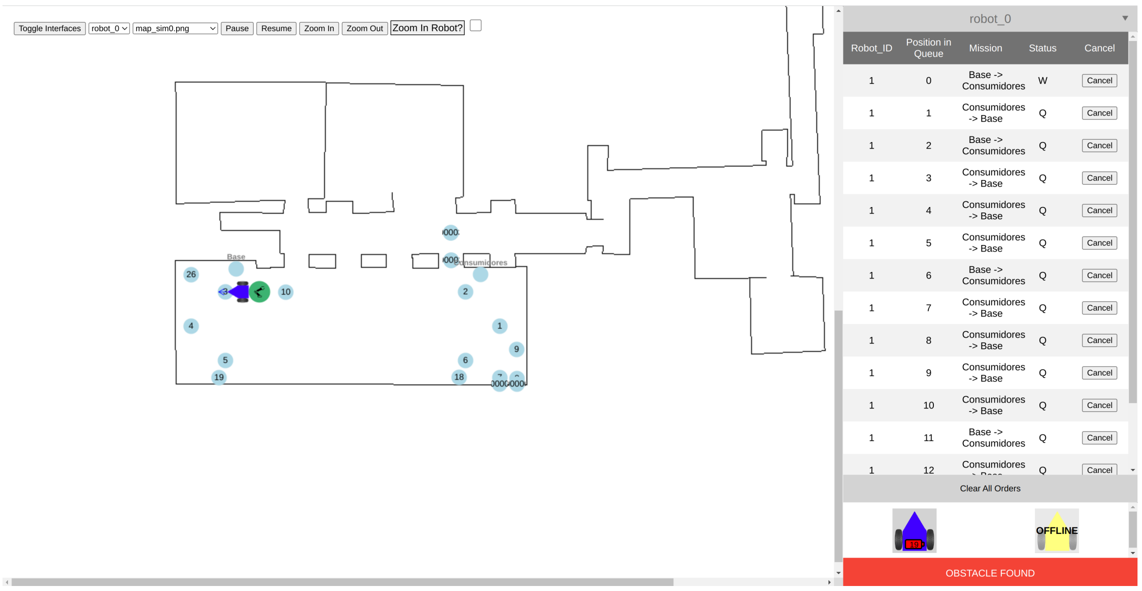
(b) Representation of the online status of the robots

Figure 5.9: Robots Status Representation

when a robot should be docked for charging.

Note that, as mentioned earlier, the information in these last three sections is updated every 500 milliseconds. This update process involves removing the previous information displayed and replacing it with the new information obtained in each iteration. This ensures that the user is provided with the most up-to-date data at regular intervals.

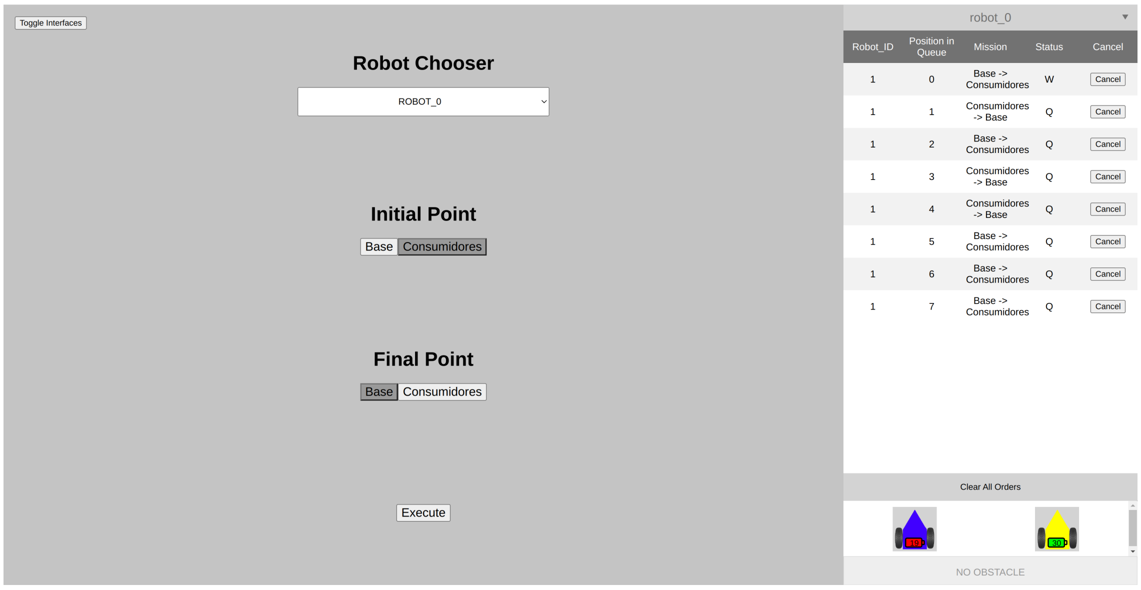
The final result of this main interface can be observed in Figure 5.10, which showcases a comprehensive integration of the various elements. It combines the primary representation of the robots within the map, accompanied by the zooming functionalities, as well as the inclusion of secondary representations mentioned earlier, such as the table displaying the robot orders, the obstacle detection warning section and the robot status section.



| Robot_ID | Position in Queue | Mission | Status | Cancel |
|---|---|---|---|---|
| 1 | 0 | Base -> Consumidores | W | Cancel |
| 1 | 1 | Consumidores -> Base | Q | Cancel |
| 1 | 2 | Base -> Consumidores | Q | Cancel |
| 1 | 3 | Consumidores -> Base | Q | Cancel |
| 1 | 4 | Consumidores -> Base | Q | Cancel |
| 1 | 5 | Consumidores -> Base | Q | Cancel |
| 1 | 6 | Base -> Consumidores | Q | Cancel |
| 1 | 7 | Consumidores -> Base | Q | Cancel |
| 1 | 8 | Consumidores -> Base | Q | Cancel |
| 1 | 9 | Consumidores -> Base | Q | Cancel |
| 1 | 10 | Consumidores -> Base | Q | Cancel |
| 1 | 11 | Base -> Consumidores | Q | Cancel |
| 1 | 12 | Consumidores | Q | Cancel |

Figure 5.10: Representation of the main interface

In addition to the primary interface, an alternative user-friendly interface has been developed to enhance communication with the robot. This simplified interface, illustrated in Figure 5.11, provides users with a well-organized and intuitive experience, allowing for easy selection of the desired robot and specification of the initial and final points for order dispatch. It is important to consider that when the user fails to select the initial or final point, the interface alerts the user of that and indicates what field is missing on the order. Furthermore, if the robot has already cargo and has no pending orders, this interface only allows the user to select the destination to where the robot will drop off the cargo.



| Robot_ID | Position in Queue | Mission | Status | Cancel |
|---|---|---|---|---|
| 1 | 0 | Base -> Consumidores | W | Cancel |
| 1 | 1 | Consumidores -> Base | Q | Cancel |
| 1 | 2 | Base -> Consumidores | Q | Cancel |
| 1 | 3 | Consumidores -> Base | Q | Cancel |
| 1 | 4 | Consumidores -> Base | Q | Cancel |
| 1 | 5 | Base -> Consumidores | Q | Cancel |
| 1 | 6 | Base -> Consumidores | Q | Cancel |
| 1 | 7 | Base -> Consumidores | Q | Cancel |

Figure 5.11: Representation of the simple(secondary) interface

The purpose of this interface is to offer an alternative approach to interact with the robot, distinct from the map-based representation found in the primary interface. By also incorporating secondary representation, such as the robots' actions queue, obstacle detection and the robot status within this interface, users will have access to the same information presented in the main interface without the hassle of trying to find a specific vertex to issue an order.

The development of this user-friendly interface addresses the need for a more intuitive and efficient means of communicating with the robot. By providing a clear and focused interface, users can quickly navigate through the available options and easily input their desired instructions. Therefore, this interface significantly enhances the overall usability and effectiveness of the application.

## 5.4 Service Invocation and Robot Interaction

As previously mentioned, the communication between the user interface and the REST API relies on HTTP requests. While we discussed how the REST API sends data to the user interface using GET requests, we haven't yet addressed how the user interface can send data to the REST API. In this case, the data transmission from the user interface to the REST API occurs through POST requests.

These POST requests are made through the user interface utilizing the Fetch API provided by JavaScript. The Fetch API serves as a modern alternative to the older XMLHttpRequest object, which was used in this user interface for the GET requests. With the Fetch API, the user interface can specify the server endpoint, the desired HTTP request method (in this case, POST), and the data to be sent to the REST API in JSON format. The REST API, on the other hand, receives the data through the designated endpoints with their associated URLs and leverages the ROS services to transmit the data to the ROS Master.

In this section, we will focus on the process of data transmission from the user interface to the REST API. Specifically, we will explore the different cases and scenarios where this communication occurs through POST requests.

### 5.4.1 Trajectory Orders to the Robot

To facilitate the process of giving trajectory orders to the robot using the Execute_Manual_Order service, we implemented a feature where the labelled vertices on the map are clickable. As shown in Figure 5.12, when a labelled vertex is clicked, a small menu opens on the right side of its location, allowing the user to select the desired robot and specify the final destination. This interaction provides the necessary data, such as the initial point (the ID of the clicked vertex), the robot ID, and the final point, which is sent to the REST API to trigger the Execute_Manual_Order service.

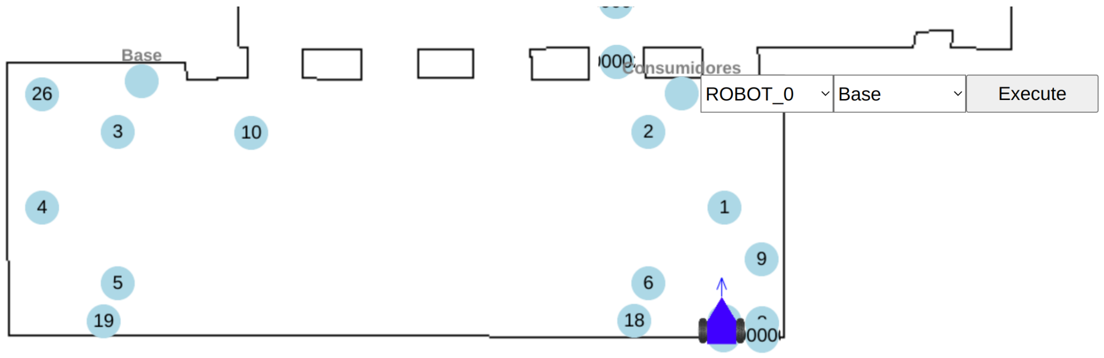
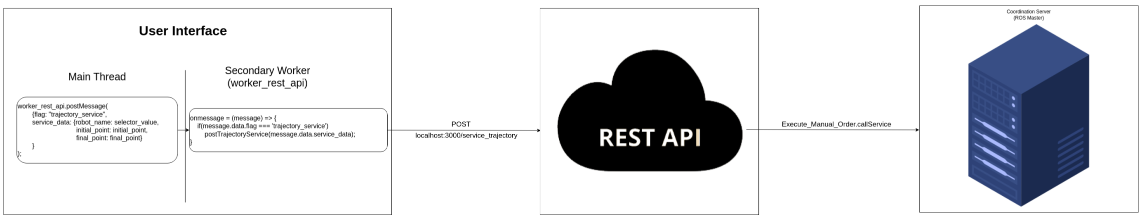


Figure 5.12: Representation of the menu to issue a trajectory order

Figure 5.13 provides an illustration of the communication flow among the workers, the REST API and the ROS Master, encompassing the entire process from the user clicking the "Execute" button to the reception of data by the robot.



```
worker_rest_api.postMessage(
    {flag: "trajectory_service",
    service_data: {robot_name: selector_value,
                  initial_point: initial_point,
                  final_point: final_point}
    }
);
```

```
onmessage = (message) => {
  if(message.data.flag === 'trajectory_service')
     postTrajectoryService(message.data.service_data);
}
```

Figure 5.13: Trajectory Order to the Robot without cargo

Firstly, the primary worker sends a message containing the associated flag and required data to the secondary worker. The secondary worker then utilizes the Fetch API to perform a POST request to the specific URL within the REST API endpoint. As a result, the user interface can send data to the REST API in JSON format. The REST API receives this data through the designated endpoint, creates a Request object using the received data, and proceeds to call the Execute_Manual_Order service, giving an order to the robot.

A very similar process occurs when the robot is carrying cargo but does not have any order associated with it. In this scenario, the robot is stationary as it does not have a designated destination to deliver the cargo. As depicted in Figure 5.14, when clicking on a vertex, the small menu that appears includes only the robot selector and a "Come Here" button. This is because the robot simply needs a final point to unload the cargo.

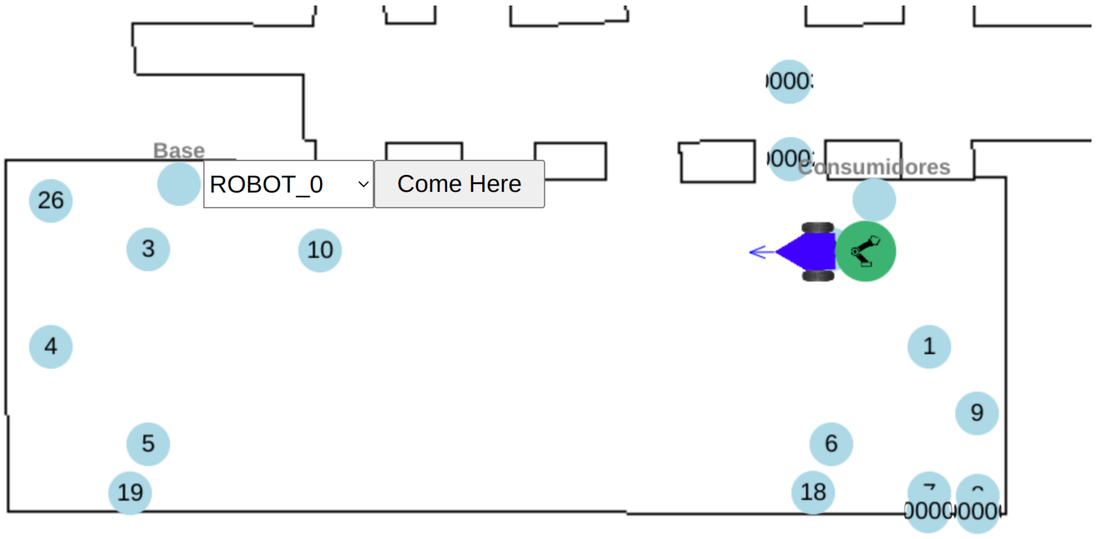
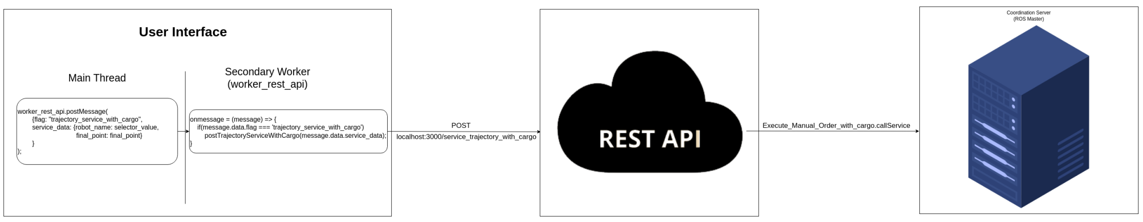


Figure 5.14: Representation of the menu to issue a trajectory order with cargo

Figure 5.15 illustrates a similar communication flow as shown in Figure 5.13, with slight differences in the data passed in the POST request, the URLs associated with the requests and the name of the ROS service, in this case, Execute_Manual_Order_with_cargo.



```
worker_rest_api.postMessage(
        {flag: "trajectory_service_with_cargo",
        service_data: {robot_name: selector_value,
                        final_point: final_point}
        }
);
```

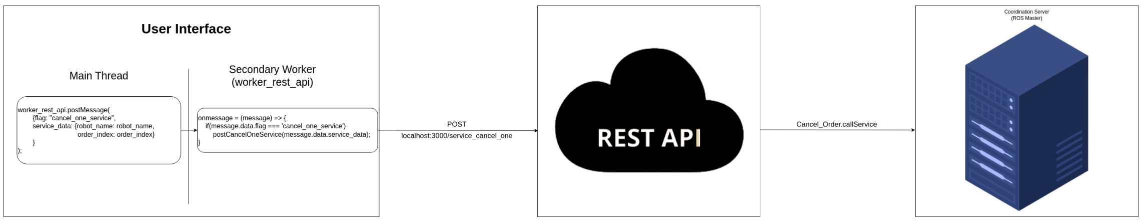
```
onmessage = (message) => {
    if(message.data.flag === 'trajectory_service_with_cargo')
        postTrajectoryServiceWithCargo(message.data.service_data);
}
```

Figure 5.15: Trajectory Order to the Robot with cargo

## 5.4.2 Cancellation of Orders

As mentioned earlier, the table containing information about the actions queue of the robots includes a "Cancel" button in each row to cancel individual orders. Additionally, there is a "Clear All Orders" button that allows for the cancellation of all orders associated with the robot.

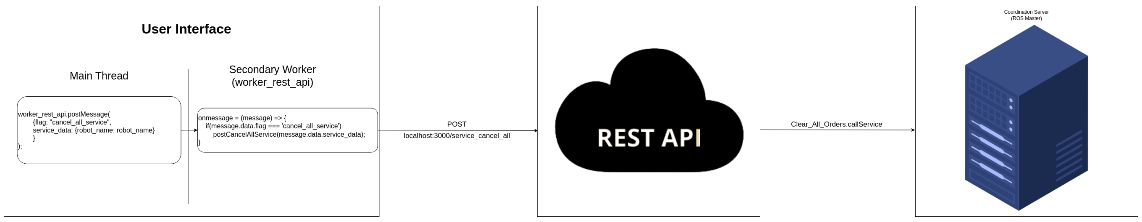
Figures 5.16 and 5.17 illustrate the process of cancelling one order and cancelling all orders, respectively. Both processes follow a similar structure, with differences in the URL of the POST request and the name of the ROS service. The main distinction lies in the arguments passed to the ROS Master. The Cancel_Order service requires the robot name and the index of the respective order as arguments, whereas the Clear_All_Orders service only requires the robot name as an argument.



```
worker_rest_api.postMessage(
        {flag: "cancel_one_service",
        service_data: {robot_name: robot_name,
                        order_index: order_index}
        }
);
```

```
onmessage = (message) => {
    if(message.data.flag === 'cancel_one_service')
        postCancelOneService(message.data.service_data);
}
```

Figure 5.16: Process of Cancelling of One Order



```
worker_rest_api.postMessage(
        {flag: "cancel_all_service",
        service_data: {robot_name: robot_name}
        }
);
```

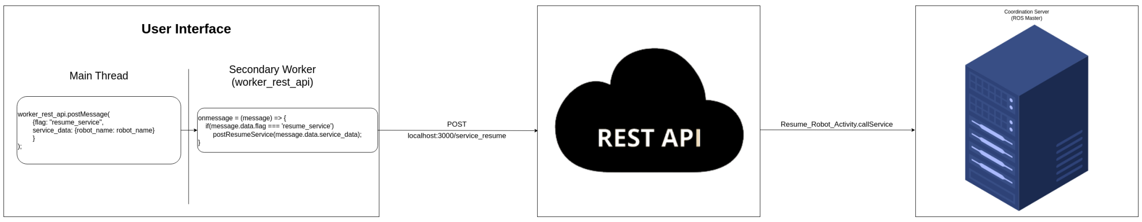
```
onmessage = (message) => {
    if(message.data.flag === 'cancel_all_service')
        postCancelAllService(message.data.service_data);
}
```

Figure 5.17: Process of Cancelling All Orders

### 5.4.3 Pause and Resume the Robot Activity

As previously stated, there are two services available to control the robot's activity: Resume_Robot_Activity and Pause_Robot_Activity. These services are activated by clicking the "Resume" and "Pause" buttons located at the top of the main interface, on top of the map image.

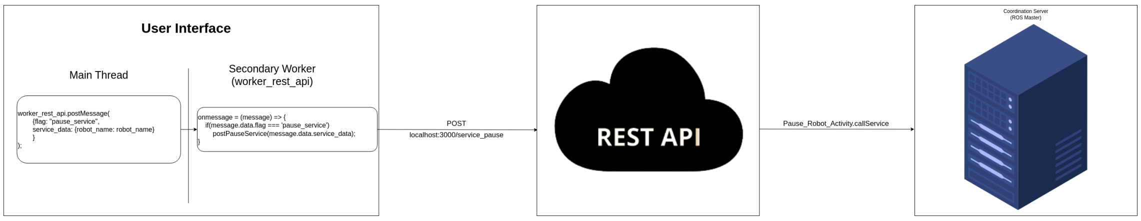
Figure 5.18 and 5.19 illustrate the process of activating these services, starting from the button click on the interface and ending with the reception of data on the ROS Master side, through the services already mentioned.



```
worker_rest_api.postMessage(
        {flag: "resume_service",
        service_data: {robot_name: robot_name}
        }
);
```

```
onmessage = (message) => {
    if(message.data.flag === 'resume_service')
        postResumeService(message.data.service_data);
}
```

Figure 5.18: Process of Resuming the Robot Activity


User Interface
Main Thread
Secondary Worker (worker_rest_api)
POST
localhost:3000/service_pause
REST API
Pause_Robot_Activity.callService
Coordination Server (ROS Master)

```
worker_rest_api.postMessage(
      {flag: "pause_service",
      service_data: {robot_name: robot_name}
      }
);
```

```
onmessage = (message) => {
   if(message.data.flag === 'pause_service')
      postPauseService(message.data.service_data);
}
```

Figure 5.19: Process of Pausing the Robot Activity

In conclusion, this chapter, together with Chapter 4, has provided a comprehensive overview of the approach and techniques employed in the development of our dissertation project. By following a structured methodology, starting from the setup and conceptualization phases and progressing through the data retrieval and representation stages, we have ensured an efficient workflow. This approach ensured a logical sequence of tasks throughout the application development process, with clear interconnections between each stage.

# Chapter 6

# Results and Discussion

This chapter presents a comprehensive analysis of the achieved goals and the performance of the developed interface. It begins with a qualitative comparison of this interface with the three previously developed interfaces, focusing on aspects such as usability, accessibility, user experience and available features. This evaluation aims to highlight the advantages brought by the implementation of the web technologies employed in this interface, and how they contribute to the project.

Furthermore, a quantitative comparison is conducted to assess the performance of the developed web interface. This comparison involves measuring and analysing various metrics to identify any improvements or limitations compared to the existing web interface, which has previously been acknowledged to have certain limitations.

Through this evaluation, which can be confirmed in the video in Appendix A, we will critically examine whether the initially defined goals have been achieved. This discussion will delve into the strengths and limitations of the web application, taking into consideration factors such as efficiency, scalability, etc.

By carefully examining the results and engaging in meaningful discussions, this chapter aims to provide a comprehensive evaluation of the developed interface, by providing insights on its effectiveness and contribution, both in terms of advancing the state-of-the-art in robot web applications and addressing the specific requirements of the current project.

## 6.1 Qualitative Analysis

In this section we will conduct a comprehensive evaluation of our application, assessing its improvements and limitations, and comparing it to the three previously developed interfaces. The evaluation will primarily focus on factors such as efficiency, usability, accessibility and available features.

The incorporation of web technologies in this project has brought significant advantages that were not feasible with the RViz-based interfaces. By utilizing the *rosbridge* protocol, the project was divided into two separate parts: the application layer, which encompasses the user interface and associated functionalities, and the software layer responsible for controlling the robots and their related operations. This division allows developers to work concurrently on the application, each with their respective responsibilities, without impacting each other's work. Moreover, this modular approach facilitated the integration of both parts into the overall project, resolving integration challenges that arose when combining the robot control software and the interface software.

Additionally, the inclusion of web technologies has greatly enhanced usability and accessibility. Cross-platform compatibility is one example of these improvements. The web technologies enabled the development of the existing web interface to run seamlessly on various platforms, including desktop computers, smartphones and tablets.

Our web application takes this feature even further. By separating the REST API from the user interface, it becomes possible to deploy these components on different servers, and most important, different devices. If the REST API and the user interface are hosted on devices connected to the same network, establishing the connection between them is straightforward using the device's IP address.

This differentiation is a key advantage of our application compared to the existing web interface in this project. It allows for the installation of all the necessary software for controlling the robots, along with the REST API, on a dedicated local server, while the user interface can be installed on a separate device such as a laptop or a tablet, without the need to install all the robot-related and REST API software.

Another significant benefit that web technologies bring to the project is scalability and extensibility. As the needs and requirements of robot interfaces evolve, web technologies provide the flexibility to add new features by integrating with third-party services. The rapid expansion of web technologies opens opportunities for integration with other web-based tools, databases, APIs, and cloud services, facilitating project scalability.

This scalability advantage is not only beneficial for our interface but also distinguishes it from both the RViz-based interfaces and the existing web interface. With the creation of the autonomous REST API, independent of the user interface, multiple web-based tools can be developed around the REST API. These tools can include configuration web pages for the project setup phase, add-ons for the existing web interfaces, or even mapping tools for the mapping mode.

Overall, web technologies and our interface bring numerous advantages to the project. The only notable drawback is the presence of multiple layers in the application, which necessitates various connections such as the WebSocket connection between the ROS Master and the REST API, as well as the TCP/IP connection between the REST API and the user interface. These connections may introduce some overhead, potentially slowing down data transmission compared to a single-system approach (as stated in the Quantitative Analysis section) where data doesn't need to be transmitted through WebSocket or TCP/IP connections.

## 6.2 Feature Analysis

In this section, we will discuss and compare the features associated with each interface, assessing their significance for the project and highlighting any improvements made in the interface presented in this dissertation.

Table 6.1 provides a compilation of features across all three interfaces, mapping out the presence or absence of each feature in each interface. This comparative analysis allows for a comprehensive evaluation of the feature set offered by each interface, enabling a clear understanding of the unique strengths and limitations of each implementation.

| | **RViz** | **Iris** | **Old Web Interface** | **New Web Interface** |
|---|---|---|---|---|
| Monitorization of the Robot | ✓ | ✓ | ✓ | ✓ |
| Zooming In/Out | ✓ | ✓ | ✓ | ✓ |
| Display of the Map's Edges | ✓ | ✓ | ✗ | ✗ |
| Display of the Map's Vertices | ✓ | ✓ | ✓ | ✓ |
| Add/Remove Vertices or Edges | ✗ | ✓ | ✗ | ✗ |
| Changing Vertice's Type | ✗ | ✓ | ✗ | ✗ |
| Display of Multiple Robots | ✓ | ✓ | ✗ | ✓ |
| Switch Between Maps/Floors | ✓ | ✓ | ✗ | ✓ |
| Display of a Simple Interface to issue Orders | ✗ | ✗ | ✗ | ✓ |
| Give Orders to the Robots -> Initial + Final Point | ✗ | ✗ | ✓ | ✓ |
| Possibility of Canceling Orders | ✗ | ✗ | ✓ | ✓ |
| Display of the Queue of Orders for each robot | ✗ | ✗ | ✓ | ✓ |
| Toggle between Mapping Mode and Monitorization Mode | ✗ | ✓ | ✗ | ✗ |
| Real-time Mapping of the Environment | ✗ | ✓ | ✗ | ✓ |
| Obstacle Detection Display | ✗ | ✗ | ✓ | ✓ |
| Robots Status Display | ✗ | ✗ | ✓ | ✓ |
| Automatically Reconnection to the ROS Master | ✗ | ✗ | ✗ | ✓ |

Table 6.1: Interfaces' set of Features

As previously mentioned, the RViz interface comprises the most fundamental features, while the Iris interface serves as an enhanced version of the former. Although the Iris interface supports mapping mode, unlike the other three interfaces, mapping is not an objective of our application's development. Moreover, this feature is primarily associated with the configuration phase, which is not directly accessible to users upon project deployment. Consequently, there is no immediate need to develop a web interface specifically for mapping mode.

Furthermore, our web interface not only performs the tasks already accomplished by the old web interface but also incorporates additional functionalities to further enhance the project. One of these extra features is the simple interface, which, as mentioned earlier, allows the user to give orders to the robot through a simple interface. Another main differentiator is the scalability of our application, enabling users to switch between different maps and floors of the environment and seamlessly display multiple robots within a single interface.

A significant additional feature, as discussed in Chapter 4, is the automatic reconnection to the ROS Master on the REST API side. This eliminates the need for a specific order in starting the ROS Master, the REST API and the user interface, providing convenience to the user. With this enhancement, users can launch the REST API and the user interface independently, without the requirement of a specific startup sequence or the need to restart the ROS Master. This flexibility simplifies the usage of the web application and improves the overall user experience.

Another differentiator between our web application and the old web interface is the display of the robot's status. As mentioned in Table 6.1, both user interfaces have the robot's status available. However, unlike the old web interface, our new interface utilizes the mechanism implemented in the REST API, described in Chapter 4, to accurately detect if the robot has gone offline for any reason. This enables us to display the status reliably.

## 6.3 Quantitative Analysis

Moving on to the quantitative analysis, in this section, we will conduct a comparative assessment between our web application and the existing user interface. Specifically, we will compare these interfaces based on their time efficiency, which refers to the responsiveness of the interface to various actions such as drawing the robot and issuing an order to the robot. By evaluating this factor, we can gain insights into the performance and effectiveness of our web application in comparison to the already-developed user interface.

To measure the efficiency of our interface, we utilized the browser's (in this case Google Chrome) built-in performance tool, which allowed us to track the execution time of specific tasks and compare them with the already-developed interface.

### 6.3.1 Time Analysis for Drawing the Robot

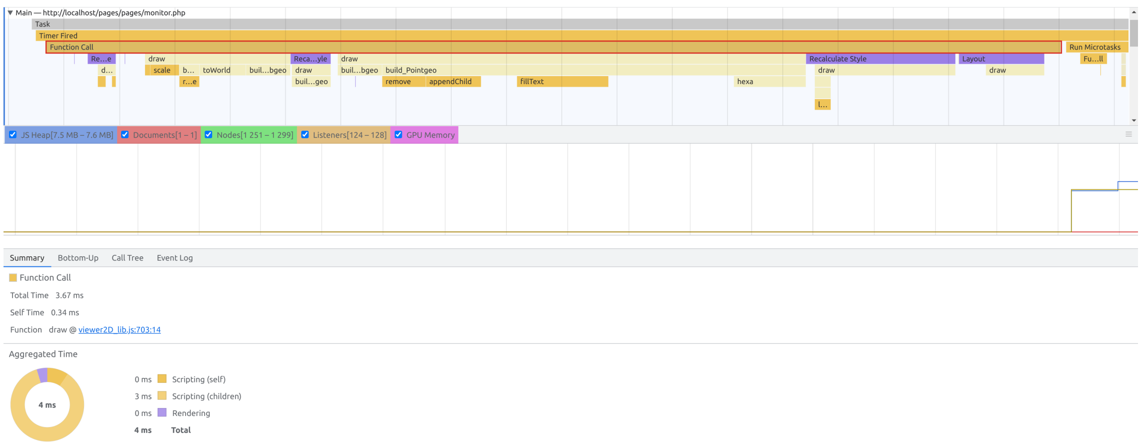
In this task, we focused on measuring the time taken by each interface to render and display the robot on top of the map. It is important to mention that our user interface performs this function every 100 milliseconds whereas the other user interface does so every 500 milliseconds. Therefore, this task holds significant importance when analysing the overall efficiency of the interfaces, as it needs to be performed periodically throughout the entire lifecycle of the interface.

The increased frequency in our user interface ensures a smoother and more fluid experience for the user when visualizing the movement of the robot within the map. Despite the augmented frequency, we carefully considered the potential issue of overloading the main thread. However, this concern can be disregarded due to the relatively small time period required for this function, which is significantly smaller than the interval at which the function is invoked. On the other hand, it doesn't make sense to increase even more the frequency, as the robot only sends messages in a period of time of approximately 100 milliseconds.

Figures 6.1 and 6.2 depict the time consumed by the old interface and the new interface to represent the robot, respectively.

As we can see, our user interface has a much smaller interval to display the robot, with an approximate interval of 1.1 milliseconds. In contrast, the old interface takes nearly four times longer compared to our user interface, with a time of 3.67 milliseconds.

By analyzing both figures, the first thing we observe is the number of functions called within the process of drawing the robot for each interface. In our interface, the drawing of the robot



Function Call
Total Time 3.67 ms
Self Time 0.34 ms
Function draw @ viewer2D_lib.js:703:14

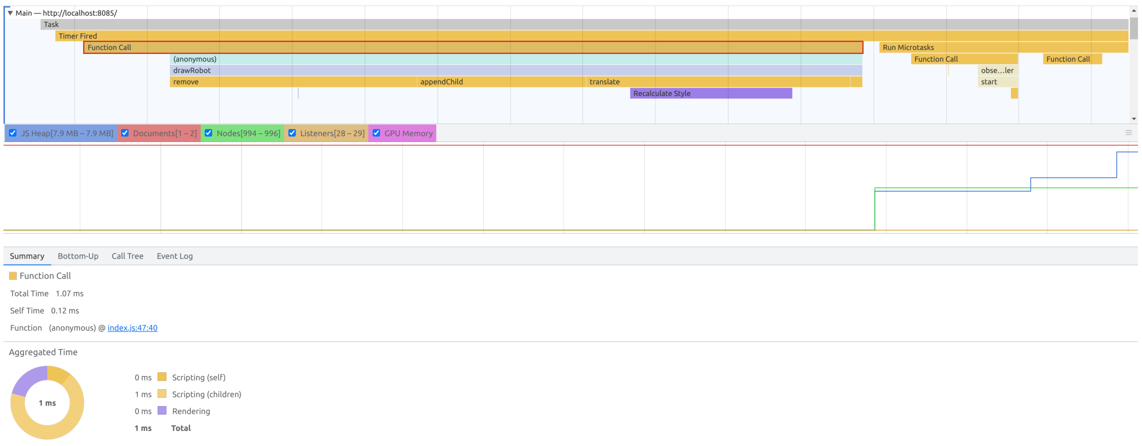
| | |
|---|---|
| 0 ms | Scripting (self) |
| 3 ms | Scripting (children) |
| 0 ms | Rendering |
| **4 ms** | **Total** |

Figure 6.1: Time analysis for drawing the robot in the old interface

only requires methods provided by JavaScript, such as *remove* (to remove the previous occurrence of the robot) or *apppendChild* (to append the robot to the map canvas). On the other hand, the process of drawing the robot in the old interface is much more complex. The function is divided into multiple parts, necessitating the invocation of various functions within the main function to draw the robot. This results in a much longer process, with much more overhead to the function that blocks the main thread for a longer period of time.

Although, it is important to consider that the time period of 3.67 milliseconds also includes the drawing of vertices and labels on the map, which significantly contribute to this prolonged duration. In our application, the drawing of the vertices of the application takes approximately 10 milliseconds. Adding both these time periods would give a total time of 11.1 milliseconds in our new interface to represent both the vertices and the robot. However, this does not give the whole picture of the efficiency of our application.

In the old user interface, the vertices and the robot were removed and displayed every 500 milliseconds. However, in our new interface, we only redraw the vertices when the user zooms in



Function Call
Total Time 1.07 ms
Self Time 0.12 ms
Function (anonymous) @ index.js:47:40

| | |
|---|---|
| 0 ms | Scripting (self) |
| 1 ms | Scripting (children) |
| 0 ms | Rendering |
| **1 ms** | **Total** |

Figure 6.2: Time analysis for drawing the robot in the new interface

or out or switches between robots or floors. This represents a substantial improvement over the old interface, as the vertices remain static without the need for continuous updates. Consequently, there is no unnecessary burden placed on the main thread, avoiding unnecessary blocking.

Another contributing factor to the delayed representation of the robot in the old interface, when compared to our interface, is the data retrieval process. In our application, the data is retrieved by a secondary worker/thread and then stored in global variables accessible to the main worker at all times. This allows for quick and efficient access to up-to-date information regarding the robot's position.

In contrast, the old interface, having only one thread, retrieves data every time the function to draw the robot is called in order to obtain the most recent information regarding the robot's position. This additional data retrieval step adds to the overall processing time, resulting in a longer delay in updating the robot's representation.

### 6.3.2 Time Analysis for Issuing an Order

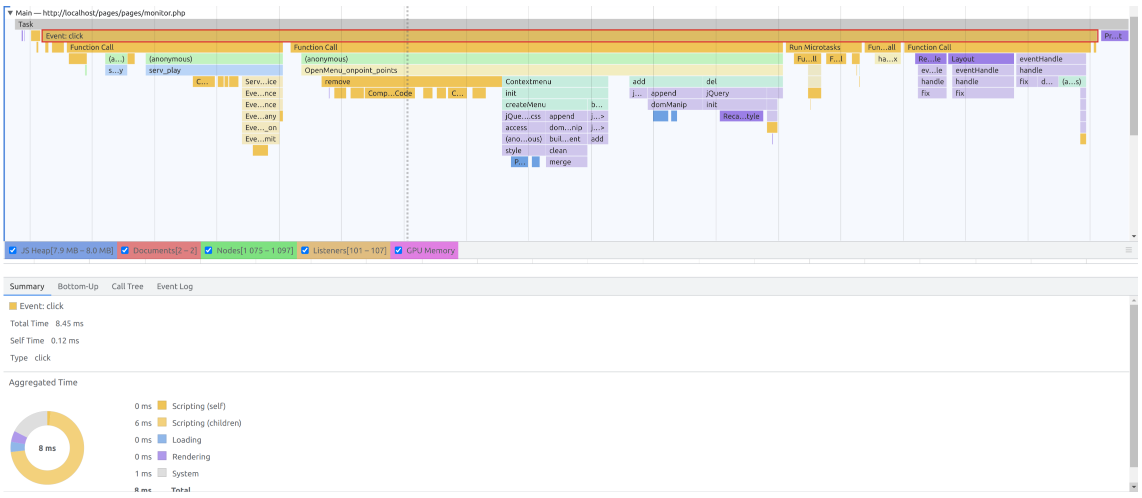
Another crucial aspect to consider when comparing both interfaces is the handling of robot orders. In both interfaces, these orders are sent using the *callService* method provided by the *roslibjs* library.

To accurately assess the performance of both web applications, we needed to compare the time elapsed from clicking the "Execute" button until the actual *callService* invocation. However, analyzing our new interface alone does not provide a comprehensive understanding of the time required from the button click to the service call. This is because the user interface interacts with the ROS Master through an intermediate level represented by the REST API.

Nevertheless, when examining Figure 6.4, we can deduce, by the time representation on top of the image, that approximately 2 milliseconds have elapsed from the button click to the posting of the message to the secondary worker and subsequently to the REST API. On the other hand, Figure 6.3 reveals that a considerable 8 milliseconds have passed from the button click until the service call in the old interface, as seen in the bottom part of the image.

It is evident that this is not a fair comparison, as the transmission of messages between the user interface and the REST API would likely take more than 8 milliseconds. However, it demonstrates that despite the potential latency compared to the old interface, the main thread remains blocked for an almost negligible period of time. This allows for the execution of other tasks while the communication between the secondary worker, the REST API and the robot occurs.

Through both qualitative and quantitative comparisons of the interfaces, it is evident that the inclusion of multiple layers in the application introduced some delay in the transmission of messages between the robot and the user interface, having to pass through the REST API at an intermediate level. However, the modular design of our application brings significant advantages to the project, particularly in terms of accessibility and usability. Although there is a slight delay in the communication between the robot and the user interface, it does not have a critical impact on the overall user experience, as the difference is only in the milliseconds range. Furthermore, this



Event: click
Total Time 8.45 ms
Self Time 0.12 ms
Type click

Aggregated Time

| | |
|---|---|
| 0 ms | Scripting (self) |
| 6 ms | Scripting (children) |
| 0 ms | Loading |
| 0 ms | Rendering |
| 1 ms | System |
| 8 ms | Total |

Figure 6.3: Time analysis for issuing an order in the old interface

augmented latency is distributed across two threads, which represents a significant improvement from the old user interface. The blocking time of the main thread is drastically reduced in many operations, as demonstrated when issuing an order to the robot.

Nevertheless, as highlighted in this chapter, substantial efforts have been made to minimize the data transmission time between the robot and the user interface. One example of these is the optimized representation of vertices, where they are only rendered every time the user tries to zoom in or zoom out on the map, instead of every time the robot is displayed, as was the case in the old interface.

Concluding, while there may be minor delays in communication, the benefits gained in terms of accessibility, usability and possible scalability outweigh any negligible impact on the overall user experience.

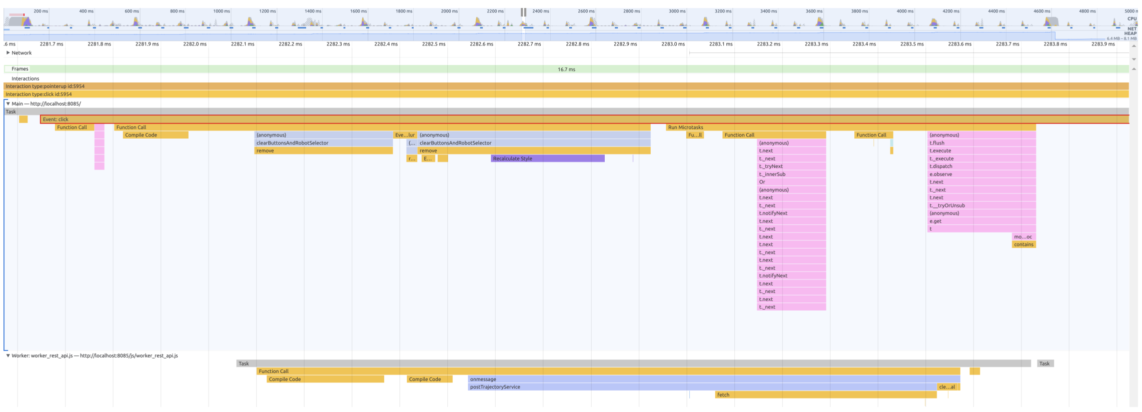


Figure 6.4: Time analysis for issuing an order in the new interface

# Chapter 7

# Conclusions

Throughout this research, a comprehensive understanding of the requirements and challenges associated with robot interfaces was attained. By conducting an extensive literature review and examining existing interfaces, we were able to identify areas for improvement and determine the most suitable technologies to integrate into our web interface.

This research aligns itself with the state-of-the-art of robot applications by exploring the relatively new space opened by the *rosbridge* protocol. By leveraging this emerging technology, this research demonstrates the potential of integrating web technologies into the field of robotics. Furthermore, we capitalize on the rapidly evolving world of web development, specifically utilizing frameworks to optimize code efficiency and open up the world of robotics to developers who may not have a background in robotics.

Through the processes described in Chapters 4 and 5, we successfully developed an interface that fulfils the initially defined requirements. By comparing the interface with previous implementations in this project, both qualitatively and quantitatively, we conclude that there have been significant improvements. By harnessing web technologies, we have overcome limitations and created an interface that enables control, monitoring and communication with mobile robots while enhancing key factors that were lacking in the previous interfaces, such as accessibility, usability and scalability.

Additionally, this web application, by separating the REST API from the user interface represents a huge improvement in scalability when compared to the previous interfaces. This enables the project to scale up by adding many modules around the intermediate level, the REST API, without affecting the performance of the latter or of the other components of the entire application. This separation allows for easier maintenance and extensibility, making it possible to integrate additional functionalities and modules without disrupting the core functionality of the interface.

It is important to acknowledge the limitations of this research. Although the web interface has shown promising results, there are still performance challenges, particularly in terms of communication efficiency among the various layers of the web application. This implies that the interface may undergo further modifications in the future to optimize performance or even to expand its functionalities.

For instance, a desirable future enhancement could be the capability to represent multiple robots simultaneously, providing users with a user-friendly interaction for monitoring multiple robots without the need to switch between sections on the web page. This feature was not implemented in our interface as it was assumed that the graph received for each robot is unique to that specific robot. However, it can be modified in the future to accommodate situations where the same graph is desired for multiple robots.

Another limitation that our web application may encounter in the future is its efficiency. Currently, we are using the *setInterval* method provided by JavaScript to update the data displayed in the interface. However, this approach has a drawback as it blocks the main thread each time the functions within the *setInterval* method are invoked. Although this is not a problem for our relatively small user interface, it could become problematic if we need to accommodate more information or additional features to the point that could overload the main thread.

To overcome this potential issue, we can integrate a JavaScript frontend library or framework. These modern tools come with built-in methods that efficiently update the DOM(Document Object Model) whenever a variable is modified, surpassing the capabilities of plain JavaScript. By utilizing such a library or framework, the user interface can update only the necessary parts of the interface whenever the data variables are modified through the secondary worker, thereby reducing the burden on the main thread.

Additionally, some frontend libraries and frameworks are specifically designed for single-page applications, which further enhances the efficiency of updating the represented data. In such cases, the application has the advantage of updating only the relevant content, further optimizing the performance and responsiveness of the interface.

Moreover, as the field of robotics continues to evolve, future research can aim to further enhance the overall project within which this dissertation is situated. With the growing Robot Web Tools community, there is potential for the software required to communicate with robots to be implemented in JavaScript, moving away from the current reliance on C++ or Python as the primary development languages. This shift would align with the advancements in web technologies and open up new possibilities for seamless integration with the web interface.

Lastly, conducting evaluations of the web interface in real-world scenarios and actively gathering user feedback will offer invaluable insights for the ongoing refinement of the application. By soliciting input from users and observing how the interface performs in practical settings, we can identify areas for improvement and ensure the application remains aligned with the needs and expectations of its intended users.

In conclusion, this dissertation has made significant contributions to the advancement of web interface development for mobile robots, bridging the gap between humans and machines and opening up new possibilities for human-robot interactions.

# Appendix A

# Videos

## A.1 Demonstration Video

https://www.youtube.com/watch?v=WRx_9ifH1W8

# References

[1] Christopher Crick, Graylin Jay, Sarah Osentoski, Benjamin Pitzer, and Odest Chadwicke Jenkins. *Rosbridge: ROS for Non-ROS Users*, pages 493–504. Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-29363-9_28.

[2] Javier Verdú and Alex Pajuelo. Performance scalability analysis of javascript applications with web workers. *IEEE Computer Architecture Letters*, 15(2):105–108, 2016. doi:10.1109/LCA.2015.2494585.

[3] 2022 stack overflow developer survey. https://survey.stackoverflow.co/2022/#section-most-popular-technologies-web-frameworks-and-technologies. Accessed: 2022-12-20.

[4] Sai Sahith Velamala, Devendra Patil, and Xie Ming. Development of ros-based gui for control of an autonomous surface vehicle. In *2017 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 628–633, 2017. doi:10.1109/ROBIO.2017.8324487.

[5] Benjamin Pitzer, Sarah Osentoski, Graylin Jay, Christopher Crick, and Odest Chadwicke Jenkins. Pr2 remote lab: An environment for remote development and experimentation. In *2012 IEEE International Conference on Robotics and Automation*, pages 3200–3205, 2012. doi:10.1109/ICRA.2012.6224653.

[6] Mustafa Karaca and Ugur Yayan. Ros based visual programming tool for mobile robot education and applications. *arXiv preprint arXiv:2011.13706*, 2020.

[7] Jihoon Lee. Web applications for robots using rosbridge. *Brown University*, 2012.

[8] Yug Ajmera and Arshad Javed. Shared autonomy in web-based human robot interaction. In Kohei Arai, Supriya Kapoor, and Rahul Bhatia, editors, *Intelligent Systems and Applications*, pages 696–702, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-55190-2_55.

[9] E Saks. Javascript frameworks: Angular vs react vs vue. *Diplomová práce. Haaga-Helia University of Applied Sciences*, 2019.

[10] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, number 3.2, page 5. Kobe, Japan, 2009.

[11] Russell Toris, Julius Kammerl, David V. Lu, Jihoon Lee, Odest Chadwicke Jenkins, Sarah Osentoski, Mitchell Wills, and Sonia Chernova. Robot web tools: Efficient messaging for cloud robotics. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4530–4537, 2015. doi:10.1109/IROS.2015.7354021.

[12] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022. doi:10.1126/scirobotics.abm6074.

[13] Brandon Alexander, Kaijen Hsiao, Chad Jenkins, Bener Suay, and Russell Toris. Robot web tools [ros topics]. *IEEE Robotics & Automation Magazine*, 19(4):20–23, 2012. doi:10.1109/MRA.2012.2221235.

[14] Sarah Osentoski, Graylin Jay, Christopher Crick, Benjamin Pitzer, Charles DuHadway, and Odest Chadwicke Jenkins. Robots as web services: Reproducible experimentation and application development using rosjs. In *2011 IEEE International Conference on Robotics and Automation*, pages 6078–6083, 2011. doi:10.1109/ICRA.2011.5980464.

[15] Html: Hypertext markup language. https://developer.mozilla.org/en-US/docs/Web/HTML. Accessed: 2022-12-19.

[16] Dave Raggett, Jenny Lam, Ian Alexander, and Michael Kmiec. *Raggett on HTML 4*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[17] Css: Cascading style sheets. https://developer.mozilla.org/en-US/docs/Web/CSS. Accessed: 2022-12-19.

[18] Javascript. https://developer.mozilla.org/en-US/docs/Web/JavaScript. Accessed: 2022-12-19.

[19] Canvas api. https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API. Accessed: 2022-12-19.

[20] The websocket api(websockets). https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API. Accessed: 2022-12-19.

[21] Web workers api. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API. Accessed: 2022-12-19.

[22] Marco Krauweel and Sung-Shik T. Q. Jongmans. Simpler coordination of javascript web workers. In Jean-Marie Jacquet and Mieke Massink, editors, *Coordination Models and Languages*, pages 40–58, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-59746-1_3.

[23] Javascript library vs javascript framework - the differences. https://www.microverse.org/blog/javascript-library-vs-javascript-frameworks-the-differences#toc-what-is-the-difference-between-a-library-and-a-framework-in-react-js-. Accessed: 2022-12-20.

[24] React: A javascript library for building user interfaces. https://reactjs.org/. Accessed: 2022-12-23.

[25] Angular: What is angular? https://angular.io/guide/what-is-angular. Accessed: 2022-12-23.

[26] Vue.js: Introduction. https://vuejs.org/guide/introduction.html. Accessed: 2022-12-23.

[27] Angular vs react vs vue- javascript framework you need for your business. https://radixweb.com/blog/angular-vs-react-vs-vue. Accessed: 2022-12-23.

[28] Stefan Tilkov and Steve Vinoski. Node.js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, 2010. doi:10.1109/MIC.2010.145.

[29] Christian Peters. Building rich internet applications with node. js and express. js. *Rich Internet Applications w/HTML and Javascript*, page 15, 2017.

[30] M. Fowler and J. Lewis. Microservices. https://martinfowler.com/articles/microservices.html. Accessed: 2023-05-24.

[31] Rui Chen, Shanshan Li, and Zheng Li. From monolith to microservices: A dataflow-driven approach. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 466–475, 2017. doi:10.1109/APSEC.2017.53.

[32] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-67425-4_12.

[33] Scott M. Lewandowski. Frameworks for component-based client/server computing. *ACM Comput. Surv.*, 30(1):3–27, mar 1998. doi:10.1145/274440.274441.

[34] Qt is the complete software development framework. https://www.qt.io/product/framework. Accessed: 2022-12-19.

[35] Italo R. C. Barros, Luıs F. Costa, and Tiago P. Nascimento. Turtleui: A generic graphical user interface for robot control. In *2019 Latin American Robotics Symposium (LARS), 2019 Brazilian Symposium on Robotics (SBR) and 2019 Workshop on Robotics in Education (WRE)*, pages 174–179, 2019. doi:10.1109/LARS-SBR-WRE48964.2019.00038.

[36] James Trevelyan. Lessons learned from 10 years experience with remote laboratories. In R. Farana, P. Noskievic, W. Melecky, V. Kebo, Z. Weiss, I. Vondrak, R. Bris, J. Polak, V. Roubicek, and A. Dudacek, editors, *Proceedings of International Conference on Engineering Education Research*, volume 1, pages CD–ROM. iNEER, 2004. Lessons learned from 10 years experience with remote laboratories ; Conference date: 01-01-2004.

[37] Gustavo A. Casañ, Enric Cervera, Amine A. Moughlbay, Jaime Alemany, and Philippe Martinet. Ros-based online robot programming for remote education and training. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6101–6106, 2015. doi:10.1109/ICRA.2015.7140055.

[38] Dinodi Divyanjana Rajapaksha, Mohamed Nafeel Mohamed Nuhuman, Sewmini Dananji Gunawardhana, Atchuthan Sivalingam, Mohamed Nimran Mohamed Hassan, Samantha Rajapaksha, and Chandimal Jayawardena. Web based user-friendly graphical interface to control robots with ros environment. In *2021 6th International Conference on Information Technology Research (ICITR)*, pages 1–6, 2021. doi:10.1109/ICITR54349.2021.9657337.

[39] Artem Ivanov, Aufar Zakiev, Tatyana Tsoy, and Kuo-Hsien Hsia. Online monitoring and visualization with ros and reactjs. In *2021 International Siberian Conference on Control and Communications (SIBCON)*, pages 1–4, 2021. doi:10.1109/SIBCON50419.2021.9438890.

[40] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, may 2002. doi:10.1145/514183.514185.