Ros Web Application

Applicazioni e Servizi Web

Giulia Brugnatti - 000980270 {giulia.brugnatti@unibo.it}

August 26, 2021

Abstract

Recently there has been a growing interest in robotic systems, especially for what concern science exploration. Indeed, the number of projects, challenges concerning that scenario have risen; however, in general, it is not easy for students without an engineering background to get started to use the tools and the platforms for robotic management. This project aims to help students without that kind of scientific education to get familiar with robot environments.

Contents

1 Introduction **5**
1.1 Background 5
1.2 Introduction 5

2 Analysis **7**
2.1 Requirements 7
2.1.1 Functional Requirements 7
2.1.2 Non-Functional Requirements 8
2.2 Domain analysis 8

3 Design **9**
3.1 Methodology 9
3.1.1 User-centered design 9
3.1.2 Target users analysis 10
3.1.3 Scenarios 11
3.2 User interfaces 12
3.2.1 Mock-up 12
3.2.2 Final interfaces 16
3.3 Architecture 20
3.3.1 Web page application 20
3.3.2 Database 22
3.3.3 ROS 23

4 Develop **24**
4.1 Technologies 24
4.1.1 Overview 24
4.1.2 Web page and Data base 24
4.1.3 Robot application 28
4.1.4 Docker 28

5 Testing **29**
5.1 Overview 29
5.2 Methodology 29
5.2.1 Nielsen heuristics 29

5.2.2 Usability Testing 29

6 Deployment **31**
6.1 Background 31
6.2 Setup 31
6.3 Beyond Docker 32

7 Conclusion and future works **33**
7.1 Conclusion 33
7.2 Future works 33

A Appendix **34**
A.1 Alma-X and ERC 34
A.1.1 Alma-X 34
A.1.2 European Rover Challenge 35
A.2 ROS 36
A.2.1 Overview 36
A.2.2 History 36
A.2.3 Philosophy 37
A.2.4 Why ROS 38
A.2.5 Architecture 40
A.3 Catkin 52
A.3.1 Overview 52
A.4 Willow Garage 55
A.5 Docker 56

List of Figures

3.1 Diagram of the user centre design process 10
3.2 Mock-up homepage 13
3.3 Mock-up sign up 14
3.4 Mock-up simulation 15
3.5 Final home interface 16
3.6 Real registration interface 17
3.7 Real simulation interface 18
3.8 Real simulation charts interface 18
3.9 Real rover simulation's options 19
3.10 Architecture of the web application 21
3.11 Architecture of the database: user table 22
3.12 Architecture of the ROS simulated environment 23

4.1 MEVN stack 25
4.2 MEVN stack structure 25
4.3 Vue Angular React icons 26

A.1 Alma-X logo 34
A.2 ERC logo 35
A.3 ROS history 37
A.4 Table exposing key differences between ROS and an operating system 38
A.5 ROS file system's graph 40
A.6 ROS package's common files and directories 41
A.7 ROS builtin types 42
A.8 ROS graph level 43
A.9 ROS computational communication's graph 44
A.10 ROS: master-client registration's graph 45
A.11 ROS publish-subscribe's graph 45
A.12 ROS client-server interaction's graph focused on topics and services 47
A.13 ROS client-server synchronous interaction's diagram 49
A.14 ROS Community level's basic components 50
A.15 ROS distributions 51
A.16 ROS Catkin: main folder description 53
A.17 ROS Catkin: typical structure of catkin Source Space 53

A.18 Publish subscriber pattern interaction's graph 55
A.19 Willow Garage logo 56

Chapter 1

Introduction

This chapter provides a brief introduction about the project besides some preliminary requirements readers should have.

1.1 Background

The heart of this project is the connection between the *single-page application*(SPA) and the robotic environment. Therefore, since the purpose of this paper is not to describe the robotic platform, it is not provided with a ROS-user guide. However, the appendix section A.2 contains some relevant features of the platform useful to understand how the web page can connect to the ROS middleware. Furthermore, since the idea of the project was born within the Alma-X group to help specifically target users, having some preliminary knowledge [1] about the group and the competition in which it is involved is helpful to clarify the whole project purpose.

1.2 Introduction

The primary goal of this project is to introduce non-expert users to the rover simulation world; without needed them to learn how to use a professional platform for robot simulation. Therefore, the web application must create a level of abstraction between the students and the robotic platform to simplify their learning process, making them focus only on the relevant feature of the simulation. Learning to use a robotic platform can be helpful, but challenging sometimes, and not everybody wishes to enhance ones' background with that kind of knowledge. Simulation can concern a vast set of domains and, to not be able to install and load it should not be an obstacle to explore that world. This project aims to overcome the initial difficulties users can have when approaching

[1] In the appendix sections A.1.1 and A.1.2 can be found some information about the group and the competition

that kind of environment by providing them with an intuitive web interface to navigate unexplored realities.

Chapter 2

Analysis

This chapter is about an analysis of the functional and non-functional requirements of the web application, the study of its domain, and the development of its model.

2.1 Requirements

In this section will be discussed firstly the functional requirements the application must meet, followed by the non-functional requirements.

2.1.1 Functional Requirements

The proposed application must provide pages for:

- authentication and registration: pages where users can en-register themselves and authenticate to access to the simulation;
- a homepage: a page where the user arrives when the application starts and from which its navigation of the page begins;
- a simulation page: a page that handles the management with rover application. The page must provide the following elements:
  - a section for streaming video: from which users can see what happens in the simulation environment;
  - a module to control the navigation in the environment word;
  - an element to establish and tear down the connection with the robotic platform and change its parameters if needed;
  - a module to show some relevant data from the simulation using some diagrams.

2.1.2 Non-Functional Requirements

The most indicative non-functional requirements the project needs to provide are usability and safety. Indeed, as users without a STEM[1] background use the application, it should be designed to be simple to navigate and check the rover data. Furthermore, since non-expert users are error-prone, the app should provide a way to let them interact with the rover environment without giving them the possibility to produce potentially harmful situations[2].

2.2 Domain analysis

In this section it is described the model of the *application domain*, illustrating the entities involved and their relationships.

The web application can connect to the robotic platform where the robot's simulation is running.

When the page establishes the connection, users can send commands to the robot and, they can also check some of its data on the other.

To avoid some security issues the web page asks users to be registered before loading the robot's controlling page.

As the main challenging aspects of the development of the platform mainly concern its connection with the robotic middleware, especially the real-time features, the authentication ones have to take second place.

For this reason, this first version of the software will not include a page for page administrators, assuming that all the users after the registration can access the robot controlling page.

[1] STEM: Science, Technology, Engineering, and Math
[2] Container usage + authentication

Chapter 3

Design

This chapter describes the strategies and methodologies developed to meet the requirements identified in the analysis phase

3.1 Methodology

As a primary methodology for the design process was chosen the *User-centered design*.

3.1.1 User-centered design

This UCD[1] approach was chosen because considered helpful in understanding users' needs and preferences regarding features of a product, task, goals, user flows, etc and in this way improve the user experience.

[1]The term User-centered design (UCD) originated in Donald Norman's research laboratory at the University of California San Diego (UCSD) in the 1980s, it is used to describe an iterative design process in which designers focus on the users and their needs in each phase of the design process. In UCD, design teams involve users throughout the design process via a variety of research and design techniques, to create highly usable and accessible products for them[10][18].

Figure 3.1: Diagram of the user centre design process

3.1.2 Target users analysis

3.1.2.1 Personas

The platform is thought to be used by university students with some background in the ERC competition. Therefore, even if it is not mandatory to have a complete robotic background, it is helpful to know the environment in which the rover operates. In this case, the platform is designed to be used by Alma-X team members[2].

Alexandra
Alexandra is a 23 years old student who is about to graduate in Geology. She is passionate about astronomy since she was a child. Alexandra has no interest in computer science or, robotics but she joined the Alma-X rover team about two years ago because of her interest in space. She is part of the Science Division since then however, she has limited knowledge about ROS or the physical and software architecture of the rover.

Lorenzo
Lorenzo is a *PhD* student at the *DEI*[3] with a strong robotic background and especially of the ROS platform. He joined the team in August to help undergraduate students better understand robotic words. Lorenzo is not part of any particular division he works transversely with all divisions giving suggestions when he is asked for.

[2]For that reason the web application uses the official ERC simulation as a robotic environment to connect to.
[3]Department of Electronic and Engineering

Marco
Marco is a chemical engineer student passionate about space and STEM subjects in general. He is an eclectic person, and he is not afraid of trying new experiences; because of that, he joined the team at the beginning of the year. Marco is part of the Science division, he has low awareness of Linux, and he cannot configure a ROS environment correctly.

3.1.3 Scenarios

This section describes the main scenarios of the web application using the personas of the previous paragraph.

Alexandra
Story about Alexandra: *Alexandra has just finished developing the new navigation plan that the rover should follow during the Navigation task. She wants to check if the path she designed along with the Science division is feasible with the rover structure. Indeed, she starts the Docker container, and she loads the Ros Web App. After having logged in with her credentials, she reaches the rover simulation page. Then, she starts to move around the Marsyard using the route decided. At a certain point, she stuck the rover on a massive crater. So, Alexandra realizes that their calculus was inexact. The rover seems not to be able to mount from that crater once he has descending inside to explore the area. As Alexandra has no clue about how to unstuck the rover, she shut everything down and, text her teammates for help.* Alexandra is not fond of technologies and, in particular, she does not like to change their plan to meet the physical constraints of the rover; for this reason, when something does not work, she only closes the web page. Therefore the only operations the does are:

- load the simulation, login and connect to the ROS environment;
- use the buttons to drives the rover around to test her traversal plan;

Lorenzo
Story about Lorenzo *The navigation division has designed a new planner, and the members would like Lorenzo to test it to be sure their algorithms work fine. However, Alexandra asks him to help her understand if it is not feasible for the rover to exit from the crater or if, on the contrary, it is simply her that is not a great pilot. Since Lorenzo has a small-time, he decides to do a video call with Alexandra, testing both the issues at the same time. To do so, Lorenzo loads the web page and reach the simulation page, and drives the rover in the position Alexandra indicates there, he tries to climb the rock from different part of the crater analyzing the odometry data shown by the chart below the rover commands. From that data, he deduces that the Leo-rover has not powered required to exit from the crater. Then he says goodbye to Alexandra and, he starts to test the new Navigation planner. Therefore, he opens the advanced settings ad change the address port of simulation to connect to the new ROS*

environment. While he has started to shut down the previous docker image, he tests how the new planner works. When he finishes he updates the leader of the Navigation Division about its results and shut down its pc. Lorenzo is an expert user that does not like so match to use the web app because of its latency however he knows that it is useful and easy to use for non-engineering rather than ROS. Since Lorenzo is an expert he can exploit all the operations the platform provides. Therefore, besides the basic operations, Alexandra can do; he can also check the address of the simulation and change it if it is necessary.

Marco
Story about Marco: *Marco is in the same division as Alexandra however, he has quite a different approach to the web app, indeed he does not get mad when something does not work, as he does not perceive errors as a failure but as great opportunities to improve oneself. For this reason, when Alexandra text him that she is stuck he immediately turns his computer on trying to check out what the problem was. Marco is a robotic self-taught, but he is not afraid to experiment around, therefore he tries to analyze the odometry data when the rover is trying to climb the crater. After some time, he arrives at the same conclusion as Lorenzo: it was not feasible for that rover to exit from that crater without any help.* Marco is not an expert user as Lorenzo is, but he tries very hard to do his best to exploit all the web application provides him, because of that besides the simple operation also performed by Alexandra, he also uses the data chart to analyze how the rover behaves in the Marsyard environment.

3.2 User interfaces

This section presents some user interfaces developed in reason of personas and use cases analyzed in the previous paragraphs. The user interfaces are designed following a mockup approach. The mockups are created using a popular tool named: *Moqups*. The tool was chosen because it is fully online and free.

The project for its intrinsic nature[4] is thought to be used on Desktop devices, because of that, it was not developed following the *mobile-first philosophy*. Nevertheless, the interfaces are responsive therefore, they can also work on mobile devices.

3.2.1 Mock-up

Before beginning to develop the final user interface, the mock-up of the main sections was designed and evaluated to be sure of the usability of the application.

Home page
The home page is the first element of the application seen by the users; the main idea for this first item was to show something students were familiar with. For

[4]To conduct analysis and experiments on mobile devices would be challenging because of the reduced dimensions of screens.

this reason, the page opens with a popular image from the ERC. Next, to be user-friendly it was thought to display the login form as well as the registration button on the navigation bar to reduce the number of operations users have to do to reach the simulation page. Finally, on the bottom of the page, it is present also a footer section that shows the icons of some quite popular platform among users that allow reaching the main sites of the Alma-X group.

Figure 3.2: Mock-up homepage

Registration page
The registration form follow a standard template, the logo of the group is shown on the top and next it is asked the user to fill a set of a text field to create an

account.

Figure 3.3: Mock-up sign up

Simulation page
The simulation page is the more complex element of the application since it must condense all the rover-related elements. Because of this intrinsic complexity, it was thought that it would be better to design its interface following the *KISS* principle and the *Less is more* approach. For this reason, the simulation page is designed not to have decorative elements but only useful ones.

Figure 3.4: Mock-up simulation

As can be seen from the figure 3.4 this page is made up of many elements: on the top, there is the navigation bar, next to a section for the rover connection primary operations, followed by the rover video, commands and odometry data charts. Given that no action can be performed before the connection with ROS is established, the buttons that allow the connection with the robotic platform and the setup of the address used to communicate are shown on the top of the page. Next, it is shown the heart of the simulation page: the video detected by the rover camera and the buttons responsible for the rover motions around the Marsyard. Finally, below all the primary elements are shown also two charts

that plot the odometry data: on the left related to the position and on the right those related to the orientation.

3.2.2 Final interfaces

As will be seen, the interfaces were refined following the users' suggestions. The main changes performed can be summarized as follow.

Home page
On the page had been added a title, followed by a small summary of the purpose of the application.

Next, it has been added a quote below the slider to make the application more captivating.

Figure 3.5: Final home interface

The **navigation-bar** only changed in the colour of the button; that move from a lighter colour, to a darker one to improve the readability.

Furthermore, this component has been fixed at the top of each page of the application to simplify the navigation of the page. In this way, the login button is substituted by the logout button when the user is logged in, but the user can always find it in the same position and can come back to the home page just by clicking on the Alma-X logo.

Registration page
On the registration page, the main changes concern the fields the user must

fill; users affirm to prefer to write their names and surnames in different fields because this is how they usually used to it. Furthermore, the application in its final version asks the user to confirm the user password because it has emerged that in this way the user is most likely to remember its password.

Figure 3.6: Real registration interface

Simulation page

On the simulation page, the primary changed concerns the charts and the setting button.

Figure 3.7: Real simulation interface

The appearance of the charts in their colours and type change to better meet the users' requirements indeed, those latter consider the proposed charts too colourful and the charts titles not readable enough. The button for setting

Figure 3.8: Real simulation charts interface

the rover network option replaced its icon with a title, to be much notable by users. Furthermore, instead, to display a new page to set up the rover options concerning the page connection with ROS middleware, users affirm to prefer a sidebar menu.

Figure 3.9: Real rover simulation's options

3.3 Architecture

The project system is composed of three main components: the web application, the database and robot middleware. As advanced, it is not the purpose of the project to describe the details of the robotic platform, or the rover simulation, because of that in this section will focus on web page components giving only a brief overview of the database and the robotic application.

3.3.1 Web page application

The web application is developed as a single page application with only the main process, the `index.html`, responsible for handling all the features of the project and where all the components are declared. The components can be divided into two main categories:

- basic components: they are components that do not use other components to provide their functionalities;
- composed components: they are components made up of one or more components;

3.3.1.1 Simple components

Footer : it is the component that provides the footer section for the main page. That latter shows the main sites of the organization.

Slider : this component provides a slider for the main page of the web page showing users a set of images of the ERC competition.

Navigation Bar : it is the navigation bar users can use to move around the site whether they are logged, or not. If users are not logged then they can do it, or they can also register their-selves[5]. When the user is logged, the component allows the user to access the simulation page, or to perform the logout operation. This component is always shown, no matter which operation the user is doing.

Rover charts : it is the component responsible for displaying the charts about the odometry values of the rover. It also allows the user to download the data in a different format.

Rover commands : it provides the buttons that allow students to control the rover motion within the simulation environment.

Rover video : it shows the video detected by the rover camera, allowing users to see Marsyard.

[5] To do this operation the navigation bar will re-direct them to the Registration component.

Rover modal : it allows to change the address at which the robotic platform and the web page are communicating.

3.3.1.2 Compound components

Home page : it is the main page of the projects, it contains both the slider and the footer components.

Rover Simulation : it the heart of the project it contains all components for the rover management[6]. This component is the one in which the connection with the ROS platform is established or disabled; it has a set of listeners to detect the rover data and of function to translate users commands in ROS topics that must be published over the ROS bridge.

Rover Settings : it is a sidebar menu. It can be used by expert users to check the address of the connection between the robotic platform and the web page. It also contains the rover modal component.

Figure 3.10: Architecture of the web application

[6]It also contains the rover setting component

3.3.2 Database

The database is quite simple; it is made up of only one schema the users one. The user schema has the following fields:

User	
PK	**ObjectId** name surname email password salt

Figure 3.11: Architecture of the database: user table

- ObjectId;
- name: user name;
- surname: user surname;
- email: field that is used for login operation and this reason must be unique;
- password: hash of the password chosen by the users when registered;
- salt: salt value used to hash the password.

Since odometry data are only used[7] in real-time analysis it was chosen to save that in a database would have been pointless.

[7] Alma-X group does not analyse those data offline.

3.3.3 ROS

As can be seen from the following graph[8], the ROS environment used for simulation is quite complex. Because of that, it will not be analyzed, as the purpose of the project is exactly to provide a way for users to abstract from this complexity and focus on what matters for their works.

Figure 3.12: Architecture of the ROS simulated environment

[8]The graph was created using a ROS tool named `rqt_graph`.

Chapter 4

Develop

This section provides an overview of the main technologies used in the project.

4.1 Technologies

4.1.1 Overview

The project involved three main components, the web application and the database used to handle user registration and the robotic platform.

Given that, it is the purpose of its project is to abstract to robotics' details managements, it will be not given an analysis about how to handle the robotic simulation (i.e. its detailed architecture, how robot's components can communicate with each other and among the system, etc.). This section describes in detail only the web application architecture.

4.1.2 Web page and Data base

4.1.2.1 MEVN

The MEVN[1] stack was chosen as a framework to develop the web page application. MEVN stack is the open-source JavaScript software stack that has emerged as a new and evolving way to build powerful and dynamic web applications. Its software components can be used to effectively design *frontend* and *backend* development and improve the functionality of your website or app. It is composed of four main elements that will be described in the next paragraphs.

[1] The acronym MEVN stands for MongoDB, Express.js, VueJS, Node.js

Figure 4.1: MEVN stack

MongoDB : a document-oriented, No-SQL database used to store the application data.

ExpressJS : a framework layered on top of NodeJS, used to build the backend of a site using NodeJS functions and structures. Since NodeJS was primarily developed to run JavaScript on a machine instead of making websites, ExpressJS was created for the latter purpose.

Vue JS : VueJS is referred to as a client-side framework and is especially used in front-end web development. It has two-way data binding that allows seamless frontend development along with MVC capability and interactive server-side applications.

NodeJS : The JavaScript runtime environment. It is used to run JavaScript on a machine rather than in a browser.

Since Vue JS does not use the usual MVC[2] architectural pattern, but, like many other similar frameworks, one of its declination the: $MVVM$[3] pattern the web application is developed use that latter pattern.

Figure 4.2: MEVN stack structure

[2]MVC: Model View Controller
[3]MVVM: Model View ViewModel

Why Vue JS?
For the client side of the application *Vue.js*[4] was elected as target platform.

That latter was chosen after analysis with two other relevant frameworks: *React*[5] and *Angular*[6]. The analysis took into account several parameters in order to understand which platform could fit better for the project. The main

Figure 4.3: Vue Angular React icons

characteristics analyzed are the size and load times of the frameworks, their integration with other libraries and their learning curve. Indeed, even if at first sight Vue, Angular, and React, can appear quite similar, they have some relevant differences at an in-depth examination. Certainly, despite Angular is the most mature platform, due to the complexity of its syntax, it has the steepest learning curve, besides it also has the largest, therefore the heaviest to load, framework.

On the other hand, React is easier to learn, but it has a poor[7] framework that often required the user to use third-party libraries for advanced features. Finally, although Vue is the newest framework, it is the easiest to learn and, it offers higher customizability. *However, simplicity and flexibility of Vue is a double-edged sword — it allows poor code, making it difficult to debug and test*[4].

Undoubtedly, each of these libraries has its benefits and drawbacks, however, for project development, Vue has been preferred because of its simplicity and flexibility.

4.1.2.2 Besides MEVN

Besides the MEVN stack, other frameworks and libraries were used to handle the tasks the web page must perform. These will be described in the next paragraphs.

[4]https://vuejs.org/
[5]https://it.reactjs.org/
[6]https://angular.io/
[7]React's framework is the less complete among the three platforms

Database management :
Mongoose: is a MongoDB object modeling tool for node.js designed to work in an asynchronous environment. It includes built-in type casting, validation, query building, business logic hooks and more.[14].

Server request management
Axios: is a promise-based HTTP Client for node.js and the browser. It is isomorphic [8]. On the server-side it uses the native node.js http module, while on the client (browser) it uses `XMLHttpRequests`[2].

Design management
Bootstrap: is a free and open-source CSS framework directed at responsive, mobile-first front-end web development. It contains CSS- and (optionally) JavaScript-based design templates for typography, forms, buttons, navigation, and other interface components[34].

Authentication management

- **Bcrypt**: is a password-hashing function designed by Niels Provos and David Mazières. Besides incorporating a salt to protect against rainbow table attacks, bcrypt is an adaptive function: over time, the iteration count can be increased to make it slower, so it remains resistant to brute-force search attacks even with increasing computation power[33].
- **JSON Web Token (JWT)**: is an open standard that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed[12].

ROS communication

- **Roslib**: is the base dependency of all ROS Client Libraries and tools. It contains common tools like the generators for Messages and Services as well as common message definitions like Header and Log. It also contains the common path-bootstrapping code for ROS Python nodes and tools[30]. This library provides the API responsible for the connection between the rover middleware and the web page; the simulation component used the API of the roslib library to publish and listen for topics from the ROS environment. The ROS environment is running a similar component called `rosbridge_server`[9] which publish and subscribe for the data the web application required.

[8] can run in the browser and nodejs with the same codebase

[9] `rosbridge_server`: Rosbridge server is part of the rosbridge_suite of packages, providing a WebsSocket transport layer. A WebsSocket is a low-latency, bidirectional communication layer between clients (web browsers) and servers. By providing a WebsSocket connection, rosbridge server allows webpages to talk ROS using the rosbridge protocol.

Rosbridge server creates a WebSocket connection and passes any JSON messages from the WebSocket to rosbridge_library, so rosbridge library can convert the JSON strings into ROS

- **Mjpegcanvasjs**:is a JavaScript tool that allows for an easy way to display, manage, and modify ROS image streams from `web_video_server`[10]. Both single-stream and multi-stream support are available. That is, the canvas can either hold a single stream or allow a user to select and change the stream currently being rendered to the canvas.[23]. This component is used for streaming video management; it provides some API used by the web application to listen to the video from the robotic platform. Also in this case the tool works because the ROS environment is running the corresponding node, `web_video_server` that re-publish the video topic over the network.

Data visualizazion
Apexcharts: is a modern JavaScript charting library to build interactive charts and visualizations with simple API[16]. These are the elements used to plot the odometry data.

4.1.3 Robot application

4.1.3.1 ROS

A section will not be suitable to describe the ROS environment, and as advanced many times before in this paper this is not the purpose of the project. Nevertheless, if one is interested in having some information can find it at the end of the document in the Appendix sectionA.2.

4.1.4 Docker

Since one of the most important goals of the project is the user usability of the platform it has been chosen to deliver the application using Docker A.5.

Logically the application delivered is composed of three parts: the web page, the database and the ROS environment, however, to improve the modularization of the simulation environment it was chosen to decompose it in three different containers: one for the `rosbridge_server`, one for the `web_video_server` and one for the rover simulation[11]. In this way, in case one want to change the robot, then one just must provide a new Dockerfile for that and the whole system will continue to work.

calls. The reverse also happens, with rosbridge library converting any ROS responses into JSON, then passing it to rosbridge server to send over the WebSocket connection[29].

[10] `web_video_server`: the web_video_server opens a local port and waits for incoming HTTP requests. As soon as a video stream of a ROS image topic is requested via HTTP, it subscribes to the corresponding topic and creates an instance of the video encoder. The encoded raw video packets are served to the client. Parameters can be specified by adding additional them to the query string. To connect to the node with your browser, use the following URLs[31]

[11] For the rover simulation it was chosen to use the official simulation by the ERC.

Chapter 5

Testing

5.1 Overview

Since the project was thought to be used by the Alma-X team members they were asked to participate in the testing phase. In particular, they were divided into two groups; one was involved from the beginning in the design process, while, the other was asked to test the application only when finished.

In fact, even if it was helpful to have feedback from the final users of the application, it was necessary to avoid bias, therefore the group was split up.

The testers were not only the members of the Science Division but, also some robotic-expert from other divisions who helped in the choice of the topic to display, as well as the kind of graph that better represented the data.

The help and the continuous feedback and suggestions from the whole Alma-X group were essential to discover how the application could better meet their requirements.

5.2 Methodology

5.2.1 Nielsen heuristics

Nielsen heuristics[1] were used before the usability test with some expert users, to deliver a pre-corrected interface to the final users.

5.2.2 Usability Testing

Usability testing is evaluating a product or service by testing it with representative users.

[1] Heuristic evaluation is a usability engineering method for finding the usability problems in a user interface design so that they can be attended to as part of an iterative design process. Heuristic evaluation involves having a small set of evaluators examine the interface and judge its compliance with recognized usability principles (the "heuristics")[15].

These tests were mainly held with final target users who were asked to accomplish some specific task (i.e. to perform sign-up and login operations, to move to connect to the ROS platform etc.). After they completed their task they were asked to give feedback about the usability of the application as well as their user experience.

Chapter 6

Deployment

6.1 Background

The project can work on almost all operating systems, but it is necessary to install both `ROS` and `NodeJS`, to avoid this operation, the application had been deployed as a docker image; however, that latter only works on a Linux operating system.

6.2 Setup

The following steps illustrate how to load the simulation in a *Linux* environment; it will not be described how to set up the project in any other operating system[1].

- Download the source code from the *GitHub* repository: `https://github.com/GiuliaB97/ros-web-app`;
- From the `aws-project` folder launch the command:

```
xhost +local:root
docker-compose up
```

- Connect to web page:

```
http://localhost:7000/
```

[1]This project aims to simplify the setup operation to allow non-expert user to work with a ROS simulation; for this reason since Windows or macOS image for the project are not provided discuss how to set up on those environments will be pointless

6.3 Beyond Docker

Given that running the project only on Linux operating system would have reduced the usability of the application, it was implemented another possibility. Thanks to the *Data Analysis Division* of the Alma-X group, who provided an exposed server, the project is also remotely available.

Indeed, by changing some minor configuration in the network of the Docker container, as well as, the addresses used within the project, now users can use the project from all operating systems without needing to download and build the docker simulation. The application is therefore available at the following link: `https://webleo.fo.cac-lab.com` However, the access to the server is not provided by a professional network, but a home one it could happen to lose the availability of the application.

Because of that, it is on the user the choice to configure the project locally or access it remotely.

Chapter 7

Conclusion and future works

7.1 Conclusion

This project gave me the possible experience for the first time how to deliver a web application using the *MEVN* stack and to build a real *SPA*.

However, what I like most about the project was the possibility of its integration with the ROS platform. Indeed, even if it was almost the most challenging part of the application, it is the one that made me much proud of my work.

The project is far from being perfect the streaming video latency is still a major problem, however, I am satisfied to succeed in finishing the web application managing to integrate all the topics that I wanted to and having so positive feedback from the Alma-X group.

7.2 Future works

Future works of the application could concern its extension on one hand and, improving its performance on the other. In the first case, the page of the rover simulation could show more topic data and give the user the possibility to choose how to control the rover (i.e. keyboard, joystick, or buttons), in second it could fix that latency that affects the publication of the topic from and to the robotic platform; in this way, the streaming video performances' could have a significant improvement.

Appendix A

Appendix

A.1 Alma-X and ERC

A.1.1 Alma-X

Alma-X is an association made up of students of the *Alma Mater Studiorum, University of Bologna* that attends to the international competition *European Rover Challenge* (ERC A.1.2). The team is a heterogeneous group of 30 students

Figure A.1: Alma-X logo

from the campus of Bologna, Cesena and Forlì, where the team was born and the headquarter is therefore located. Alma-X members are divided into five main groups: the *navigation, robotic, vehicle dynamics, data analysis* and *science* division.

Navigation division works on the algorithms that drive the rover around, with the primary goal of refining the autonomous navigation routines. It is responsible of performing the *navigation task* during the competition.

Robotics division is dedicated to the control of the robotic manipulator and its handling of the maintenance task. Also, it is concurrently pursuing the project of the design of a new prototype of an arm we will include in our rover, based on the outcomes of the previous years, where we studied deeply filament

deposition printing with the association partners;

Vehicle dynamics division, works on the rover physical structure developing the *CAD* model and implementing the control of the adaptive suspensions.

Data analysis division manages the general controller and must keep all the information flow between all the other subsystems working. The group also handles the energetic ecosystem (battery, *BMS*, switches) and thermal simulations.

Science division is made up of a group of geology students that works on gathering information on the Mars surface and plan our mission for the science task, alongside the Navigation group.

A.1.2 European Rover Challenge

The European Rover Challenge (ERC) is an annual international Martian robots competition attended by teams from around the world. The competition has been organized since 2014 in Poland [8].

ERC is the biggest robotics and space event in Europe addressed to the world of science and business, the new technologies sector as well as the general public. Guided by the slogan "we inspire a new generation", ERC brought the cosmos closer to the general public, making people aware of the growing role of modern technologies in our lives since its establishment. Furthermore, the event is also a meeting place for representatives of the European world of science and business interested in the use of space and robotic technologies[7].

Figure A.2: ERC logo

The challenge is a prestigious space & robotics event combining international competitions of mobile robots with scientific and technological shows, however, the core of ERC is an international Martian robot challenge attended by students, as well as graduates, and researchers from universities around the world. The teams consist of a heterogeneous group of specialists in mechatronics, automation, robotics, etc., who work hard on a fully functional next-generation Mars robot for 12 months.

In 2020, the ERC was the first Robot competition to be carried out in the new remote formula, in which competitors from several continents remotely controlled a robot[1] located on the *Marsyard* on the campus of the *Kielce University of Technology* in Poland.

[1] The robot chosen was the Leo Rover.

Since 2021, the competition takes place both in the *remote* and *on-site formula*[8][7].

A.2 ROS

A.2.1 Overview

The Robot Operating System, better known as ROS, is a collection of libraries, tools, and conventions whose main aim is to simplify the creation and management of robust robot software for a wide variety of robot platforms, sensors, and effectors.

A.2.2 History

ROS development dated back to 2007, when in the Stanford Artificial Intelligence Laboratory, Morgan Quigley[3], Eric Berger and Andre Y. Ng[32] creates a library to support the STAIR[2] project and the PR[3].

That library was named Switchyard, and its primary purpose was to "supports parallel processing through message passing along a user-defined, task-specific graph of connections between software modules"[20]. Switchyard was such a puissant idea that in 2009 the Stanford team, with the collaboration of the Willow Garage[4], presented *ROS: An Open-Source Robot Operating System* at the *IEEE International Conference on Robotics and Automation* (ICRA) in Japan.

Next, in 2010 the first stable implementation of the framework was released, and since then ROS ecosystem has not stopped to evolve becoming more elaborated and wider day by day.

[2]STAIR: STanford Artificial Intelligence Robot
[3]PR: Personal Robots
[4]Further information about Willow Garage research lab are in the Appendix sectionA.4

Figure A.3: ROS history

A.2.3 Philosophy

ROS initial goals can be sum up in the following points:

- **Peer-to-peer**: ROS systems are made up of several processes, possibly on different hosts, linked at runtime in a peer-to-peer topology. This latter arrangement needs some kind of lookup mechanism to allow processes to find each other at runtime[5].
- **Tools-based**: ROS has a microkernel design, where numbers of small tools are used to build and execute the different ROS elements[6]
- **Multi-lingual**: ROS is designed to be language-neutral; supporting several languages like C++, Python, Octave, Java, Lua, and LISP, with other language ports in various states of completion.
- **Thin**: ROS build system performs modular builds inside the source code tree; placing virtually all complexity in libraries, and only creating small executables which expose library functionality to ROS, allows for easier code extraction, reuse, and testing beyond its original intent.
- **Free and Open-Source**: ROS source code is publicly available[21].

[5]ROS provide a specific tool for this purpose: the master, which will be analyzed in the next chapter.

[6]ROS developers believed that the loss in efficiency caused by the choice of not constructing a monolithic development and runtime environment is more than offset by the gains instability and complexity management.

A.2.3.1 What ROS is not

ROS is an open-source framework that provides useful tools and libraries to help software developers in the creation of robotics application, and it should not be confused with:

- a **programming language**; indeed ROS is not a programming language ,but it supports different languages in C++, Python, and Lisp.
- an **operating system**, as exposed in the former section, even though it shares some features with operating systems ROS is only a meta-operating system. In the next tableA.4 the major differences between ROS and a common operating system are shown.
- an **integrated development environment**; although ROS provides tools for debugging, implementation and simulation is not an IDE.
- a **library**; ROS does include client libraries, but also a central server, a build system, a set of command-line tools, and graphical tools[17].

What Makes the Difference...?

Conventional OS	ROS
Explicitly a general purpose OS	Exclusive for Robotic Platform
Ortho-Operational	Meta-Operational
Native Language Programming	Language-independent architecture
Sequential Architecture	Asynchronous Distributed architecture
Programming IDE	Software Frameworks
Propriety/Open-Source	Open-source under BSD license
Heavily coded frameworks	ROS frameworks are very light
Programs	Nodes
Communication	Messages
Kernel is Included	Kernelless

Figure A.4: Table exposing key differences between ROS and an operating system

A.2.4 Why ROS

The most relevant features that make ROS the de facto standard in robotics software[3] outlined as follows:

- **Distributed computation**: many modern robot systems rely on software that spans several processes and runs across several different comput-

ers. ROS provides two considerably basic mechanisms for communication between processes that may or may not live on the same computer.

- **Software reuse**: the rapid progress of robotics research has resulted in a growing collection of efficient algorithms for common tasks (i.e navigation, mapping, etc.), however, their existence is only useful if there is a way to apply them in a new context without needing to re-implement them. In this respect, ROS can help to prevent to *reinventing the wheel* offering a set of software for operating robots that can be used as *off the shelf* programs or be extended.

- **High-end capabilities** ROS has ready for use highly configurable capabilities, for instance, *SLAM*[7] and *AMCL*[8] packages that can directly be utilize in robots software.

- **Tons of tools** ROS is comes with several tools, as `rqt_gui`, *RViz* and *Gazebo*, for debugging, visualizing, and performing simulation.

- **Support high-end sensors and actuators** ROS has a multitude of device drivers and interface packages of various sensors and actuators in robotics (i.e *Velodyne-LIDAR*, *Laser scanners*, etc.).

- **Inter-platform operability** The ROS message-passing middleware allows communicating among different nodes that can be programmed in any language that has ROS client libraries.

- **Modularity** One of the issues that can occur in most standalone robotic applications is, if any of the threads of the main code crash, the entire robot application can stop. ROS overcome this limitation on one hand by making the programmer writing different nodes for each process in this way if one node crashes, the system can still work, and on the other by providing robust methods to resume operation even if any sensors or motors are dead.

- **Concurrent resource handling** ROS simplifies the operation of handling a hardware resource by more than two processes by making it possible to access the devices using ROS topics from the ROS drivers. In this way multiple ROS nodes can subscribe to the same message from a specific driver and each node can perform different functionalities. This feature can reduce the complexity in computation as well as increase the debug-ability of the whole system.

- **Active community** One of the most relevant factors, that must be considered while choosing to use a library or software framework from an open-source community, is if it is supported by a developer community. In ROS, the support community is highly active, besides there is a web portal to handle the support queries from the users[9][17][11].

[7] SLAM: Simultaneous Localization and Mapping
[8] AMCL: Adaptive Monte Carlo Localization
[9] ROS Answers site: (*http://answers.ros.org*)

A.2.5 Architecture

A.2.5.1 Overview

The structure of ROS is made up of three main level concepts:

- the **File system level**
- the **Computational Graph level**
- the **Community level**

Those levels will be discussed with further details in the following sections, along with the two types of naming structures, **Package Resource Names** and **Graph Resource Names**, that ROS defined.

A.2.5.2 File system

Like operating system, ROS file system organizes its files and folder into a hierarchical structure on disc as shown by the following graph:

Figure A.5: ROS file system's graph

In the following paragraphs, will be given a brief delineation of each element of the former diagram[11].

Meta packages
Meta packages are specialized structures whose aim is to depict a group of packages that have a special purpose.

Unlikely, other kind of packages they contain only one file: the `package.xml` file. In the `package.xml` file, the meta package contains an export tag, as follows:

```
<export>
    <metapackage/>
</export>
```

Also, in meta packages, there are no `<buildtool_depend>` dependencies for Catkin[10], there are only `<run_depend>` dependencies, which are the packages grouped in the meta package.[11]

Meta packages manifest : it is an `.xml` file which contains the main information about the package (author, license, dependencies, etc.) besides including packages inside as runtime dependencies[11].

Packages
ROS packages are the most basic unit of the ROS software; they contain ROS runtime processes (nodes), libraries, configuration files, and so forth, which are organized together as a single unit. Packages are the atomic build item and release item in the ROS software.[17]

Directory	Explanation
include/	C++ include headers
src/	Source files
msg/	Folder containing Message (msg) types
srv/	Folder containing Service (srv) types
launch/	Folder containing launch files
package.xml	The package manifest
CMakeLists.txt	CMake build file

Figure A.6: ROS package's common files and directories

Package manifest : it is similar to the meta-package manifest; however, unlike it, the package manifest can not include packages inside it as runtime dependencies[11].

Messages
ROS Processes communicate with each other by publishing messages to topics. A message is a basic data structure, comprising typed fields which support

[10] A description of Catkin build system can be found in chapter A.3 of the Appendix section

standard primitive types as well as arrays of primitive types and constants. The following table (A.7) illustrates some of the built-in field types that can be used in a message:

Primitive type	Serialization	C++	Python
bool(1)	unsigned 8-bit int	uint8_t(2)	bool
int8	signed 8-bit int	int8_t	int
uint8	unsigned 8-bit int	uint8_t	int (3)
int16	signed 16-bit int	int16_t	int
uint16	unsigned 16-bit int	uint16_t	int
int32	signed 32-bit int	int32_t	int
uint32	unsigned 32-bit int	uint32_t	int
int64	signed 64-bit int	int64_t	long
uint64	unsigned 64-bit int	uint64_t	long
float32	32-bit IEEE float	float	float
float64	64-bit IEEE float	double	float
string	ascii string(4)	std::string	string
time	secs/nsecs unsigned 32-bit ints	ros::Time	rospy.Time
duration	secs/nsecs signed 32-bit ints	ros::Duration	rospy.Duration

Figure A.7: ROS builtin types

Furthermore messages can include arbitrarily nested structures and arrays[24] as well as being created ex-novo by users. The extension of the message file is `.msg`, besides they can be defined as follow:

```
int32 number
string name
float32 speed
```

The message definition can consist of two types: fields and constants. The field is split into field types and field names; field types are the data type of the transmitting message, and field name is the name of it, whereas constants define a constant value in the message file.

Services
ROS services are a kind of request/response interaction between processes. The reply and request data types can be defined inside the srv folder inside the package (`my_package/srv/MyServiceType.srv`)[11].

A.2.5.3 Computation graph level

ROS uses a network of processes[11] to implement its computation; the peer-to-peer computation network of ROS processes is called *Computation Graph*. The basic building blocks of the Computation Graph are: *nodes*, *Master*, *Parameter Server*, *messages*, *topics*, *services*, and *bags*, all of which provide data to the Graph in different ways.

[11] Processes are called *nodes* in ROS environment.

Figure A.8: ROS graph level

Nodes
Nodes are processes that perform computation; each of them is written using ROS client libraries which are also used to implement different types of communication techniques in nodes. Notably, one node can communicate with other nodes using *topics*, *services*, and *parameters*. To identify a node in the system a name should be assigned to it. The `rosnode` command can be used to get introspect ROS nodes. The list of `rosnode` usage can be summarized as follows:

- `$ rosnode info [node_name]`: it prints the information about the node
- `$ rosnode kill [node_name]`: it teardowns a running node
- `$ rosnode list`: it list the running nodes
- `$ rosnode machine [machine_name]`: it lists the nodes running on a particular machine or a list of machines
- `$ rosnode ping`: it checks the connectivity of a node
- `$ rosnode cleanup`: it purges the registration of unreachable nodes[11]

Figure A.9: ROS computational communication's graph

Master

The ROS Master provides name registration and lookup system nodes; therefore, nodes would not be able to find each other, exchange messages, or invoke services without it. ROS requires only one master per system.

ROS Master has many features in common with a DNS server; indeed when a node starts in the ROS system, it looks for the master and registers its name in it. In that way the Master has the details of all nodes currently running on the ROS system, and when any of these change, it generates a call-back and update with the latest details.

ROS client libraries, as `roscpp` and `rospy` for instance, are used to write nodes. Those libraries interact with the Master using `XMLRPC` (XML Remote Procedure Call) based APIs, which act as the backend of the ROS system APIs. ROS master's IP and port are stored in an environment variable called `ROS_MASTER_URI`, that latter is used by nodes find ROS Master and start the communication.

Registration process

A node starts publishing a topic, providing details, like name and data type, about it to ROS Master as soon as the Master receives those data will check if nodes subscribed to that topic are present in the system. If any nodes are subscribed to the same topic, the publisher details will be shared by ROS Master with them, after that procedure has been completed, the Subscriber and Publishers nodes will interconnect using the *TCPROS protocol*(mainly based on TCP/IP sockets). In these terms, ROS Master's primary purpose is to connect the nodes, when it performs that operation it has no longer a role in controlling

Figure A.10: ROS: master-client registration's graph

them[11].

Figure A.11: ROS publish-subscribe's graph

Parameter Server
The parameter server is the part of the Master who provides a mechanism to save by key the data in a central location accessible by all nodes. In this way, nodes can modify the information present in the server[11].

Messages
Nodes communicate (peer-to-peer) with each other passing messages to topics.

ROS naming convention is used to name message types: package name + message name; thus ROS can automatically generate source code to use messages in a ROS node, both in C++ and python.

Nodes can also communicate using service calls; indeed, services too are messages, especially the service message definitions are defined inside the `.srv` file.

Furthermore, apart from the message data type, an MD5 checksum comparison is used by ROS to confirm whether the publisher and subscriber exchange the same message data types.

ROS inbuilt tool to get information about ROS messages is called `rosmsg`; it can be used with a set of parameters to perform different operation as follows:

- `$ rosmsg show [message]`: it shows the message description
- `$ rosmsg list`: it lists all messages
- `$ rosmsg md5 [message]`: it displays `md5sum`[12] of a message
- `$ rosmsg package [package_name]`: it lists messages in a package
- `$ rosmsg packages [package_1] [package_2]`: it lists packages that contain messages[11]

Topics
ROS messages are routed via a transport system with publish-subscribe semantics using strongly typed message buses called topics. Nodes send messages through a topic, by publishing it; topics have a name that is used to identify the content of the message, therefore a node that is interested in a certain kind of data will subscribe to the appropriate topic. Nevertheless, since the publish-subscribe model implements many-to-many communication, it is not suitable for request/reply interactions; to implement that kind of communication, one has to use ROS services. To decouple the production of data from its consumption publishers and subscribers are not aware of each others' existence. That is the reason why in a ROS system one could have multiple concurrent publishers and subscribers for a single topic, a single node may publish and/or subscribe to several topics, and also a node could even subscribe to a topic that might not have any publisher.

The ROS nodes communicate with topics using TCP/IP-based transport known as `TCPROS`[13]. This method is the default transport method used in ROS. Nevertheless, ROS also provides another type of communication that is called

[12] MD5 is a computer program that calculates and verifies 128-bit MD5 hashes, as described in RFC 1321. The MD5 hash functions as a compact digital fingerprint of a file[35]

[13] TCPROS: is a transport layer for ROS Messages and Services. It uses standard TCP/IP sockets for transporting message data. Inbound connections are received via a TCP Server Socket with a header containing message data type and routing information.[25]

`UDPROS`, that latter has low-latency, loose transport, and is only suited for teleoperation[11].

Likewise for messages, also for topics exists a tool to retrieve information about them: that last is called `rostopic` and its main usage can be summarized as follows:

- `$ rostopic bw /topic`: it displays the bandwidth used by the given topic
- `$ rostopic echo /topic`: it prints the content of the given topic
- `$ rostopic find /message_type`: it finds topics using the given message type
- `$ rostopic hz /topic`: it shows the publishing rate of the given topic
- `$ rostopic info /topic`: it prints information about an active topic
- `$ rostopic list`: This command will list all active topics in the ROS system
- `$ rostopic pub /topic message_type args`: it publishes a value to a topic with a message type
- `$ rostopic type /topic` : it displays the message type of the given topic[11].

Figure A.12: ROS client-server interaction's graph focused on topics and services

A.2.5.4 Services

The ROS services are a type of request-response communication between ROS nodes. One node will send a request and wait until it gets a response from the other. analogously to the message definition, a service description language is used to define the ROS service types (`.srv`).

```
#Request message type
string str
---
#Response message type
string str
```

Listing A.1: Example of service description format

As can be seen, one can define a service definition that contains two parts: one is for requests, and the other is for responses. Indeed in the example provided the first section of the service is the message type of request, and it is separated by "—" from the following section that is the message type of response. In ROS services, one node acts as a ROS server in which the service client can request the service from the server. If the server completes the service routine, it will send the results to the service client. As well as ROS messages, also ROS services use MD5 checksum to control service message correctness.

There are two tools to get information about the ROS service: the first tool is `rossrv`, which is similar to *rosmsg* and is used to get information about service types, whereas the second is *rosservice*, is used to list and query about the running ROS services.

`rosservice` usages can be sum up as follows:

- `$ rosservice call /service args`: it calls the service using the given arguments
- `$ rosservice find service_type`: it finds services in the given service type
- `$ rosservice info /services`: it prints information about the given service
- `$ rosservice list`: it lists the active services running on the system
- `$ rosservice type /service`: it prints the service type of a given service
- `$ rosservice uri /service`: it prints the service `ROSRPC URI`[11]

Figure A.13: ROS client-server synchronous interaction's diagram

Bags
ROS uses a standard format called Bags to save and playback ROS message data; notably, a bag file stores ROS message data from topics and services. Bag files are defined by files having `.bag` extension. Data logging is rosbag main purpose; indeed, the robot information can be logged and visualized online whereas processed offline.

The rosbag command is used to work with rosbag files. Here are the commands to record and playback a bag file:

- `$ rosbag record [topic_1] [topic_2] -o [bag_name]`: This command will record the given topics into a bag file that is given in the command. We can also record all topics using the -a argument.
- `$ rosbag play [bag_name]`: This will playback the existing bag file.There is a GUI tool to handle record and playback of bag files called `rqt_bag`[11].

Community level
The community-level purpose is to enable communities to exchange software and knowledge. ROS has three different types of resources that habilitate the aforementioned process: wiki, distributions, and repositories.

Wiki
The ROS community Wiki is the primary forum for documenting data about ROS; it provides a ways for anyone to contribute to ROS ecosystem by uploading ones own documentation, suggesting corrections or updates, as well as writing tutorials, and so forth.

Figure A.14: ROS Community level's basic components

Distributions
ROS distributions are a collection of versioned meta-packages that can be installed; they maintain consistent versions across a set of software[17]. Similar to Ubuntu and Android, ROS has been naming their versions to start in alphabetical order; for instance, the Kinetic Kame version is the 11th version thus starting with the alphabet K. Moreover, each version has an illustration in the form of a poster with a turtle icon, which is also used in the official simulation tutorial called *turtlesim*[14]. The turtle symbol for ROS was stemmed from the educational programming language Logo. Indeed in 1969, a turtle robot was developed by Papert team using Logo, the aforementioned device was able to move on the floor drawing pictures according to the instructions given by the computer. Based on this robot, ROS *turtlesim* was developed and later also Turtlebot robots[19].

[14]Turtlesim simulator is analyzed in the use case section

Distro	Release date	Poster	*Tuturtle*, turtle in tutorial	EOL date
ROS Noetic Ninjemys (Recommended)	May 23rd, 2020			May, 2025 (Focal EOL)
ROS Melodic Morenia	May 23rd, 2018			May, 2023 (Bionic EOL)
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame	May 23rd, 2016			April, 2021 (Xenial EOL)
ROS Jade Turtle	May 23rd, 2015			May, 2017
ROS Indigo Igloo	July 22nd, 2014			April, 2019 (Trusty EOL)
ROS Hydro Medusa	September 4th, 2013			May, 2015
ROS Groovy Galapagos	December 31, 2012			July, 2014
ROS Fuerte Turtle	April 23, 2012			--
ROS Electric Emys	August 30, 2011			--
ROS Diamondback	March 2, 2011			--
ROS C Turtle	August 2, 2010			--
ROS Box Turtle	March 2, 2010			--

Figure A.15: ROS distributions

Repositories
ROS depends on a federated network of code repositories, where several institutes can elaborate and release their robot software components[11].

A.2.5.5 Names

Graph Resource Names
Graph Resource Names provides a hierarchical naming structure that is used for all resources in a ROS Computation Graph. It is a significant technique for providing encapsulation: each resource can create resources within their namespace, and they can access resources within or above their namespace.

To facilitate programming names are resolved relatively; thus, resources are unaware of the namespace in which they are located; this is the reason why nodes that work together can be written as if they are all in the top-level namespace[11].

Package Resource Names
To streamline the procedure of referring to files and data types on disk Package Resource, besides File System-Level concepts are used. Package Resource Names are basic structures made up of the name of the Package that the resource is aside from the name of the resource. They can be used to refers to message types, service types, as well as, nodes. For instance, the name `std_msgs/String` refers to the "String" message type in the `std_msgs` Package[11].

A.3 Catkin

A.3.1 Overview

`catkin` is the official build system of ROS starting from *ROS Groovy* and the successor to the original ROS build system, `rosbuild`. catkin combines CMake macros and Python scripts to provide some functionality on top of CMake's normal workflow (improved automatic dependencies management and compilation of large project)[5].

catkin was designed to be more conventional than `rosbuild`, allowing for better distribution of packages, better cross-compiling support, and better portability. catkin's workflow is very similar to `CMake`'s but adds support for automatic 'find package' infrastructure and building multiple, dependent projects at the same time.

The name catkin comes from the tail-shaped flower cluster found on willow trees[15].

the catkin build system is organized in a workspace containing different spaces and packages, this feature is very useful for having a common files/directory structure and for building multiple packages with complex dependencies.

[15] a reference to Willow Garage where catkin was created

Figure A.16: ROS Catkin: main folder description

Figure A.17: ROS Catkin: typical structure of catkin Source Space

For further information about the catkin build system look at the following URL http://wiki.ros.org/catkin/conceptual_overview.

Why Does ROS Have a Custom Build System?
ROS is a very large collection of loosely federated packages. That means lots of independent packages which depend on each other, utilize various programming languages, tools, and code organization conventions. Because of this, the build process for a target in some packages may be completely different from the way another target is built. catkin specifically tries to improve development on large sets of related packages consistently and conventionally. In other words, both `rosbuild` and now catkin aim to make building and running ROS code easier by using tools and conventions to simplify the process. Efficiently sharing ROS-based code would be more difficult without it.

Decoupling from ROS
One of the philosophies behind ROS is to minimize the number of ROS-specific tools needed to create, manage, and utilize ROS packages and to always try to defer to well-established, widely-used, third-party, open-source tools (e.g. using libtinyxml instead of writing a custom XML parser) instead. Catkin is independent of the ROS ecosystem and can even be used on non-ROS projects. That means catkin lets you easily mix your codebase with non-catkin projects. Catkin adds a lot of features on top of vanilla CMake that make it an appealing development tool even for non-ROS-related projects.

Publish Subscribe

Introduction

The publish/subscribe pattern is a well know interaction paradigm that allows subscribers to express their interest in one or more events and be notified if any event, generated by a publisher, matches their registered interest.

The basic system model for publishing/subscribe interaction relies on an event notification service providing storage and management for subscriptions and efficient delivery of events.

Subscribers register their interest in events by calling a subscribe() operation on the event service, without knowing the effective sources of these, moreover, they can use symmetric operation unsubscribe() to terminate a subscription. It is important to underline that the subscription information remains stored just in the event service and is not forwarded to publishers.

Symmetrically, publishers can generate events by calling a publish() operation, subsequently, the event service propagates the event to all relevant subscribers (every subscriber will receive an event for every event conforming to its interest).

Figure A.18: Publish subscriber pattern interaction's graph

Decoupling

The decoupling that the event service provides between publishers and subscribers can be decomposed along the following three dimensions.

- **Space decoupling**: the interacting parties do not need to know each other; the publishers do not hold references to the subscribers or know how many of them are, the speculate situation happens to the subscribers.
- **Time decoupling**: the interacting parties do not need to be actively participating in the interaction at the same time. In particular, the publisher might publish some events while the subscriber is disconnected, and conversely, the subscriber might get notified about the occurrence of some event while the original publisher of the event is disconnected.
- **Synchronization decoupling**: publishers are not blocked while producing events, and subscribers can get asynchronously notified of the occurrence of an event while performing some concurrent activity. Decoupling these operations increases scalability by removing all explicit dependencies between the interacting participants[9].

A.4 Willow Garage

Willow Garage was a robotics research lab and technology incubator started in late 2006 by Scott Hassan aiming to develop hardware and open source software for personal robotics applications[36].

In January 2007, the company hired its first employees: Jonathan Stark, Melonee Wise, Curt Meyers, and John Hsu to work at the first projects of the Willow Garage, among which there was also an SUV entrant into the DARPA Grand Challenge and an autonomous solar-powered boat for deploying scientific payloads in open oceans. The next year, Eric Berger and Keenan Wyrobek pitched Willow Garage on collaborating a common hardware (PR1) and software (ROS) platforms and the idea of creating a Personal Robotics Program at Willow Garage.

Willow Garage continues to work on a large number of projects until it shut down in early 2014[37].

Figure A.19: Willow Garage logo

A.5 Docker

"Docker is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers."

Image A Docker image is a file. An image never changes; you can not edit an existing file. Creating a new image happens by starting from a base image and adding new layers to it. We will talk about layers later, but you should think of images as immutable, they can not be changed after they are created. Containers are created from images, so when we ran hello-world twice we downloaded one image and created two of them from the single image.

Well then, if images are used to create containers, where do images come from? This image file is built from an instructional file named Dockerfile that is parsed when you `run docker image build`. Container Containers only contain that which is required to execute an application; and you can start, stop and interact with them. They are isolated environments in the host machine with the ability to interact with each other and the host machine itself via defined methods (TCP/UDP).

Bibliography

[1] D. Adams. *The Hitchhiker's Guide to the Galaxy*. San Val, 1995.

[2] axios http.com. *Axios - Documentation*. Available at https://axios-http.com/docs/intro.

[3] Brian Bergstein. Inventors (2013). Available at https://www.technologyreview.com/innovator/morgan-quigley/.

[4] Shaumik Daityari. Angular vs react vs vue: Which framework to choose in 2021.

[5] Diego Dall'Alba. Introduction to ros. Altair robotics lab nta3, Università di Verona, 2020.

[6] devops withdocker. Devops with docker. Available at https://devopswithdocker.com/part1/.

[7] ERC. About the erc. Available at https://roverchallenge.eu/en/about-the-erc/.

[8] ERC. European rover challenge. Available at https://en.wikipedia.org/wiki/European_Rover_Challenge.

[9] Patrick Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35:114–131, 06 2003.

[10] Interaction Design Foundation. *User Centered Design*. Available at https://www.interaction-design.org/literature/topics/user-centered-design.

[11] Lentin Joseph. *Mastering ROS for Robotics Programming*. PACKT publishing, 2015.

[12] jwt.io. *JSON Web Token*. Available at https://jwt.io/introduction.

[13] Anis Koubaa. *Robot Operating System (ROS), The Complete Reference (Volume 1)*. Janusz Kacprzyk, Polish Academy of Sciences, 2019.

[14] mongoosejs.com. *Mongoose - Documentation*. Available at https://github.com/Automattic/mongoose.

[15] Jakob Nielsen. *How to Conduct a Heuristic Evaluation*. Available at https://www.nngroup.com/articles/how-to-conduct-a-heuristic-evaluation/.

[16] npmjs.com. *ApexChart*. Available at https://www.npmjs.com/package/apexcharts.

[17] Jason M. O'Kane. *A Gentle Introduction to ROS*. Independently published, October 2013. Available at http://www.cse.sc.edu/~jokane/agitr/.

[18] Chadia Abras Diane Maloney-Krichmar Jenny Preece. *User Centered Design*. Available at https://d1wqtxts1xzle7.cloudfront.net/6190316/10.1.1.94.381-with-cover-page-v2.pdf?Expires=1629287796&Signature=ZtRWUtaeLUKqJxl-jQYUfJ-CUCNZoOm5nArDEBf~qAq3FqGSTAV6eKDl9mPBLpWussm99MKRHFd33tYB7AijX6M
_&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA.

[19] YoonSeok Pyo, HanCheol Cho, RyuWoon Jung, and TaeHoon Lim. *ROS Robot Programming)*. ROBOTIS Co.,Ltd., 2017.

[20] Morgan Quigley, Eric Berger, and Andrew Y. Ng. Stair: Hardware and software architecture. Technical report, Stanford University, 2007.

[21] Morgan Quigley, Eric Berger, Andrew Y. Ng., Brian Gerkey, Josh Faust, Ken Conley, Tully Foote, Jeremy Leibs, and Rob Wheele. Ros: an open-source robot operating system. Technical report, Stanford University and Willow Garage and University of Southern California, 2009.

[22] Morgan Quigley, Brian Gerkey, and William D. Smart. *Programming Robots with ROS*. O'Reilly Media, In, 2015.

[23] ros.org. *Mjpegcanvasjs*. Available at http://wiki.ros.org/mjpegcanvasjs.

[24] ROS.org. *ROS-Messages*. Available at http://wiki.ros.org/Messages.

[25] ROS.org. *ROS-TCPROS*. Available at http://wiki.ros.org/ROS/TCPROS.

[26] ROS.org. *ROS-Tutorial-Creating Msg And Srv*. Available at http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv.

[27] ROS.org. *ROS-Tutorial-Examining Publisher Subscriber*. Available at http://wiki.ros.org/ROS/Tutorials/ExaminingPublisherSubscriber.

[28] ROS.org. *ROS-Tutorial-Writing Publisher Subscriber(python)*. Available at http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29.

[29] ros.org. *Rosbridge_server*. Available at http://wiki.ros.org/rosbridge_server.

[30] ros.org. *Roslib*. Available at http://wiki.ros.org/roslib.

[31] ros.org. *Web_video_server*. Available at http://wiki.ros.org/web_video_server.

[32] Wikipedia. Andrew ng. Available at https://en.wikipedia.org/wiki/Andrew_Ng.

[33] Wikipedia. *Bcrypt*. Available at https://en.wikipedia.org/wiki/Bcrypt.

[34] Wikipedia. *Boorstrap*. Available at https://en.wikipedia.org/wiki/Bootstrap_(front\protect\discretionary{\char\hyphenchar\font}{}{}end_framework).

[35] Wikipedia. md5sum. Available at https://en.wikipedia.org/wiki/Md5sum.

[36] Wikipedia. Willow garage. Available at https://en.wikipedia.org/wiki/Willow_Garage.

[37] Keenan Wyrobek. The origin story of ros, the linux of robotics. Available at https://en.wikipedia.org/wiki/Willow_Garage.