SAPIENZA
UNIVERSITÀ DI ROMA

Design and Implementation of a Modular Architecture for a Welcoming Robot

DIPARTIMENTO DI INGEGNERIA INFORMATICA, AUTOMATICA E GESTIONALE "ANTONIO RUBERTI"

Laurea Magistrale in Intelligenza Artificiale e Robotica

Kevin Munda
ID number 1905663

Advisors
Prof. Daniele Nardi
Ing. Vincenzo Suriani

Academic Year 2022/2023

Design and Implementation of a Modular Architecture for a Welcoming Robot
Tesi di Laurea Magistrale. Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: munda.kevin@gmail.com

Contents

Abstract

Nowadays, robots are beginning to be employed not only in industrial processes but also in normal daily activities in close contact with humans. Several industries are adopting the use of robots, from scientific research to entertainment, from healthcare to hospitality, all fields in which strong cooperation with humans is necessary. In this project, the robot ALTER-EGO, developed by the Italian Institute of Technology in Genova, was used to test the hosting capabilities of a robot in an environment used for conventions, namely the Temple of Hadrian in Rome, recreated through a simulation environment. The purpose of this project was to equip a robot, initially designed for tele-operations, with autonomous capabilities to be self-sufficient in interactions with people. To do this, a modular architecture was designed and implemented with the necessary functionalities required for this type of situation, from voice interaction with humans to gesture generation for non-verbal interactions, passing through autonomous localization and navigation, while paying particular attention to ensuring safe execution of the robot's actions within a human-friendly environment. Three are the interactions designed to test the performances of the developed architecture: a simple greeting, a request of information and a request to be guided to a specific destination. All interactions were successfully concluded, demonstrating how the robot is able to handle these types of situations independently. The modularity of the architecture and the way the functionalities in it have been implemented make it easy to add any new use cases or to add modules with completely new functionalities.

Chapter 1

Introduction

1.1 What is Human-Robot Interaction?

"Human-Robot Interaction (HRI) is a field of study dedicated to understanding, designing and evaluating robotic systems for use by or with humans" [1]. Nowadays, with the exponential growth of the technology industries in the last decades, intelligent robots and automated systems are widely accepted as reliable and efficient components of everyday life. At the beginning of the introduction of robots into people's routines, their use was mostly concerned to factories; today's trend instead is to insert autonomous systems in various fields, not only industrial, to improve work processes, simplify interactions, increase efficiency or allow people to avoid risky situations, in fact is possible to find robots in sectors like scientific exploration, search and rescue, hospitality, entertainment, healthcare, military and many others. Depending on the domain in which a robot is employed, it will have specific ways to interact with the surrounding space and specific type of interactions with the humans that assist its operations, therefore develop accurate behaviors for the robot to use in interactions with one or more humans is of crucial importance. Human-robot interaction is a multidisciplinary field, emerged in the mid 1990s and early years of 2000, that involves artificial intelligence, robotics, natural language, psychology, cognitive science, social sciences, engineering and human-computer interaction. If we think of a simple interaction between two people will be quite straight why so many disciplines and researchers from them are involved in this field: each one of us uses various types of languages when taking part in a discussion, like body language, natural language, facial expressions; each of them adds to the simple meaning of words a specific characteristic that alters the way in which we receive the meaning of the speech. Moreover we also follow social rules to carry on the conversation, like keep silent while the interlocutor is speaking, just to make an example. Obviously roboticists will take care of the part regarding the control schemes that allow the robot to realize actions, like speech replies or gestures, in response to some interaction, while psychologists and behavioural scholars will be in charge of developing a socially acceptable model of interaction that allows the robot to interface an interaction with a human in the same way a person would. From these concepts we can easily understand why human-robot interaction is different from human-computer interaction (HCI): it concerns machines which have complex

and dynamic control systems, characterized by a specific level of autonomy and cognition, and which operate in dynamic real-world environments. In particular, the environment in which the robot operates allows the distinction between two general categories of interaction: **remote interaction**, when the human and the robot are spatially or even temporally separated, so they are not in the same environment, and this is referred to as *tele-operation*; **proximate interaction**, when the human and the robot are co-located, so they share the same space. Real examples of these kind of interactions are showed in Fig. 1.1. A robot requires a specific set of capabilities depending on which category it belongs, for example a tele-operated robot should require mobility and physical manipulation, while a robot that has to interact with a human in the same environment should be equipped with social and physical interaction capabilities.

(a) **(b)**

Figure 1.1. Examples of remote and proximate interactions. **(a)** A surgeon using a control station through which performs surgery remotely. **(b)** A robot hands a tool to a person while he works

1.2 The HRI Problem

"The HRI problem is to understand and shape the interactions between one or more humans and one or more robots" [1]. When thinking about an interaction there are several aspects to consider. To better comprehend how an interaction between a robot and a human is structured we can analyse its main components: **autonomy**, **information exchange**, **structure of the team**, **adaptation, learning and training** and **structure of the task**. Even the so called autonomous robots need to interact with the humans. This because autonomy does not mean total independence from external contribution, but it is referred to the capability of replying with a specific action to an external input coming from the environment in which the robot acts. In fact, autonomy in this field is considered productive if it is useful to achieve a beneficial interaction between a robot and a human. Depending on the nature of the task the robot is designed for, it will need a specific *level of autonomy (LOA)*, that defines how much the robot can act depending on its own choices; this level can range from zero autonomy (i.e. totally tele-operated) to full autonomy (i.e. totally independent from any approval for its actions). A graph,

taken from [1], relating levels of autonomy to the type of interaction with humans is showed in Fig. 1.2. From a HRI point of view, developing an efficient peer-to-peer interaction between a robot and a human is more difficult than developing a fully autonomous robot, because the robot must be able to cooperate with a partner and execute autonomous choices at appropriate times, being also able to conduct social interaction.

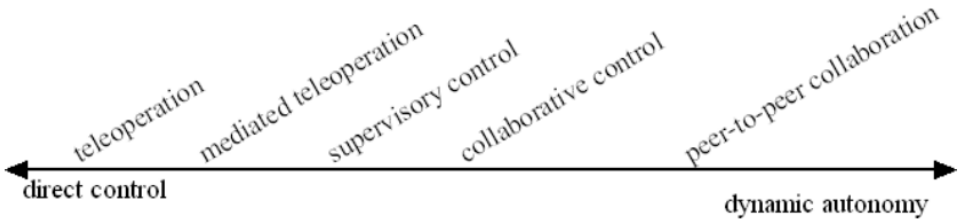


Figure 1.2. Graphic taken from [1] showing the relation between the type of human-robot interaction and the relative level of autonomy

The way in which the interaction is carried on is defined by how the information is passed between a human and a robot. This exchange is basically defined by two aspects: the *communication medium* and the *communications format*. The first one can be realized in different ways: *visual displays*, which typically present a graphic user interface or augmented reality interfaces; *gestures*, like hand and facial expressions; *speech* and *natural language*, which include both auditory speech and text-based responses; *non-speech audio*, used in alerting for example; *physical interaction* and *haptics*, used in tele-operation or in augmented reality. The recent approaches try to combine all these kind of means into *multimodal interfaces*, in order to create more natural interactions but also to give more complete ways through which interact with the robot. For what concerns the *format* of the information exchange, it depends basically on which of the previous medium is used. To make some examples, an audio-based exchange can include sounds related to specific situations, like an alert, or it can use natural language to conduct a dialogue with the human; in the last case, the rules of such exchange must be defined. Instead, in a context of tele-operation, the human that controls the robot will wear most likely some wearable device that will transfer to his body some haptic information through vibration for example, which can suggest the proximity to an object or a surface. The structure of the humans-robots team and how many components it has is an aspect to be defined before starting any operation. To operate in a productive way there should be a minimum number of human operators that could control efficiently a fixed number of robots; this minimum number depends on the level of autonomy of the robotic units. A particular case of team structure is when is also present an intelligent interface that acts as intermediary between the robot and the human. The software agent can monitor the actions and the behaviours of both robots and humans to better coordinate the work of the team. Another important aspect to consider is the learning and training of both humans and robots. An implicit goal of HRI is to design systems that require very short and simple training for human operators. From one side, it could be helpful to reduce the amount of

training required by a human to interact with a robot, however this criterion can be applied only to specific domains, because on the other side, there are cases in which the human must have been prepared with a proper training. For example, people that interact with robots employed in therapeutic and educational field can directly adapt and learn from the interaction, without having any particular training, while a human operator that executes a surgery operation with a tele-operated robot should receive an appropriate training in order to use safely the tools of the robot. Also robots learn and improve their capabilities: they improve perceptual capabilities through efficient communication with a human, improve planning and reasoning skills through interaction and so also autonomous capabilities. There are several ways in which a robot can learn, like teaching or programming by demonstration, task learning and skill learning. The last thing to consider for defining completely an HRI problem is the structure of the task. A robot is introduced in some domain to prevent man from performing dangerous tasks or to simplify the task for the humans. In both cases, the introduction of the robot changes completely the way in which the task should be accomplished. In this case the new task of the human should be redefined in order to be complementary to the part of the task achieved by the robot.

1.3 Interaction Model

When is the turn of the robot to take action in the environment in which is operating, it must follow a well defined scheme. Analyzing its current state and the input received from a human or from the environment, the robot will execute an action relative to the perceived event. Obviously the type of interaction and the corresponding action that it will execute depend on the type of task for which it is employed, so these aspects are domain-dependent. However, it is possible to define ah high-level scheme that a robot should follow when involved in any kind of interaction. The model defined in this section is taken from human-computer interaction and is called "Norman's seven stages of interaction" [2]. As the name suggests, Norman considers seven different stages of interaction:

1. **Formulation of the goal** - Define in high-level terms what is the goal to achieve
2. **Formulation of the intention** - Analyze specifically what are the characteristic that satisfy the goal
3. **Specification of the action** - Define the sequence of actions that realize the intention
4. **Execution of the action** - Physical execution of the action
5. **Perception of the system state** - Evaluate what has occurred based on the action specification and the action executed
6. **Interpretation of the system state** - Use system knowledge to understand what happened and what is the state after the action

7. **Evaluation of the outcome** - Understand if a progress has been made and decide the next action

These stages are repeated until the goal and the intention are achieved or until the goal has to be changed. The same author of this model, found that in its formulation there are two possible issues: the gulf of execution, which is a mismatch between user's intention and available actions of the system, and the gulf of evaluation, which is a mismatch between the system's representation and the user's expectation. When grounding this model to an HRI domain, it must be taken into account the role that the robot plays in the interaction with the humans. This means that the robot will act following a precise interaction model that depends on the role it has. Scholtz defined a set of possible interactions in HRI domains [3].

- **Supervisor interaction** - A supervisor controls the situations. For example, he evaluates the performance of a group of robot, equipped with planning systems, that should achieve a specific goal. The supervisor can modify the plans or specify an action, using a formal representation for them.
- **Operator interaction** - An operator has the task to modify internal software or models when the robot behaviour is not satisfactory. He has to determine if the actions are executed correctly and in accordance to the desired goal.
- **Mechanic interaction** - A mechanic physically intervenes on the robot. Also this figure has to determine if the action are carried out correctly, this time from an hardware point of view.
- **Peer interaction** - Teammates of the robot propose new actions within the plan that lead to the goal. This means that a peer can suggest a new way to realize the intentions or to achieve the goal but related to the same plan, only the supervisor has the possibility to change the larger goal or the intentions. This kind of interactions can be carried on using high-level behaviours, suggested with simple units of dialogue (e.g. "follow me", "wait until").
- **Bystander role** - The bystander can interact with the robot using just a subset of the available actions. Here the problem is to advise the bystander of the capabilities of the robot that he can exploit.

1.4 A Welcoming Robot

After a general introduction about what is HRI, how to approach an HRI problem and what are the possible relationships between a robot and a human, in this section are briefly described the characteristics of this project. The main aim is to implement an efficient software architecture that can make autonomous a robot that welcomes guests during an event. Imagine to take part to a conference and when you arrive at the building where the event takes place, instead of being accepted by a valet, you are welcomed by a robot. In order to be able to play such a role, the robot should be equipped with *multi-modal interaction capabilities*, to better communicate in a way that resembles the human one (so with speech replies,

gestures, facial expressions), *dialogue capabilities*, to understand the requests of the users and be able to adequately reply, *mapping* and *planning capabilities*, so that it can orientate itself inside the building and give also correct information about specific locations inside it. These are the main characteristics developed in this work. In particular, the implemented architecture has made autonomous the ALTER-EGO robot, originally designed for tele-operation, making it capable of responding to certain events. Specifically, the robot is able to reply in a coherent way to verbal interactions, ranging from a simple greeting to a more articulated request about the events held during the day. Moreover, the robot is capable of satisfying requests regarding the location of a room inside the building in which it is located, being also able to physically lead the guest to the desired destination. These are the main features developed in this work, which allow a robot to have the necessary functionalities to be used in the hospitality industry. Obviously, these are a subset of all the possible features that can be implemented in a machine of this kind, however they are sufficient to respond to the basic requests of a user. Precisely to encourage the addition of new features, the implemented functionalities can be thought also as a template for other possible types of interaction, meaning that the same structure defined for a particular event can be adapted easily to another request, by modifying the parameters that define the new interaction. The developed architecture and all the interactions were tested in a simulated environment with a robot model based on the real structure of the ALTER-EGO robot inserted into a simulated building, with a user being able to interact with the simulator through a microphone. The next chapters are structured as follows: in chapter 2 are mentioned some of the works carried out in the last years in HRI applied to several fields, in chapter 3 are described the main software and hardware tools used in this project, in chapter 4 are treated in detail the functionalities of each module of the software architecture developed for ALTER-EGO, in chapter 5 the execution times of the main functions of the modules are shown and analyzed and at last chapter 6 concludes this paper by summarizing the results obtained and suggesting possible future developments.

Chapter 2

Related Works

In this chapter are briefly described some of the works in human-robot interaction developed in the last years. As explained in the Introduction, robots are starting to be employed in various fields, from entertainment to healthcare, from rescue to hospitality. In particular, the papers reported here deal with the following topics: verbal communication with anthropomorphic and non-anthropomorphic robots used for entertainment [4]-[5]-[6], use of robots in museums for educational purposes [7]-[8]-[9], robotics applied to space applications [10], robotics applied to support people in need [11]-[12], robots used in catering [13] and rescue robotics [14]-[15]-[16]. Arkin et al. [4] developed an architecture based on ethological and emotional models for autonomous dog-like behaviour. In particular they combined the ethological model obtained from the studies on canine behaviours with an emotional basis, thus using the first one to define a possible set of behaviours and the second one to select a specific behaviour based on the external stimuli and the internal drives. To implement the internal state of the robot they used a set of internal variables which are regulated based on the "emotions" caused by a specific event. Their experiments were first realized with Sony's AIBO robot and then extended to the humanoid robot SDR-4X, showing how their emotionally grounded architecture was successful in learning new objects by associating their effect on motivational and emotional variables which define how to behave when the robot encounters these objects. Brezeal [5] studied the way in which a person naturally interacts with a robot able to interpret and respond to human social cues. In particular, this work focuses its interest in exploring dynamic, expressive and unconstrained face to face social interaction between a human and an anthropomorphic robot called Kismet in two case of study: the communication of affective intent and the dynamics of proto-dialog between human and robot. The experimental data demonstrated that for social interactions with humans, expressive robotic faces are a benefit that improve the quality of the interaction for both the robot and the human. In particular, the researchers noticed that, in the case of communicating affective intent, people used the robot's expressions to ensure that the correct intent was understood, while in the case of proto-conversation, the subjects used the robot's cues to regulate when they should exchange turns. In [7], the authors have created a robotic system for educational purposes to be placed within a museum; specifically this system aims to bring humans closer to the world of insects through tele-presence. The Insect

Telepresence project has developed a robotic tool that allows the visitor to see insect world from their own point of view; the robot consists of an implant controlled by a joystick through which the user moves a camera positioned inside the display case in which the insects are located, the camera images are then projected to a screen placed in front of the user, realizing tele-embodiment in the insect world. This project has achieved very satisfactory results, increasing visitor interest, from a few seconds of attention to almost more than a minute per individual user, for that specific section of the museum, providing a more realistic and educational experience. In [8], the mobile robot Eldi has been employed in daily operation at the Elder Museum of Science and Technology at Las Palmas. To entertain the guests, the robot projected over a glass board different shows and played different games, like an instance of 8-puzzle interacting with the user. Moreover, it was able also to perform a choreography combining music, video and game board light effects. Another project developed in a museum is the one of Thrun et al. [9], where the robot Minerva was employed as an interactive tour-guide. The task designed for the robot was to attract people and explain to them the exhibits while guiding them through the museum. The main purpose of this task was to test probabilistic algorithms used for planning, control and perception. However, Minerva was also provided with learning algorithms at the user interaction level that gave her the ability to learn behaviours for attracting people and to design tours so as to meet the desired tour-length regardless the crowding in the museum. Moreover, the robot was equipped with a web interface that gave people control of it when the museum was closed to the public. Minerva successfully operated for a two-week period at the Smithsonians's National Museum of American History, completing 620 tours and visiting 2.668 exhibits. An interesting project concerning the use of a humanoid robot in space is presented in [10]. In this work the authors presented Robonaut, the first anthropomorphic robot possessing the fine motion and force-torque control required for dexterous tasks needed in space environments. The purpose of this robot is to act as an assistant to astronauts during their duties, handling lower-skilled work and saving human time for more important tasks. Robonaut is able to manage tools and science instruments; in particular it was successfully tested in representative tasks, including those for applications in space, which require complex manipulation, in geology, that require significant strength and the ability to handle multiple tools, and in medicine, which require fine positioning and dexterous movement. A very useful project carried out in the healthcare field is presented in [11]. The authors of this paper developed an architecture to make the Care-O-Bot II robot capable of assisting an elderly person or a person with disabilities. In fact, this project is motivated by wanting to implement a system that can enable the above mentioned people to live independently in their homes. The robotic platform, which consists of a 6 DOF manipulator arm, adjustable walking supporters, a tilting sensor head, and a hand-held control panel, was equipped with all the necessary modules to be useful to the people it addresses, modules that made it capable of performing household tasks, such as picking up and carrying objects, being supportive during a walk, avoiding obstacles, and being able to socialize with the user. Another project similar to the previous one is the one carried out by Pineau et al. [12], where the authors tested successfully a mobile robotic assistant in a nursing home. The robot had the task to provide reminders and guidance to the elderly residents and support

nurses during their daily activities. In addition to offering physical help, a robot can also be built to offer mental help. This is the case with the Paro robot, whose appearance and functionalities have been evaluated by a large number of users in [6]. Paro is a mental commit robot, a class of robots whose purpose is to elicit mental effects, such as pleasure or relaxation, in the person using it. Its appearance was designed using a baby seal as a model and its surface was covered with white fur. The purpose of the authors of this paper was to have as many users as possible, belonging to different nations, test the functionalities of this robot, to study the different reactions and evaluations of the various groups, so that these responses could be used to evaluate possible improvements to be implemented to the robot's current capabilities and its design to make it more suitable for the type of interaction for which it was designed. An example of robot employed in the hospitality industry is presented in [13]. After developing an architecture that implements the basic functionalities for interacting with customers in an environment such as a bar or restaurant, the robot EURKEA Gen-1 was employed as a waiter in a coffee shop in Cardiff. The robot is capable of playing games with the user, singing, dancing, playing videos, and answering questions, but it also has the ability to register users in its database by associating to each one name and photo in order to perform customized features through face recognition. There are also important studies in the rescue robotics field or more generally in those areas that can be dangerous to humans. In these types of situations, the environment in which the robot must operate is often uncertain and dangerous, difficult to traverse with ordinary vehicles. For this reason, the aim of researchers employed in this field is to develop machines with rough-terrain mobility superior to existing wheeled vehicles and one such example is BigDog [14]. BigDog is a legged robot developed at Boston Dynamics with the goal to achieve animal-like mobility on rough and rugged terrains. It is able to adapt to the terrain in two ways: it adjusts body height and attitude to conform to the local terrain and it adjusts footfall placement to compensate for orientation of the robot body and ground plane relative to gravity. The use of robots for the inspection of buildings after a disaster represents a relevant application in the rescue robotics field. The WALK-MAN project [15] was developed precisely to test the operation of robots in a situation like the one mentioned, specifically in an area destroyed after an earthquake. The results obtained are very satisfactory, the robots were able to successfully inspect visually the rooms inside the building, performing different manipulation activities for both object retrieval and path planning. Moreover, the visual information were collected and sent to a centralized control station, where they could be evaluated by experts. An application in which the use of a robot takes humans out of a potentially dangerous situation is shown in [16], where the anthropomorphic robot PETMAN was used to test chemical protective clothing, tested in a controlled chamber, where the robot was exposed to chemical agents. Chemical sensors embedded in the skin of the robot measured if and where chemical agents were detected inside the suit.

Chapter 3

Tools

This chapter describes the software and hardware tools used during the development of the project, which enabled the design of a complex architecture that managed to bring together all the functionalities offered by each of them. The main and only hardware component is the **ALTER-EGO** robot on which the architecture was tested, while the software components used are: **Robot Operating System (ROS)**, a framework specialized in creating robotic applications, **spaCy**, a Natural Language processing library, and **Gazebo**, a 3D robotics simulator.

3.1 Robot Operating System (ROS)

In this work, the Robot Operating System (ROS) is the main component which allowed the creation of the software architecture. ROS is designed to support the development of robotic software applications. It was initially developed by Stanford AI Laboratory in 2007, nowadays is maintained by the Open Source Robotics Foundation. Even if it is called operating system (OS), actually ROS is not an OS but a framework which offers some of the functionalities of an operating system, like hardware abstraction, low-level device control, device drivers, communications between processes, package management and many others. It also provides tools and libraries for building and running code across multiple devices. The main goal of ROS is to support code reuse in robotics research and development. For this reason, each process or multiple processes can be grouped into one or more packages which can be easily shared and distributed over different computers. Another important characteristic of this framework is that allows the user to design complex software without knowing how the underlying hardware works. The ROS runtime graph is a peer-to-peer network of processes, called *nodes*, with a central hub, that are loosely coupled using the ROS communication infrastructure. There are different ways in which such a network can be created through the communication systems offered by ROS: among them we found the message passing *publisher/subscriber* model, the *ROS services* and the *ROS parameters*, all used in this project. As said before, each node is a process and should be considered as a single and independent unit of execution which performs a specific task. It may happen that two nodes have to share information between them and the channel used for this exchange is called *topic*. A topic can be considered as a logical link between two

nodes where each of them sends or receives a message. Each topic is specific for a particular kind of message, ROS itself provides various standard types of messages but it offers also the possibility to define custom messages. The node that sends a message into a topic is called *publisher* while the node that waits for a message to arrive is called *subscriber*. This distinction is not binding, because each node can publish into a topic and at the same time subscribe to the same or another topic, depending on which information it needs. A service instead works in a different way: it is a specific functionality that can be requested at any time by any node with the client/server mechanism. When a node starts a service, it acts like a server because waits a request of the service function from a client node; when a client needs some data to be processed by the service function, it first makes a request to the server and then, if the service is not used by any other process, it will be free to call the service function, providing the relative data. Also services require a specific kind of data and also in this case ROS provides the possibility to define customized data. The difference between messages and service data is that messages carry on data that the subscriber node can read without returning anything to the publisher, instead when a service ends it has the capability to return some information to the node that called it; also this returned information has to be defined in the file that specifies which kind of data the service has to process.

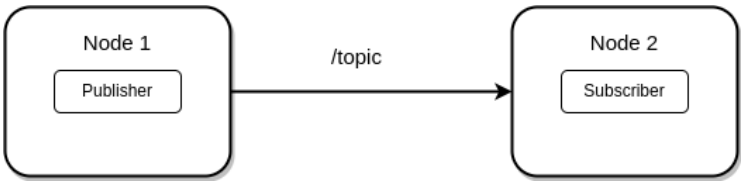


Figure 3.1. Message passing between two ROS nodes

Another way to exchange some information between nodes is offered by the ROS parameters. ROS maintains a parameter server, which is a multi-variate dictionary accessible via network APIs. In this server are stored parameters which can be retrieved by any node at runtime. Generally it is preferable to use the parameter server for static data such as configuration parameters.

3.2 ALTER-EGO

3.2.1 General overview

The robot used for developing and testing of the architecture designed in this work is ALTER-EGO [17], a robot developed by the Italian Institute of Technology. To be precise, instead of the real robot, its simulated counterpart was used that however presents the same mechanic structure and most of the hardware capabilities of the real one. The robot presents an anthropomorphic upper body mounted on a two-wheel, self-balancing mobile base. The system is provided of any sensor that can make it operate autonomously but it also can be used in tele-operation mode from a pilot station with wearable interfaces, using tele-impedance control to pair the pilot's actions with the robot's behaviour. In Fig. 3.2, taken from [17], are showed some of the kinematic and mechatronic characteristics of the robot. The mobile

base is equipped with two independent wheels actuated by two DC motors. The upper body has two arms, each one composed by a chain of five revolute joints which provide 5 degrees of freedom (DoF), with a robotic head mounted on a 2-DoF neck, that allows the head to rotate and tilt. All the DoFs of the upper body are powered by variable-stiffness actuators (VSAs), which have a stiffness behaviour similar to that of human muscles, that allow safe and correct physical interaction with the environment and with humans. The end effector of each arm is a soft, anthropomorphic, synergistic artificial hand whose structure and relative motor capacities are inspired by human motor synergies. The robot's footprint clearance is 500 mm wide and 260 mm deep, while the robot's height is 1000 mm. The weight of the robot is approximately 21 Kg and the average speed at which it moves is 0.25 m/s. This chapter is about the software and hardware characteristics of the real robot, most of them shared also by the simulated model.

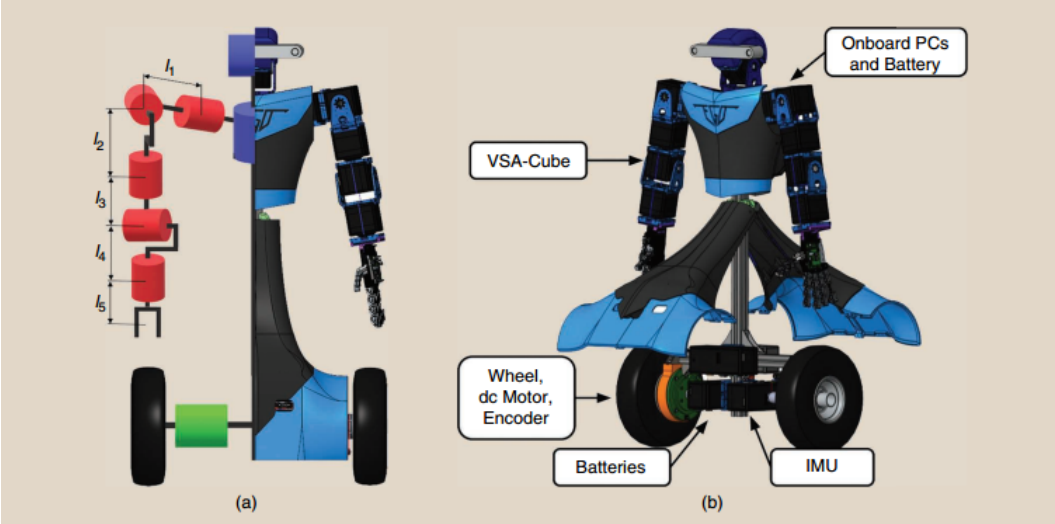


Figure 3.2. Mechatronic architecture of ALTER-EGO

3.2.2 Locomotion

The lower part of the robot is connected to the upper body by a rigid frame. The mobile base is equipped with only two wheels and this is a specific design choice. In fact, most robots employed in structured environments are equipped with three wheels, but this structure introduces tradeoffs between mobility and agility; using only two wheels the robot's footprint is minimized and the agility increased. Each wheel has a diameter of 260 mm, is equipped with a low-profile, off-road tire and is powered by a 12-V Maxon DC motor in combination with a harmonic drive gearbox (160:1). As showed in Fig. 3.2b, the lower body includes a nine-axis inertial measurement unit (IMU) between the two wheels, to estimate the pitch angle, and two magnetic encoders, to measure the wheel rotation. Moreover, the base is equipped with two Sharp infrared sensor which allow the robot to avoid and prevent collision with obstacles. The wheels are controlled independently, allowing the system to move forward, backward and turn in place. The mobile platform

requires active balance stabilization, which may incur instability issues and increase energy consumption. Balance control must be taken into account when the robot deals with manipulation tasks, because the change of the center of mass (CoM) can affect the cartesian position and orientation of the head and the end effectors. ALTER-EGO is provided with a whole-body balancing controller that takes full advantage of the system's dynamics to improve balancing performance.

3.2.3 Manipulation

As said in the overview, the end effectors of the arms are soft, human inspired hands, called SoftHand (SH), developed by the University of Pisa and the Italian Institute of Technology. A revised version of SH was used for ALTER-EGO. SoftHand is an underactuated anthropomorphic hand (19 DoF actuated by only one motor) capable of adapting its grasp to objects of different sizes, shapes and weight, including also the capability of guaranteeing a safe interaction with people and with the environment. The actuators of the arms and the neck are 12-qb move units, which are modular VSAs derived from the VSA-CUBE design, that implement an agonistic-antagonistic principle using two motors connected to the output shaft through a nonlinear elastic transmission. Both arms are connected to a frame mounted on the mobile base. Each arm has 5 DoF, meaning that there is the possibility of unreachable configurations and singularities; this structure was the result of a tradeoff which considered weight, complexity, arm length and actuator's maximum payload. Given a desired end effector pose, i.e. position and orientation, the resulting joint positions of each arm are computed with a closed-loop, inverse kinematics algorithm with damped pseudo-inverse. Given the elastic nature of VSA, to control the position of the arms in feedforward mode without a steady-state error, is necessary to compute both the desired actuator position and the expected load torque τ, to compensate for the expected elastic deflection θ. The vector τ can be obtained from robot's dynamics as:

$$\tau = B(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) - J^T f_e \tag{3.1}$$

while the expected deflection can be reconstructed by inverting the elastic model of the qb move:

$$\tau = k_1 sinh(a_1(q-\theta_1)) + k_2 sinh(a_2(q-\theta_2)) \tag{3.2}$$

where k_1, k_2, a_1 and a_2 represent model parameters of the actuator, q is the link position, and θ_1 and θ_2 are the motor positions. Because $k_1 \simeq k_2 = k$ and $a_1 \simeq a_2 = a$, it is possible to write τ as:

$$\tau = 2k cosh(a\theta_{pre}) sinh(a(q-\theta_{eq})) \tag{3.3}$$

where

$$\delta = q - \theta_{eq}, \theta_{pre} = \frac{\theta_1 - \theta_2}{2}, \theta_{eq} = \frac{\theta_1 + \theta_2}{2} \tag{3.4}$$

are the deflection, stiffness regulation and equilibrium angles. Given a desired θ_{pre} and q, is possible to obtain the expected deflection $\delta = \delta(q, \theta_{pre})$ from (3.3). Thus, the expected motor trajectory is $\theta_{eq} = q + \delta(q, \theta_{pre})$.

3.2.4 Control

Considering a simplified model, the state-space of the lower body sub-system is described by three generalized coordinates (θ, ξ and ϕ), which define respectively the semisum of the wheel angles, the robot's yaw angle and the robot's pitch angle. The scheme of the balance control mechanism is showed in Fig. 3.3, where $u = [u_1, u_2]$ represent the torque control for the wheels and $e = r - y$ is the error between the current y and required r robot states.

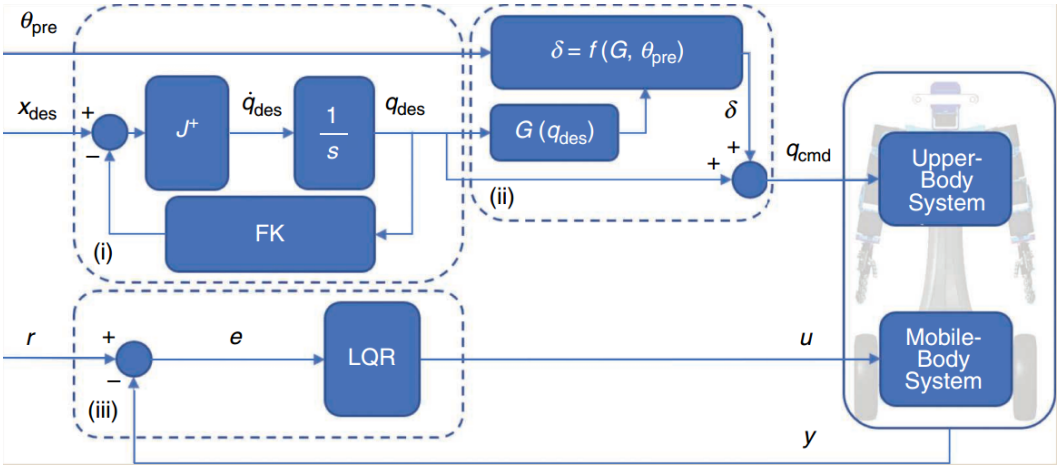


Figure 3.3. Full-state feedback control scheme with LQR taken from [17]

The developers of ALTER-EGO defined a whole-body dynamic control system that computes the joint actuation toques τ, given a desired joint-space motion. This control scheme is based on a classic LQR method but improves its basic form taking advantage of the arms for the balancing of the system. To create this system, a torque-control law in the quasi-velocity vector was developed, starting from the underactuated and kinematically constrained model of the robot. The main idea behind this scheme is here briefly explained. Consider q the generalized coordinates of the robot, n the number of DoFs, n_{fb} the number of independent variables needed to describe the floating base motion and n_c the number of constraints the robot is subject to. Let $v \in R^{n+n_{fb}-n_c}$ be the quasi-velocity vector such that $\dot{q} = S(q)v$. Consider the error dynamics:

$$\dot{v}^d - \dot{v} + K_d(v^d - v) + K_p \int_0^t (v^d - v) = 0 \tag{3.5}$$

where K_p and K_d are positive definite gain matrices and v^d are the desired quasi-velocities. The resulting generalized torques are computed as a function of the quasi-velocities vector:

$$\tilde{\tau} = \tilde{M}(\dot{v}^d + K_d(v^d - v) + K_p \int_0^t (v^d - v)) + \tilde{c} \tag{3.6}$$

where $\tilde{c} = S^T(q)(M(q)\dot{S}(q,\dot{q})v + C(q,\dot{q})S(q)v + G(q))$ and $\tilde{M} = S^T(q)M(q)S(q)$. This ensures that the resulting generalized torques are compatible with the constraints by design. In [18] there is a complete explanation of the control system

and is showed how the actual joint torques τ are obtained from (3.6) to achieve the whole-body control. A representation of this scheme is showed in Fig. 3.4.

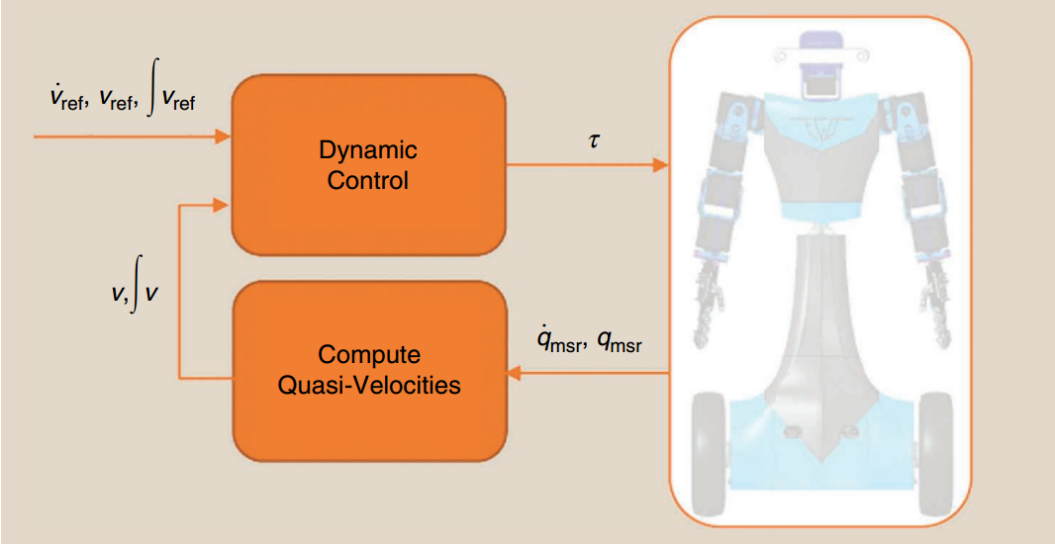


Figure 3.4. Whole-body control scheme taken from [17]

3.2.5 Sensing

The head is equipped with a Stereolabs Zed Camera, a passive red-green-blu-depth (RGB-D) camera that can take pictures, videos and a depth point cloud of the scene. The images seen by the camera can be streamed to the pilot station monitor or to the VR headset when the robot is employed in teleoperation mode. ALTER-EGO is equipped with a set of vision tools that give to it the capability of recognize objects and markers. The head architecture is completed by a 10-W speaker and a multidirectional, six-channel microphone. The two hands are equipped with position and current sensors on their motors to reconstruct the applied grasp force. Moreover, each actuation unit is equipped with three position sensors to measure the spring deflection. This measurement can be used to estimate the torque applied by each motor and to estimate the external wrenches applied to the end effectors.

3.2.6 Software and Communication Architecture

The control architecture, vision streaming and compression algorithms are managed by a computational unit. The low-level communication layer between the actuation units, wheels, sensors and end effectors is based on an RS485 protocol. The computational units are powered by a dedicated 12-V battery while the robot is equipped with a couple of 24-V batteries. The robot's software architecture is divided into four different layers. The first one comprehends the firmware running in each board used to control the joints, end effectors, motor wheels and sensors. The second layer, also embedded in the electronic boards, manages the communication bus among the different devices included in the robot's hardware. The third layer is in charge of the communication between the second layer and the ROS modules.

Each ROS module manages the nodes that deal with the functionalities of the robot (base motion, gravity compensation, balancing, localization, mapping, vision, navigation, arms inverse-kinematics, end effector wrench estimation). The fourth layer manages the communication between the robot and the pilot station and controls the robot in both autonomous and tele-operated modes. A 5-GHz wireless connection allows a bilateral communication with the pilot station for the streaming of control and vision data. A ROS communication framework is employed to receive data and send commands to the robot. In particular, from the pilot station are sent the references for head orientation, arm joint position and stiffness, hands closure and velocity of the mobile base, while the robot sends back a stream of images at a frequency of nearly 25 Hz. The aim of my project was to re-define the fourth layer, implementing an architecture able to manage the behaviour of the robot in an autonomous way for specific tasks.

3.2.7 Operating Modes

ALTER-EGO has two main operating modes: autonomous and tele-operated. For the first type, ALTER-EGO exploits the functionalities embedded in its local computational unit. In particular, for navigation, simultaneous localization and mapping, the Real-Time Appearance-Based Mapping (RTAB-MAP) algorithm is used, which is an RGB-D graph SLAM approach based on a global Bayesian loop-closure detector. A particle filter is integrated in this system, to avoid camera occlusion and slow update rate (10 Hz) that could impede robot localization in some circumstances. Moreover, ALTER-EGO is equipped with autonomous grasping and manipulation capabilities, made possible by the combination of the vision information and end effector wrench estimation. The robot has also the capability to execute cooperative manipulation tasks with a human like handling an object in co-operation with the human operator or walking hand in hand. For what concerns the tele-operation, ALTER-EGO can be controlled from a console or through an immersive VR setup. In the latter case, the setup is composed by a set of lightweight wearable devices which allow the human operator to execute the control commands freely. The standard setup is composed by two Myo armbands per arm, one placed on the forearm and one on the upper arm, and an IMU placed on the pilot's head, which is useful for the reconstruction of the position and orientation of the pilot's arms. Let ${}^{i}T_{j} \in R^{4x4}$ be the homogeneous transformation from joint i to joint j; the homogeneous transformation ${}^{S}T_{H}$ from the shoulder to the hand is obtained as:

$$ {}^{S}T_{H} = {}^{S}T_{E}{}^{E}T_{W}{}^{W}T_{H} \tag{3.7} $$

where the subscripts S, E, W and H indicate respectively shoulder, elbow, wrist and hand. The translation component of every transformation is known a priori, given by the real distances between the parts of the operator's arms. Instead the rotational component ${}^{i}R_{j}$ of the homogeneous transformations can be obtained applying a Madgwick filter on the data received from the IMU on the arm and on the hand. The result of (3.7) is scaled to match the dimensions of the robot's arms and then used as a Cartesian reference for the inverse-kinematics algorithm. Moreover, the setup has the capability to control the stiffness of the robot arms using the electromyographic data given by the armbands placed on the upper arms while

the hand closure is controlled by a linear combination of the signals coming from the armbands placed on the forearms. At last, a Wii Balance Board is employed to control the robot's mobile base by sending velocity references.

3.3 Natural Language Processing: the library spaCy

To make the robot autonomous when interacting with people, social interaction skills are required. In particular, it must be capable to understand the meaning of a specific request and respond consistently. To enable the robot to carry on these kind of interactions, it should be equipped with natural language processing (NLP) functionalities. In this project, the tool used to implement such capabilities is the library *spaCy* [19], which is a free, open-source library for advanced natural language processing in Python; it helps in building applications that analyze and understand large volumes of text. Among the ways it can be used there are information extraction and natural language understanding systems, which are also the aims for which spaCy is employed in this work. Before talking about the functionalities offered by spaCy, the concepts at the base of natural language processing are briefly introduced. Natural language processing is an area of computer science and artificial intelligence concerned with the interactions in natural language between humans and computers which is intended to give computers the ability to understand and generate human language in the same way a human does. NLP is a very challenging field due to the complex nature of human language given by the fact that the same meaning can be expressed in several different ways, because words in a sentence can be arranged in infinite combinations. Another aspect that makes complex interpreting the meaning of a sentence is the meaning that a word assumes in a specific context. Obviously, when dealing with a natural language we have also to deal with the specific grammar of the considered language and all the syntactic rules that define the creation of sentences for that language. A general task of NLP can be divided into three phases: **speech recognition**, that is the transcription of spoken language into speech, **natural language understanding**, that requires the ability of the computer to comprehend the meaning of the language, and **natural language generation**, which consists in the generation of a correct sentence in natural language. The general pipeline followed in an NLP task is showed in Fig. 3.5.

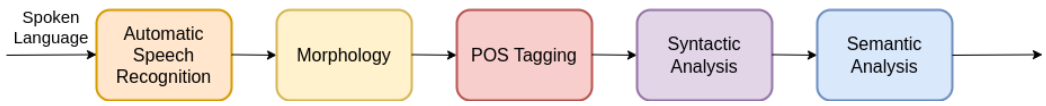


Figure 3.5. NLP Pipeline

Morphology The first block is the one that converts spoken language into text. The second block performs an important pre-processing step before the real NLP task: it verifies the morphological correctness of words and extracts information useful for the subsequent steps. In this phase the text is "normalized" with a tokenization process and a morphological analysis. Tokenization is used to segment

sentences into smaller units that can be easily analyzed; the challenge in this process is to segment units when there are spaces, punctuation or symbols near or inside a word. To address this problem, the model that processes the text is provided with a set of rules, specific to the analyzed language, that define how to recognize tokens. The morphology is the study of the way words are built up from smaller units called morphemes. The morphology analysis can be conducted in two ways: with stemming or with lemmatization. The goal of both is to reduce inflectional forms and derivationally related forms of a word to a common base form, the difference between them is how this is achieved. Stemming returns the "stem" of a word chopping off the end of words, lemmatization instead is a more fine process where the search of the common base of a word is made through the use of a vocabulary and morphological analysis, aiming to remove only inflectional endings and return the base form of a word, known as lemma.

POS Tagging The next step defined in the subsequent block is the Part-Of-Speech (POS) tagging. A POS tag is the morpho-syntactic category a word belongs to and POS tagging is the process of assigning to a word in a text a specific POS tag, based on its definition and its context. In modern linguistics there are nine part of speech: noun, verb, adverb, article, adjective, preposition, pronoun, conjunction and interjection. These are the main general categories, each one has several subcategories depending also on the language considered. The context in which a word is inserted is important in this task because it helps in assigning the correct POS tag, because each word can assume different POS tags depending on how is structured the sentence, and so the context is useful to remove ambiguities in the interpretation of a word. POS tagging is useful because of the large amount of information it retrieves in a text for each word and for its neighbours, giving information about the syntactic structure around a word. There are two ways to approach this task: the first one involves the use of hand-crafted rules to assign the tags, while the second one involves the use of stochastic models.

Syntactic Analysis After assigning to each word a specific tag, the next step is to execute a syntactic analysis, then moving from the analysis of single words to that of larger units. In this phase the aim is to comprehend the relational structures among words; this is made possible by knowing in advance the specific rules of the considered language that define what are the ways to combine words in order to generate correct sentences. Therefore in this phase is important to have a grammar, that is a declarative model that defines a language, and to parse the sentences, which means to decide if a sentence belongs to the language or not. The goal of a syntactic model is to identify constituents, i.e. span of words composing an atomic syntactic structure, which can be of different types (like noun phrases, verb phrases, prepositional phrases and so on), and recognize the correct syntactic structure, which is usually represented as a tree. Parsing is the process of mapping a sentence into one of its syntactic tree. From this definition we can understand that, when analyzing a sentence, there could be problems of ambiguity in its interpretation when it has more than one syntactic tree. There are two types of parsing: deterministic, which relies just on the predefined grammar and on an algorithm, and statistical,

which use grammar combined with probabilistic rules. The idea behind this last method is to compute the probability of each possible interpretation of a sentence and assign the most likely one. A possible way to represent a syntactic tree is using a dependency-based representation, which considers the dependency relations among words of the sentence. In these kind of trees, nodes are the words, instead of constituents, and they are connected by directed arc, labeled with the type of syntactic relationship. Each dependency relation is composed by: a label, which defines the nature of the relation, a head, that is the word governing the relation, and a dependent, that is the direct o indirect dependent of the relation.

Semantic Analysis The last block in the scheme is the semantic analysis, which aims at extracting the meaning of a text. This task is particularly difficult due to some reasons, like the ambiguity of language, the fact that sometimes information are implied and not explicitly defined, the dynamic evolution of language and others. However is important to understand correctly the meaning of a sentence to transfer knowledge between a human and a computer. Semantic analysis can be classified into two parts: lexical semantic analysis, which consists in understanding the meaning of each word individually, and compositional semantics analysis, which is the study of the meaning of the whole sentence/text. Once the meaning is extracted from a text, it must be represented in some way. In order to accomplish meaning representation, first is important to know what are the basic units of semantic systems: *entity*, refers to a particular unit or individual in specific like a person or a place; *concept*, a generalization of entities, it refers to a broad class of individual units; *relations*, establish relationships between entities and concepts; *predicate*, represent verb structures of the sentences. There are different approaches to represent meaning like first-order predicate logic, semantic nets, frames, rule-based architecture, case grammar, conceptual graph and conceptual dependency. Semantic analysis can be employed in different ways, two of the most used are text classification and text extraction. In the first one, the aim is to label a text with respect to a specific type of information that we want to gain from it. For example, in sentiment analysis the label of the text is related to the emotion that emerges from it, in intent classification a text is labeled with the intention behind it. In text extraction the goal is to obtain specific information from the text, like essential words that define the document or all the entities cited in the document.

Now that an introduction about NLP was made, it is possible to describe some of the features offered by spaCy, related to the concepts expressed above. This NLP library offers the possibility to load trained pipelines that can be used on a text to predict linguistic annotations. Each of these pipelines consists of multiple components that use a statistical model trained on labeled data. There are pipelines for a large variety of languages, which can be installed as individual Python modules, and the one used for this project is the pipeline for the english language. Typically these pipelines include a tagger, a lemmatizer, a parser and an entity recognizer. Each component processes the results of the previous module and passes the new processed document to the next module. These are the basic components inside a pipeline, however each pipeline can be structured in a particular way because

spaCy includes many others built-in modules that can be added and offers also the possibility to define a completely custom module. Each of these modules can contain a statistical model and trained weights or only make rule-based modifications to the document. The linguistic annotations returned by the spaCy pipelines are very specific and describe in detail the structure of a text, defining for each word its type (i.e. part of speech) and how it is related with the other words. The first step in the spaCy pipeline is the tokenization of the text, which segments it into words, punctuation and other single individual units, returning a *Doc* object of individual tokens. This process is strictly related to the rules specific to the considered language, which define how to split words from punctuation. After this operation, spaCy can parse and tag the tokens using a trained statistical model, which is used to make predictions of which tag most likely applies to a word in the given context. All the properties of a word obtained by this process are added as attributes to each token. At the end of this operation, we know for each word its POS-tag, its lemma, the dependency that relates it to the other words in the sentence, and other information. spaCy offers also a graphic tool through which is possible to visualize all these information in a graph where each word is connected to the others with edges specifying their relationship. The central data structures in spaCy are the *Language* class, the *Vocab* and the *Doc* object. The first one is used to process a text and convert it into a *Doc* object, which is the one that stores the sequence of tokens and all their annotations, as specified previously. Word vectors and lexical attributes are stored in the *Vocab*, which is a vocabulary shared by multiple documents. The strings stored in this structure are encoded using hash values, in order to save memory, avoiding storing multiple copies of data, and ensuring a single source of truth. As mentioned above, spaCy offers a wide variety of modules that ranges from the ones that extract specific information from the text to the ones that applies specific rules to obtain a document processed in a certain way. Among these functionalities, there is one that is worth to mention because is the one that is used mostly in this project, which is the *Matcher*. A Matcher is a rule-matching engine which helps in finding and extracting information from a *Doc* object based on match patterns describing the sequences we are interested in. The rules can refer to token annotations, like the text, the POS-tags or many others, and when such a rule is matched in a text there is the possibility to define a specific callback to act on it. When the sequence of tokens is relatively small, is better to use a token-based matching, otherwise, if the aim is to match large terminology list, there is a *PhraseMatcher* which accepts *Doc* objects as match patterns.

3.4 Gazebo

The simulation environment used in this project to test the architecture and carry on the experiments is Gazebo, which is an open-source 3D robotics simulator. It has the capability to simulate real-world physics with high fidelity, provides realistic rendering of environments with high-quality lighting, integrates a multitude of sensors that can be simulated through code. Gazebo offers the possibility to define and render your own robotic device in a simulated environment which is subject to real-world physics and which can be characterized by the presence of 3D-models, used

to better shape a real-world situation. Gazebo has its own set of models that can be easily imported into the simulated scene, among them we can find several robots, different kind of buildings and some other objects that can help in decorating the scene. It is also possible to create custom 3D-models with other programs or download them and import them into the scene. Gazebo is very useful when developing robotics application because it offers the possibility to test first the developed software in a simulated environment before installing it in the real hardware, allowing safe and controlled development. Gazebo is a standalone application, however it is very well integrated with ROS, allowing the user to use all the capabilities offered by ROS and test them in a safe and realistic simulated environment.

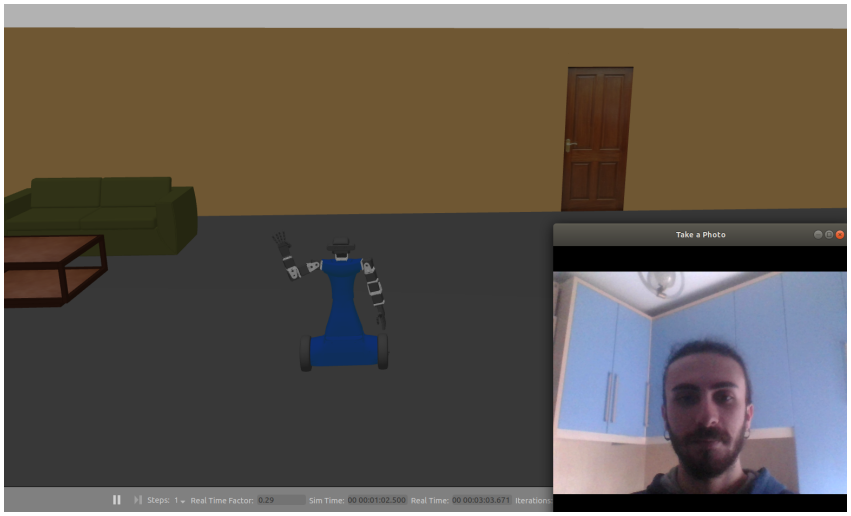
Chapter 4

Methodology

4.1 Architecture

In this chapter are described all the software components of the architecture designed for ALTER-EGO and how they are related. As defined in the previous chapters, the aim of this architecture is to create an autonomous behaviour for the robot in order to make it able to understand and respond to specific types of interactions. The architecture and its relative components are built upon a pre-existing infrastructure designed by the researchers of the Italian Institute of Technology that created the simulator of ALTER-EGO. Both the ALTER-EGO simulator and the architecture created for it were tested within a simulation environment. This chapter will focus on the modules of the new architecture, but there are two modules of the structure already present in the simulator that are worth to mention. The first one is the *Inverse Kinematic Gravity Compensation node* which defines the control scheme used to make the robot arms capable to reach a desired pose; this node takes as inputs the desired poses for the arm end-effectors, each one defined with respect to the relative shoulder, and converts them into the relative generalized coordinates for the joints. It is important to mention that this controller should have been able to realize a pose in normal real-life conditions, in particular with a normal value for the gravity, however during the experiments it was noticed that the controller suffered in executing commands with the normal value of gravity so its value was reduced in the simulation environment. The second node is the *LQR Control node*, which is in charge of the balancing and moving of the wheeled base of the robot. This node implements the control scheme explained in the ALTER-EGO section, exploiting the position of the arms of the robot to balance the whole structure. The node takes as inputs the link states, the readings from the IMU and also a desired commanded velocity, the latter when we want the robot to move; when such a case happens, the node computes the necessary torques for the two wheels to realize the desired motion. For what concerns the architecture built upon the simulator, the main nodes are five and are: *State Machine node*, *Text-To-Speech node*, *Speech-To-Text node*, *Gesture node* and *Event-Handler node*. For these nodes, except for the State Machine node, was implemented a *class* object with all the functionalities that ensure their correct behaviour.

(a)



(b)

Figure 4.1. Screens of the scene taken during the simulations. **(a)** Simulation of a user approaching the robot to interact with it **(b)** Representation of an interaction carried out via microphone in which the user greets the robot, which responds in turn

The relations between these new modules and the pre-existing ones are showed in Fig. 4.2. In particular, each arrow in the scheme represents only the direction along which information are exchanged between the nodes connected by it through topics and not the number of topics associated to that nodes, meaning that between them there may be multiple topics that exchange information, according to the scheme depicted in Fig. 3.1, but all in the direction specified by the arrow. The only case that breaks away from the scheme publisher/subscriber is the one involving the State Machine node and the Text-To-Speech node, in fact in this case the two nodes are not communicating using a topic, but instead through the call to a ROS service which returns a *Response* to the node that called it when the execution of the requested service is finished (this explains the absence of an arrow between them). The nodes colored in purple are the ones developed in this project, the nodes colored

in green are the ones already present in the ALTER-EGO simulator architecture, while the nodes colored in yellow are the ones added to the new architecture to build the navigation functionalities for the robot. Here a brief explanation of how the architecture works. The Speech-To-Text node notices the Event Handler node that an event was found when it matches a specific request made by the user. Then the Event Handler node forwards this information triggering the State Machine node. At this point, the state machine will transition all the states related to the event found, eventually requesting the functionalities of the Gesture node or the Text-to-Speech node through which, respectively, the robot executes a gesture or responds vocally to the user. In the next sections, all nodes are explained in detail, describing the way in which each of them is implemented and how they work. The following explanation follows a bottom-up scheme, starting with the nodes that implement autonomous functionalities invoked in specific cases, i.e. the Gesture node and the Text-To-Speech node, and concluding with the central node that handles all operations for each event by invoking the functionalities of the previous two nodes, i.e. the State Machine node, passing through the node in charge of triggering the above operations, i.e. the Speech-To-Text node.

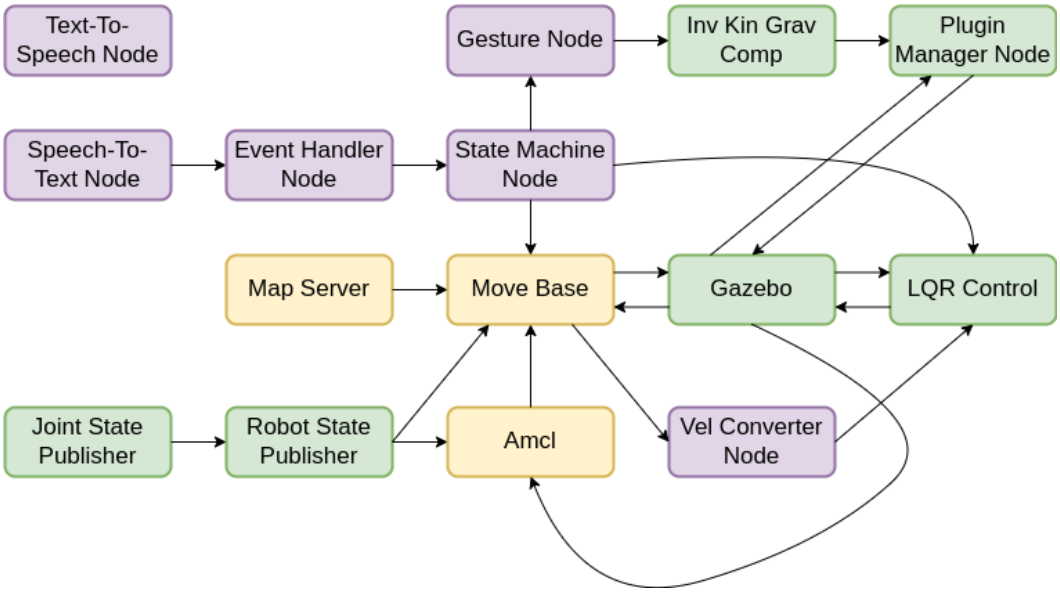


Figure 4.2. Block scheme of the architecture of the robot. Each directed edge represents how the information between node is exchanged

4.2 Gesture Node

The Gesture node is in charge of sending the right pose coordinates for the execution of a given gesture. The most interesting topic concerning this node is the formalism used to create the gestures. Before dealing with this part in more detail, the way this node works is first briefly defined. The *gestureManager* class, which implements the functionalities for this node, is subscribed to a dedicated topic where gesture commands are sent. When the subscriber notices a message in this topic, the

function that manages the sending of the poses is triggered. This function first checks if the received gesture command is included in the list of the available gestures or not. In positive case, the file in which are stored the information for the execution of the movement is analyzed and the data contained in it are used to reproduce the movement.

Gesture Formalism The most noticeable aspect of this implementation is the way in which gestures are defined. The principle behind this is to make the creation and representation of movements easily extensible. To achieve this, a simple and flexible formalism for creating movements was defined. Each executable gesture is coded into a JSON file, which provide the ability to define several fields, each containing data of a different nature. The formal structure of these files is showed below.

```
{
    "phases": [
        {
            "rightArmPose_1": {"type": "vector"},
            "leftArmPose_1": {"type": "vector"},
            "executionTime_1": {"type": "int"}
        },
        {
            "rightArmPose_2": {"type": "vector"},
            "leftArmPose_2": {"type": "vector"},
            "executionTime_2": {"type": "int"}
        },
                .
                .
                .
        {
            "rightArmPose_N": {"type": "vector"},
            "leftArmPose_N": {"type": "vector"},
            "executionTime_N": {"type": "int"}
        }
    ]
   "repeat": {"type": "int"}
}
```

Each JSON file has two fields, *phases* and *repeat*. The first one is the list of end-effector poses that the robot should execute to complete the gesture. Each element of this list is composed by three sub-fields: *rightArmPose*, *leftArmPose* and *executionTime*. The first two define the pose to reach in that phase for the right and left arm respectively and each of them is a vector of seven elements:

$$pose = \begin{bmatrix} pos_x \\ pos_y \\ pos_z \\ orientation_x \\ orientation_y \\ orientation_z \\ orientation_w \end{bmatrix} \tag{4.1}$$

where the first three elements of this vector define the desired position of the end-effector while the other four define the desired orientation expressed with a quaternion, with both position and orientation defined with respect to the relative shoulder. The *executionTime* field specifies how much time is assigned in that phase to reach the specified poses; this parameter is used to time the gesture's trajectory. Defining a gesture in this way, becomes possible to break the movement into a sequence of fixed poses, each of which should be reached in the given amount of time. The whole gesture is achieved by the sequential execution of these poses. The execution times assigned for the gesture implemented in this work were obtained running several tests, noticing how much time was required by the robot to reach the commanded poses. The total time of execution of a whole gesture is obtained by the sum of the execution times of the single phases. The field *repeat* specifies how many times the gesture must be executed. The definition of a general template makes very easy adding new executable gestures, because the creation of a new movement requires only to define a new JSON file and fill the relative fields, which means, find for the movement you want to model a finite sequence of key positions to traverse and for each of them define a time limit within which the arm has to reach it. Moreover, this kind of template allows also to expand the characterization of the gestures by giving the possibility of adding new features by just inserting new fields in the JSON file. Obviously, in this case, the function that manages the gesture must be updated to cope with the new information added for the execution of the movement, but without changing the main algorithm.

Gesture Execution A specific function is defined for reading the JSON file for the requested gesture and storing all the information into a specific data structure, in which the poses specified in all the phases of the gesture are saved into a single vector, one per arm. At this point the main function of the node enters into a loop in which, at each beginning, selects the pose of the current phase for both arms and publishes them into the relative topics for a number of seconds specified by the *executionTime* of that phase. When this window of times elapses, a flag notices the end of that phase and the loop passes to the execution of the next phase and so to the next poses. The poses are published into the topics connected to the Inverse Kinematics Gravity Compensation node, in this way this node is triggered and for each pose computes the relative joint configuration. For the way in which the Inverse Kinematics Gravity Compensation node is implemented, it is necessary to publish constantly the desired poses for the arms, otherwise the movement will not be realized and the pose not achieved. The described loop is executed a number of times equal to the value specified in the field *repeat*. An

example of execution is shown in Fig. 4.3, where in particular the movement related to the greeting is performed, which in total has three phases. It can be seen that, from the combination of a sequence of independent poses, a complex movement has been created with a specific execution time.

(a) **(b)** **(c)**

Figure 4.3. Execution of the greeting gesture. Each figure shows the execution of each of the three steps that define this movement

Until now was described how gestures are implemented and the process that controls their execution. This was the solution that worked correctly with the simulator. However, two other solutions have been tried to implement this functionality that did not work correctly but are worth to mention. In the first attempt was implemented a callback function which first retrieved from a gazebo topic the current information about the end-effector link, in particular its orientation with respect to the gazebo world frame, and then converted this orientation with respect to the relative shoulder reference frame, using the conversion formula for quaternions: ${}^{SH}Q_{EE} = {}^{W}Q'_{SH}\,{}^{W}Q_{EE}\,{}^{W}Q_{SH}$, where ${}^{SH}Q_{EE}$ is the orientation of the end-effector expressed in the shoulder reference frame, ${}^{W}Q_{SH}$ and ${}^{W}Q'_{SH}$ are respectively the quaternions expressing the orientation of the shoulder with respect to the gazebo world frame and its inverse, and ${}^{W}Q_{EE}$ is the orientation of the end-effector link with respect to gazebo world frame. It is important to mention that in this conversion i considered fixed the orientation of the shoulder with respect to the gazebo world frame, taking its value at rest. This because the desired pose that is sent to the controller is defined with respect to the shoulder frame at rest. At this point, in the main function, instead of publishing the desired pose as explained previously, was computed the difference between the desired commanded orientation and the current orientation of the end-effector link, both expressed with respect to the shoulder frame. When this difference was below a specific threshold the phase was considered concluded. This attempt did not work correctly due to the unpredictable behaviour of the arm when a pose is requested. In fact, even poses that differed very little from each other, resulted in way too much different movements of the arm, with different rotations of the joints and thus even the rotation of the shoulder. Because of this, this algorithm worked only in the cases in which the movement was realized without rotating the shoulder, while in others failed. The second solution instead

is a slight different version of the main function described previously. The main difference lies in the way the data structure of the pose is filled: in this version the poses specified in the JSON file of the gesture were interpolated, thus adding a fixed number n of intermediate poses between them. Here an example to make this clear: consider a gesture file with its *phases* field with 3 elements, meaning that the gesture is composed by a sequence of three different poses; instead of considering only these three, when filling the data structure, an interpolation function added n intermediate points between each couple of phases. Also the loop of the main function was a bit different in this case: instead of publishing each pose for a finite window of time, each pose was published only once. This last choice was made because, given the increased number of poses after the interpolation, giving a certain number of seconds to each of them would result into a total execution time too long. This solution did not work because publishing just once was not enough to allow the controller to reach the desired pose. However, even if these approaches were unsuccessful, they helped to design the final correct solution.

4.3 Speech-To-Text Node

To make interactions with the robot as natural as possible, ALTER-EGO is equipped with the capability to receive and understand spoken language through a specific module: the Speech-To-Text node has the task of analyzing the spoken interaction with an human interlocutor. To realize this functionality the robot must be able to listen, transform received audio into written text and have text analysis capabilities. In this section is explained how these mechanisms are integrated. In particular, is important to mention two tools used in the implementation of this node, which are the already mentioned library spaCy and the *SpeechRecognition* library [20], a software that offers classes and functions to perform speech recognition. When the node is launched it defines two ROS parameters that will be used by the node to determine the type of interaction in progress, in particular the first parameter is *isSpeaking*, set to *True* when the robot is speaking and is used to prevent the robot from listening while it speaks, and the second one is *isInteracting*, set to *True* when the robot is involved in a complex interaction with the interlocutor. The node has a main function called *listen()* which is repeatedly executed in a loop. This function starts by listening from the source, i.e. a microphone in my case, any possible audio input in a finite time interval. During this interval, a function from the SpeechRecognition library is in charge of recording a single phrase from the source into an *AudioData* object, which represents mono audio data, with raw audio stored as a sequence of bytes representing audio samples. This is done by waiting until the audio has an energy above a specified threshold, meaning that the user has started speaking, and recording until it encounters a specific amount of seconds of non-speaking or there is no more audio input. At this point, if the robot is not speaking, the recorded audio is analyzed. What is needed now is to convert the audio data into a text; this is done by using another function of the SpeechRecognition library, which takes as input the recorded audio and performs on it speech recognition using the Google Speech Recognition API, given a specified language. For this project, the language used is english. This function returns

the most likely transcription of the audio. In the case in which there is silence, so no sentence pronounced, the transcription is just an empty string. Now the transcription is fed into an NLP model provided by spaCy and processed as described in the section about this library, returning a *Doc* object which is then passed to a *Matcher*. As defined in the previous chapter, a *Matcher* is a spaCy class that has the capability of finding specific patterns inside a text returning the matched sequence of words. In this project was designed a set of patterns, listed below, for the interactions implemented.

- **Greeting pattern** - The pattern to match is composed by just a single word in a set of possible words. In particular the *Matcher* searches for a "*Hi*" or a "*Hello*" in the text. This pattern is defined for a simple greeting.
- **Stop pattern** - Same as before, but the word searched is "*Stop*". This pattern is defined to allow the user to stop the interaction.
- **Event-info pattern** - In this case the pattern to match is a bit longer: it is composed by a sequence of four words matched based on their dependency tags and their texts. The first two words are optional, meaning that they are not strictly necessary to return a match, while the other two are the ones that trigger the match and they are "*next*" and "*event*". Examples of sentences which result in a match are "What is *today's next event*?", "What is *Sunday's next event*?" or just "What is the *next event*?". The last case, i.e. the one in which a day is not specified, is considered incomplete, while the others are considered complete. As the examples suggest, this pattern is defined to give to the user the possibility to ask information about the scheduled events.
- **Navigation pattern** - In this case were defined two different patterns, based on combinations of POS tags and specific words, which result in the same kind of match. The first pattern is composed by a sequence of five words: the first one must be a verb, in particular "*take*" or "*lead*", the second one is optional and should be a pronoun, the third one must be the word "*to*", while the last two must be, respectively, a determiner and a noun. Examples of sentences that result in a match for this pattern are "Can you *take me to the bathroom*?" or "*Lead us to the wardrobe*". The second pattern is similar to the previous one. It is four words long: the first one must be "*where*", followed by the verb "*is*" and the last two as before, so a determiner followed by a noun. Examples that result in a match for this pattern are "*Where is the bathroom?* or "Can you show us *where is the lounge*?". These patterns are defined to understand when the user is asking to the robot information about a specific place in the environment.
- **User Reply Event-Info patterns** - These patterns are defined to match specific responses during an interaction started after that an incomplete event-info match has been found. The first pattern matches exactly one word among the set composed by "*today*" and "*tomorrow*". The second pattern matches any word that is not equal to the previous two. These patterns are defined to understand if the user responds correctly or not to the question made by the robot during the interaction.

After that the *Matcher* has analyzed the transcription, if a sequence of words matches one of the patterns described, then an "event" is found, meaning that a specific request, among those the robot is able to distinguish, was made by the user. At this point the node sends a message with the information about the event into a specific topic. In particular, the Speech-To-Text node has two publishers, one that publishes into the topic connected to the Event Handler node, used to notice an event triggered by a match belonging to the first four patterns defined above, and the other one connected to a topic, monitored by a specific state of the state machine, in which are published messages related to matches regarding the last patterns defined in the list, i.e. the User Reply patterns. The second publisher is used when the ROS parameter *isInteracting* is set to *True*. The messages sent by both publishers are the same. In particular, the message is a custom one, defined to include all the necessary information of the event found, like the event ID, which identifies the specific request, the words matched by the pattern and their linguistic properties. When this message is sent, the node finishes its work and the *listen()* function starts again.

4.4 Event Handler Node

The Event Handler node is a very simple node, inserted to better represent both from a practical and conceptual point of view the way in which events are triggered. This node is subscribed to the topic used to send messages notifying an event sent by the Speech-To-Text node. When such a message arrives, the node is triggered and forwards the same message into another topic, which is connected to the State Machine node, which will analyze the type of event.

4.5 Text-To-Speech Node

The Text-To-Speech node is the one that allows the robot to speak. This functionality is implemented through a ROS service, which is a function that can be called by any node when necessary. ROS services are defined by *srv* files, which contain a *Request* message and a *Response* message; in particular, the *Request* is sent to the service when it is called, while the *Response* is returned by the service when its operations are finished. For this node has been defined a custom *srv* file, which contains four fields, three for the request and one for the response. The request fields are: *event_id*, which is a string containing the ID of the current event, *req_type*, which is a string that can assume three possible values, *REQ_CORRECT*, *REQ_INCORRECT* and *REQ_INCOMPLETE*, that specify respectively, if the request made by the user is correct, incorrect or incomplete (this control is made by a specific state of the State Machine node), and the last field is *req_spec* which is a string that can be used to add additional information to the request. The response field is just an unsigned int equal to 0 if the operations were completed correctly, 1 otherwise. When the service is called, first it reads the request fields from which it retrieves the ID of the event to which the robot should respond. This ID is used to select the correct function that replies to the current request. In fact each event for which is defined a vocal response has a specific function that man-

ages the way in which the robot replies; in particular these events are three and are the *Greeting* event, triggered when someone greets the robot, the *Event-Info* event, triggered when the user asks information about the next scheduled event, and the *Navigation* event, triggered when the user asks to be taken somewhere in the environment. All the replies defined for the various cases of the possible interactions are stored inside a dictionary where the correct answer is retrieved using as keys the *event_id*, *req_type* and *req_spec* depending on the case. The function that replies to the Greeting event is very simple, it just selects from the dictionary the vector containing the replies for this event and selects one of them randomly. The function that manages the reply for the Event-Info event is slightly more complex. First it checks if the *req_type* is *REQ_CORRECT* or not: if not, the reply is just taken from the vector of replies related to one of the other two possible cases, i.e. *REQ_INCORRECT* or *REQ_INCOMPLETE*; instead if *req_type* is *REQ_CORRECT*, the replies are stored inside another dictionary related to the key *REQ_CORRECT*, because in this case there are two possibilities, if the next event is found or not. To better understand when these two cases can happen, it is necessary first to describe how the information related to the events are stored. In a *txt* file is listed a sequence of scheduled events each one with the following information: day of the event, name of the event and event time. The day of the event is limited to assume only two values, i.e. today or tomorrow, for the way in which the interaction was imagined for this case: is assumed that the robot could be employed for a convention of one or more days and that a specific guest could have interest in knowing about the next event of the day or at least the first event of the next day. Explained this, it is clear to understand when the event is found or not: if the user asks for the next event of tomorrow, the robot replies by naming the first event of the next day, instead if the user asks for the next event of today, if the time at which the request was made exceeds the time of the last event of the day, then the robot replies with the sentence defined when no event is found, otherwise the robot replies naming all the information of the next event of the day based on the current time. For what concerns the replies for the Navigation event, they are three: one for when the request of the user is correct, namely when the place requested is concretely inside the environment, one for when the request is not correct, that is when he asks for a place that is not in the environment, and one for when the robot reaches the desired destination. Until now was described the way in which the specific reply was selected for each possible case, however this reply is still a text. To convert this text into a spoken sentence the module Google Text-To-Speech (gTTS) was used: it takes as input the text to be read, then it writes the audio bytes to a file-like object which is then reproduced using a function of the library *pygame*.

4.6 State Machine Node

This node implements the state machine which managees the behaviours of the robot. It was implemented using the library *smach*, which is a Python library designed for the creation of state machines. The whole structure of the state machine is showed in Fig. 4.4.

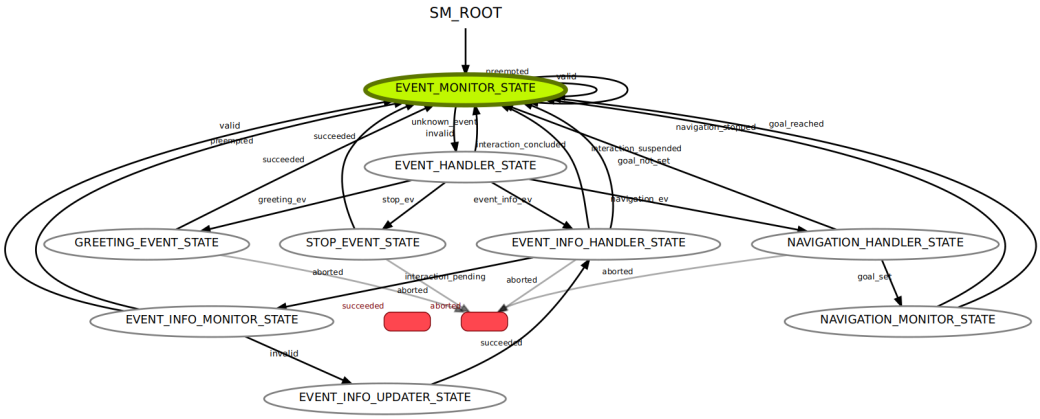


Figure 4.4. Structure of the state machine. The circles are the states while each edge represents a possible outcome of the relative state

In the picture, each circle represents a state while the edges coming out of each node represent the possible outcomes of that state. In the next, the behaviour of the state machine is explained by grouping the states that are passed through when a given event occurs.

- **Event Monitor & Event Handler States** - The root of the state machine is the *Event Monitor State*, its purpose is to monitor the topic in which messages notifying the detection of an event are sent by the Event Handler node. When a message arrives, the state transitions to the next one that is the *Event Handler State*. This state is in charge of retrieving, from the message arrived, the ID of the detected event. This ID is then used to transition to the next set of states designed to execute the right actions in response to the event.

- **Callback States** - Two are the states belonging to this group: the *Greeting Event State* and the *Stop Event State*, the first one becomes active when the user greets the robot, the second one when the user sends a stop command to the robot. These states are implemented in order to execute a callback when active, in particular a callback which require just the execution of a gesture. Their implementations are almost similar, so here are described the common operations of them while pointing out the differences. First the callback retrieves the event ID from the message received, then a publisher is used to send a message to the Gesture node, specifying which gesture should be executed. The difference between the two states is that in the *Greeting Event State* there is also a call to the ROS service offered by the Text-To-Speech node, in order to integrate the greeting gesture with a voice response.

- **Event-Info States** - This is a set of three states in which the state machine enters when the user asks information about the next event. In this case, the first state after the *Event Handler State* is the *Event-Info Handler State*. This state is responsible for figuring out whether the request made by the user is correct, incorrect or incomplete. To do so it uses the matched pattern in the user's sentence to fill a dictionary in which the fields are associated to

the necessary information that should have been specified in the request. In particular, for the way in which is modeled this interaction, the information needed to consider the request complete is the specific day whose next event the user wants to know about. As specified in the section about the Speech-To-Text node, this day must be *today* or *tomorrow*. Then, the dictionary is used to classify the request: if the day specified is exactly one between *today* and *tomorrow*, the request is considered correct, while if the day is specified but it is not one of the two mentioned above, the request is considered wrong. After this check, the relative flags are set and a *Request* to the Text-To-Speech node service is sent, that will be in charge of reply with the correct answer for the case. For these two situations, after the robot reply, the interaction is considered concluded and the state machine goes back to the *Event Monitor State*, waiting for new events. Instead, if in the request the user asks only for the *next event*, which is the shortest pattern that triggers an event for this case, without specifying the day, the node considers this request incomplete. Even in this situation, a *Request* is sent to the text-to-speech service, that will select the correct reply, that in this case is a question to the user. In particular, the robot asks to specify the day the user is interested in regarding the next event. At this point, the state machine transitions to the *Event-Info Monitor State*, which monitors the topic in which the user reply to the question is sent. When a message arrives in this topic, the state machine transitions to the *Event-Info Updater State* which takes the reply of the user and uses it to complete the original request. Then the state machine returns to the *Event-Info Handler State*, this time with a request complete of all the necessary information and the operations described above are executed again, thus checking if the date specified is admissible or not.

- **Navigation States** - These states are responsible for initiating and controlling the robot's navigation through the environment to a specific destination requested by the user. When a navigation event is found, the *Event Handler State* passes the control to the *Navigation Handler State*. If the destination specified is not one those available, the robot replies to the user saying that the specific goal is not present in the environment and asking to choose one of those available. Instead, if the location requested is one of those available, the state retrieves the relative coordinates of the goal from a dictionary containing the information about all the reachable destination, then the goal is sent to a server, which uses the goal information to create a specific message that is sent to the topic connected to the software that manages the navigation, which is *move_base*. This software provides the functionalities to perform path planning within an environment given a start and end location, using information from sensors to ensure safe navigation. In this project, the sensor used to detect obstacles is a LIDAR laser scanner. When the goal is sent to *move_base*, the software computes the shortest and safest path towards the destination and then it calculates and sends to the robot the speeds, defined within certain limits, of the movements to be performed for the duration of the navigation. When the goal is set, the state machine transitions to the *Navigation Monitor State*. As the name suggests, this state is in charge of

monitoring the state of the navigation; in particular, it controls if the goal is reached, if the navigation is stopped or if there is an obstacle along the path. These controls are realized using subscribers to the topics that allow to gather information about those situations, in particular, for the first case is used a topic of *move_base* which returns the goal status, for the second case is used the same topic checked to monitor events and for the last case is used the topic that returns scan information from the laser. When the message with the goal status "reached" arrives, the robot notifies that the destination is reached with the text-to-speech service and the state ends. During the navigation, the user can still talk to the robot, however is available only the stop command that can be used to stop the navigation. When this happens, the robots stops on the spot and returns available to a new request. If, while the robot is following the designed trajectory, the scan returns a range value below a specific threshold, the navigation is temporarily stopped, meaning that the robot found an obstacle, like for example a walking person. When the obstacle goes out of its range of action, the navigation is resumed.

4.7 Navigation

This section discusses the aspects related to robot navigation in the scene. As defined in the previous sections, to make the robot aware of its location in space and be able to move from its spot to a specific destination, the software *move_base* was used and integrated with the architecture, as shown in Fig. 4.2 (in particular in that scheme the yellow nodes are the ones related with this argument). In order to move the robot, *move_base* needs to have a map of the environment and needs to be able to localize it inside the scene. These information are provided by the two nodes *Amcl* and *Map Server*. The first one is used to localize the robot in the environment, it takes as input the initial position of the robot and keeps updating it during the simulation using the information about the position and orientation of the reference frames of the objects in the scene received from Gazebo. The second node provides to *move_base* the map of the environment, which contains information about the fixed obstacle in the scene. The map was obtained using the laser scanner of the robot, showed in Fig. 4.5, to detect the fixed obstacles and saving these information into a map through the software *gmapping*. Specifically, the map reproduced, shown in Fig. 4.6, is that of Hadrian's temple in Rome and its simulated counterpart built in Gazebo is showed in Fig. 4.7. At the center of the building has been recreated, even if with fewer elements with respect to its real counterpart, the colonnade of the temple bordering the conference area, where there is a small stage reserved for the speaker and a group of chairs reserved for the audience. On the left side of the structure there are two other areas: in the lower left corner is a small lounge designated as a relaxation area for guests, while right in front of it, at the top, there are two rooms (not clearly visible in the map due to the fact that, since they are enclosed by doors, the robot cannot enter inside to map them but they are depicted in Fig. 4.7) designated as a bathroom and checkroom. The latter three areas defined, so the lounge, the bathroom and the wardrobe, are the ones selected to test the navigation of the robot inside the building.

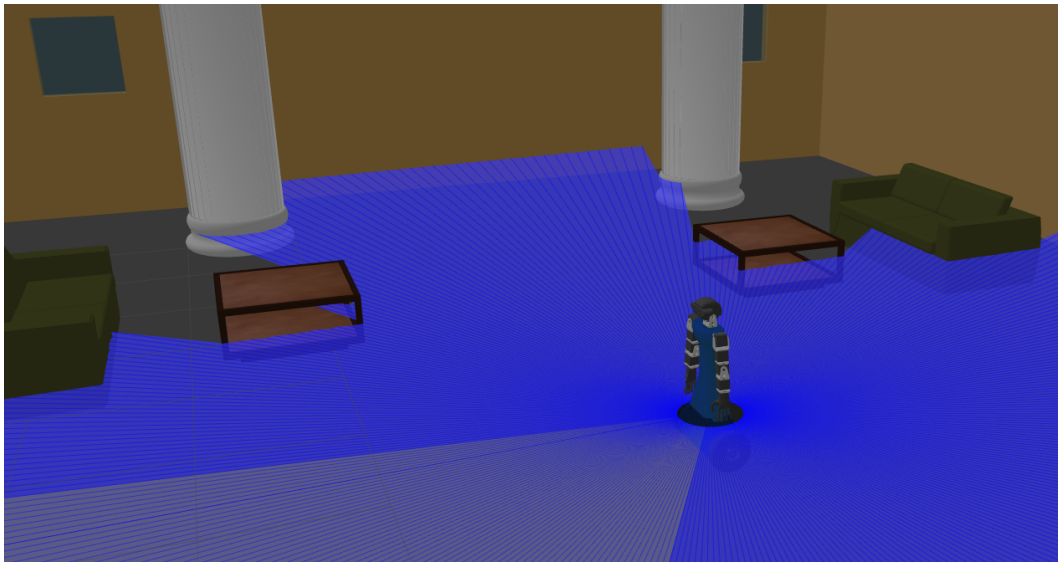


Figure 4.5. LIDAR laser mounted on ALTER-EGO

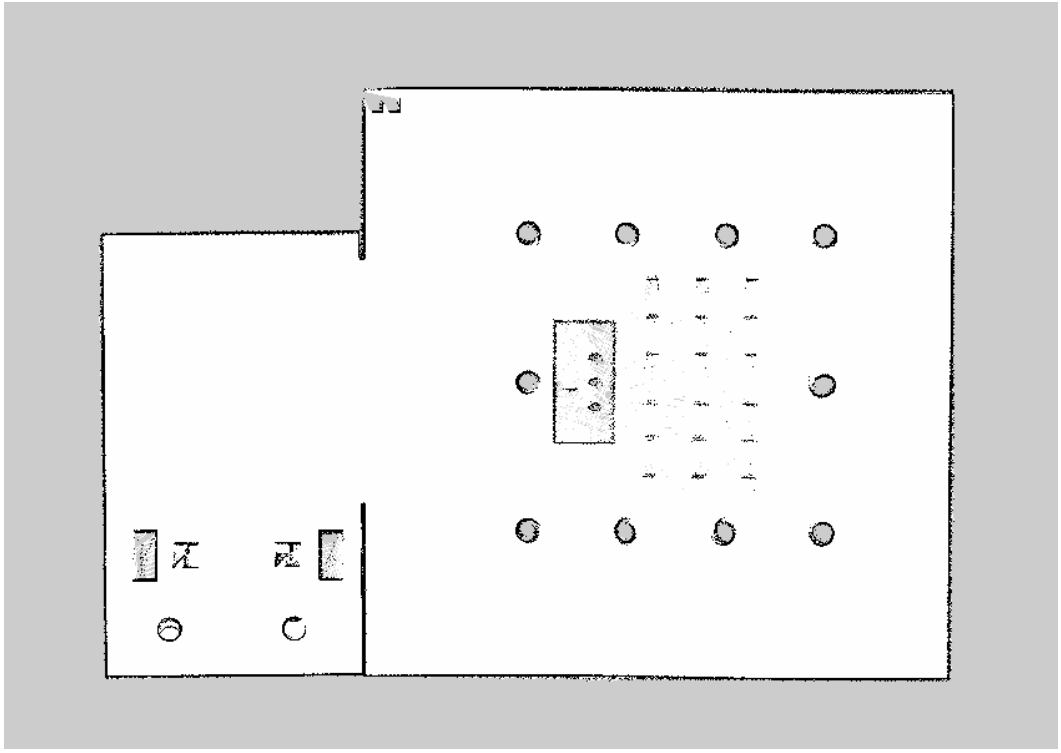


Figure 4.6. Map of the environment resembling Hadrian's temple in Rome

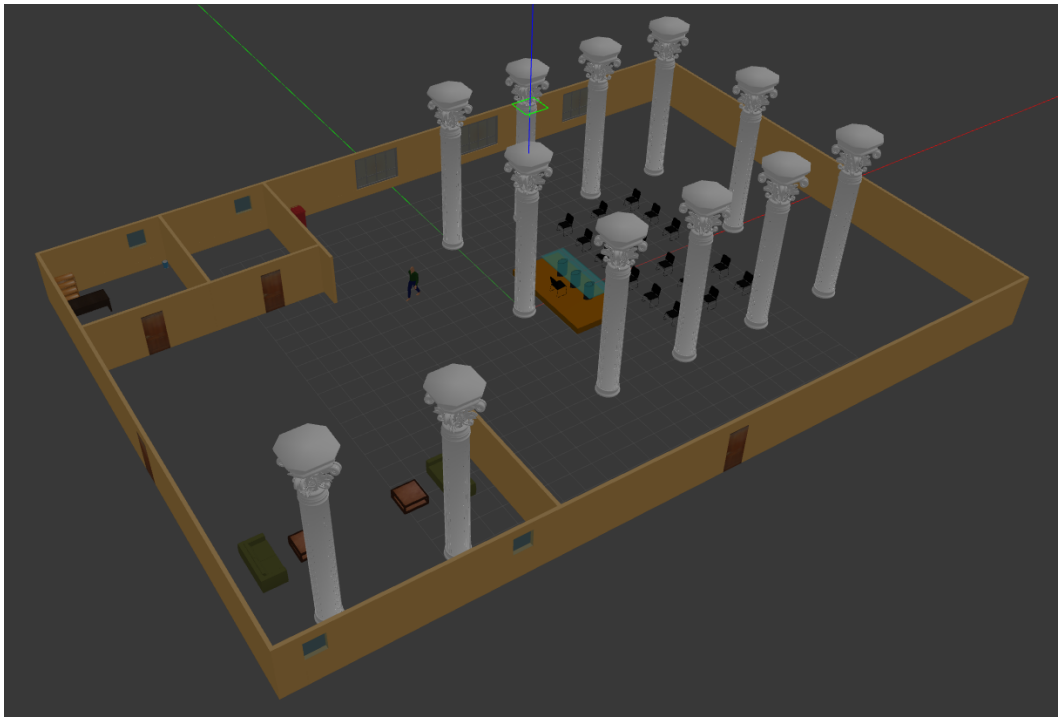


Figure 4.7. The environment built in Gazebo

Having the information about the position of the robot and the obstacles in the environment, *move_base* is able to move the robot from its position to a desired destination, but to complete this operation other parameters are needed. The navigation stack uses two costmaps to store information about obstacles in the scene. A costmap is a grid map where each cell is assigned a specific value or cost; higher costs indicate a smaller distance between the robot and an obstacle. One costmap is used for global planning, i.e. for long-term plans over the entire environment, while the other is used for local planning and obstacle avoidance. There are some parameters shared by both costmaps, while there are others that are specific for each of them. In the common parameters we find the list of sensor topics from which the two costmaps should receive updates, the *obstacle_range* parameter which determines the maximum range sensor reading that will result in an obstacle being put into the costmap, the *raytrace_range* parameter that specifies the range to which freespace is raytraced given a sensor reading, the *footprint* of the robot and the *inflation_radius* which defines the maximum distance from obstacles at which a cost should be computed. The parameters that need to be specified individually for each costmap are: the reference frame to which the costmap should refer, the reference frame the costmap should reference for the base of the robot and the frequency at which the costmap is updated. These are the minimum necessary parameters to set, however there are many other parameters that can be used to better define your navigation needs. The last component of the navigation stack is a local planner, responsible for computing velocity commands to send to the mobile base of the robot given a high-level plan. The parameters of the planner are used to set the velocity and acceleration limits, both linear and angular, for the movements of

the robot. Using the information provided by all these components, when a goal is specified, *move_base* computes the shortest and safest trajectory to reach it from the current position. Below examples of these trajectories, one for each goal defined in the scene, i.e. the bathroom, the lounge and the wardrobe.

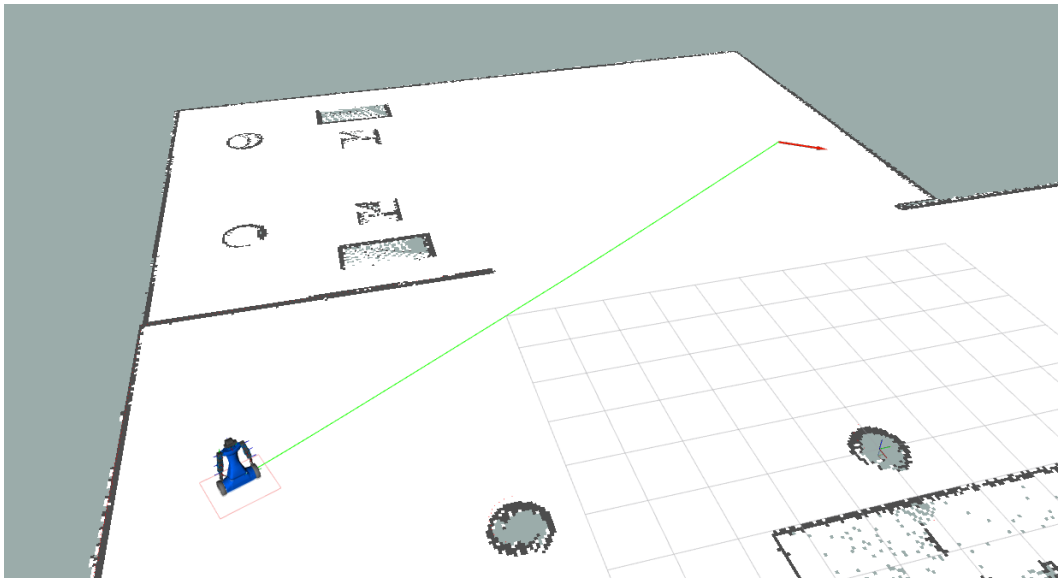


Figure 4.8. Trajectory computed to reach the bathroom from the current position

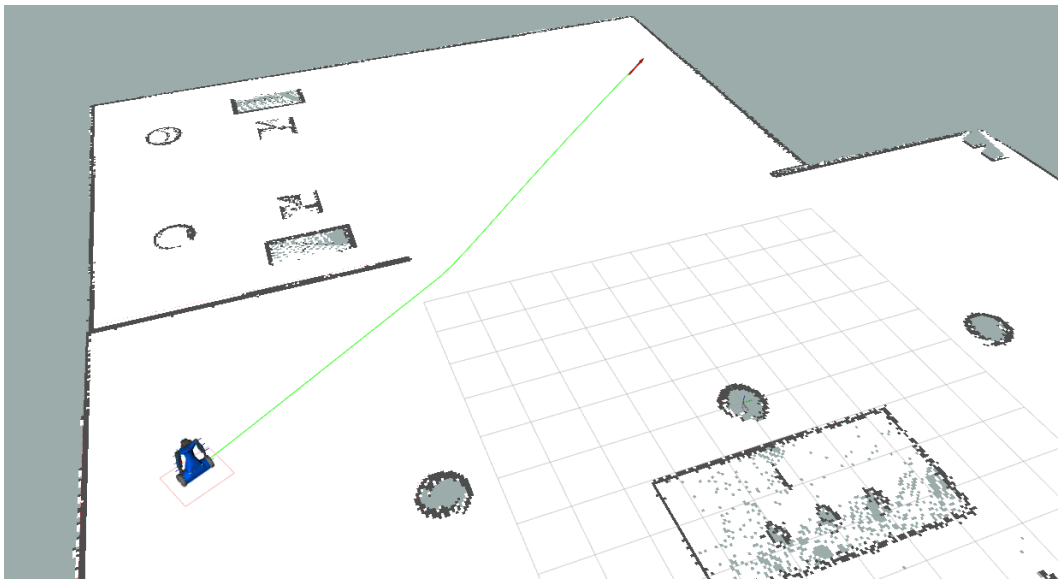


Figure 4.9. Trajectory computed to reach the wardrobe from the current position

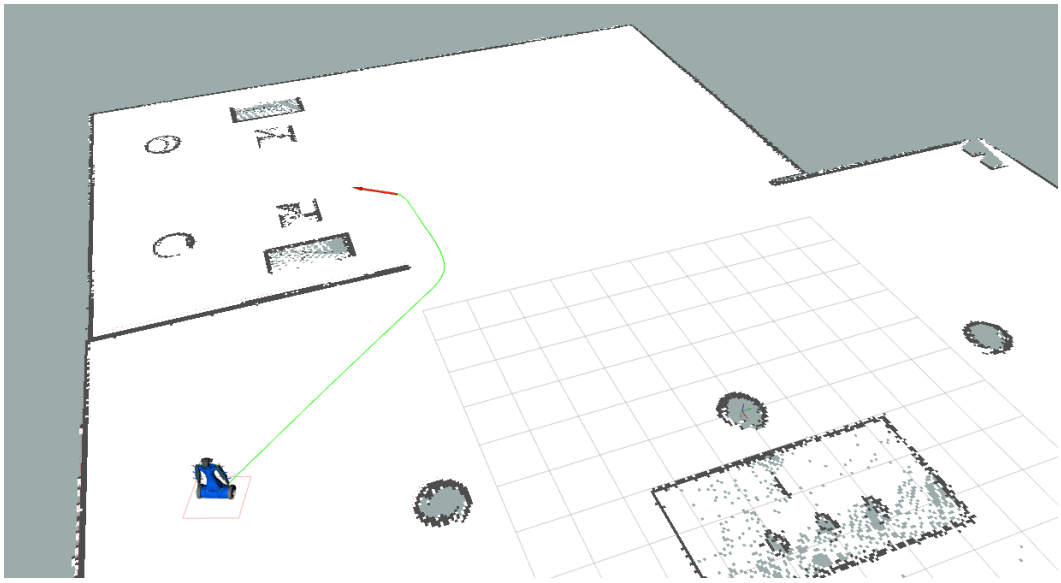


Figure 4.10. Trajectory computed to reach the lounge from the current position

Chapter 5

Results

This chapter reports the results of the tests performed on the software architecture to evaluate its efficiency. In particular, data are given regarding the execution times of some of the main functions of the architecture's modules. Profiling is an important aspect of software programming, through it is possible to determine the portions of code that are more time-consuming, helping to understand which functions need to be optimized in order to obtain a faster execution of the program. This aspect is very important in this project because it is necessary to have low computational costs due to the fact that the robot must perform all operations onboard. Here are reported the results obtained by profiling the nodes of the architecture during the execution of the three possible interactions designed for this project, which are: a simple greeting, the request for information and the request for accompaniment to a specific place. For each interaction the data are showed in a table with the relative fields, here explained. *Function* is the name of the function within which the instruction being analyzed is present, *Instruction* is the name of the above mentioned instruction, *Hits* is the number of time the instruction is called, *Time* is the amount of seconds of the running process used by the instruction, *Time per hit* is the time needed for each call of the instruction and *%* is the same of *Time* but expressed as a percentage of the total time. These parameters are used to evaluate the State Machine node, the Speech-To-Text node and the Text-To-Speech node, profiled using *pprofile*. Instead the Gesture node was profiled with *gprof* and the relative fields are: *Function*, name of the analyzed function, *%*, the percentage of the total execution time the program spent in this function, *cumulative seconds*, the cumulative total number of seconds the computer spent executing this function, plus the time spent in all the functions above this one, *self seconds*, the number of seconds accounted for by this function alone, *self ms/call*, represents the average number of milliseconds spent in this function per call and *total ms/call* that represents the average number of milliseconds spent in this function and its descendants per call. The following tables are intended to focus attention on specific functions called by nodes during their execution, particularly those functions most involved in the implementation of the response to the given event. The first interaction analyzed is the simple greeting and the data relative to its execution are showed in Tab. 5.1. The *greeting_cb* is the function of the State Machine node that defines which operations to do in this case; it sends very quickly the message containing

the type of movement to be executed via the *gesture_command_pub.publish* instruction while most of the execution time is spent by the function *setTTSRequest*, which is in charge of requesting the service of the Text-To-Speech node through the instruction *reply*. In the Speech-To-Text section we can see that the instantiation of the class *sttManager*, in which the functionalities of this node are implemented, requires a good portion of the execution time; in particular, this time is mostly spent loading the nlp model from spaCy, operation performed only when the node is launched, so it does not impact the execution time of the main function *listen*, which instead requires about 4.5 seconds to be executed at each call, value in line with the waiting time given to the module to pick up a spoken sentence from the user. The values regarding the *sttManager* and *spacy.load* instructions are quite the same for each interaction tested, so they will not be reported in the next tables. The recognition of the speech, performed via the instruction *r.recognize_google*, is quite fast, requiring about 1 second to convert the audio into a text. For what concerns the Text-To-Speech node, we can see that the function involved in this interaction, i.e. the *greeting_reply*, uses about 1.24 seconds, time mostly spent by the function *speak*, which converts the written text chosen as a response into audio. To conclude the analysis of this first case, the main function of the Gesture node, i.e. the *moveArms* function, is quite efficient, with an almost immediate readout of the gesture data via the *readJson* function.

Greeting Interaction					
Node			State Machine		
Function	**Instruction**	**Hits**	**Time**	**Time per hit**	**%**
greeting_cb()	gesture_command_pub.publish()	1	0.00269699	0.00269699	0.01%
greeting_cb()	setTTSRequest()	1	5.13672	5.13672	14.91%
setTTSRequest()	reply(req)	1	5.06899	5.06899	14.71%
Node			Speech-To-Text		
Function	**Instruction**	**Hits**	**Time**	**Time per hit**	**%**
main()	sttManager()	1	42.9461	42.9461	30.31%
main()	stt.listen()	8	36.6178	4.57722	25.841%
sttManager.init()	spacy.load()	1	40.8671	40.8671	28.84%
listen()	r.listen()	8	29.9044	3.73805	21.10%
listen()	r.recognize_google()	3	3.46645	1.15548	2.45%
Node			Text-To-Speech		
Function	**Instruction**	**Hits**	**Time**	**Time per hit**	**%**
tts_reply()	greeting_reply()	1	1.24309	1.24309	0.89%
greeting_reply()	speak()	1	1.24284	1.24284	0.89%
Node			Gesture		
Function	**Cumulative seconds**	**Self seconds**	**Self ms/call**	**Total ms/call**	**%**
moveArms()	0.68	0.04	40.00	606.56	2.41%
readJson()	1.66	0.00	0.00	5.00	0.00%

Table 5.1. Profile of the nodes involved during a greeting interaction with the robot

The second interaction taken into analysis is that of a request for information regarding the next scheduled event. In the case tested to get the data, the request is complete, thus correctly specifying the day in the request. However, even if the request was worded incorrectly, thus specifying a wrong date, the execution times would be equally comparable to those shown below, since the number of interactions would be the same, only the verbal response changes. The data of this execution are reported in Tab. 5.2. We can see that the time spent by the state dealing with this event in the State Machine Node is used in good part by the *sTTRequest* function, while filling the dictionary that collects the information needed to analyze whether the query is complete or not, i.e. *fillEventInfoFF*, is almost immediate. In this case the *r.recognize_google* function required a bit more time with respect to the previous case, perhaps because of the slightly greater complexity of the request. The text-to-speech service is requested just once and the function in charge of replying to this event, i.e. the *event_info _reply*, has a time of execution of about 1 second.

Information Request Interaction - Complete					
Node			State Machine		
Function	**Instruction**	**Hits**	**Time**	**Time per hit**	**%**
EventInfoHandler.execute()	fillEventInfoFF()	1	0.000123978	0.000123978	0.00%
EventInfoHandler.execute()	setTTSRequest()	1	4.42561	4.42561	13.97%
setTTSRequest()	reply(req)	1	4.27948	4.27948	13.51%
Node			Speech-To-Text		
Function	**Instruction**	**Hits**	**Time**	**Time per hit**	**%**
main()	stt.listen()	4	23.1008	5.77519	17.80%
listen()	r.listen()	4	17.7802	4.44504	13.70%
listen()	r.recognize_google()	1	2.47581	2.47581	1.91%
Node			Text-To-Speech		
Function	**Instruction**	**Hits**	**Time**	**Time per hit**	**%**
tts_reply()	event_info_reply()	1	1.06949	1.06949	0.84%
event_info_reply()	speak()	1	1.0563	1.0563	0.83%

Table 5.2. Profile of the nodes involved during an interaction in which the user correctly asks for information about the upcoming event

The third interaction examined is the same as the previous one, this time with an incomplete request made by the user, meaning that the day whose information the user wants to know about is not specified. As described in the previous chapters, in this case the robot first asks to specify the date and then it replies giving the requested information. This means that in this case each function involved in the process is called at least two times (this in case the answer to the date query is correct, since in case it is wrong, the user would have to ask the question again, thus increasing the number of calls to the functions). The data regarding this case are showed in Tab. 5.3. The times concerning the functions of the State Machine Node are practically the same of the previous case for what concerns the Time per hit, while the *r.recognize_google* function, despite the extra call with respect to the previous case, has a better performance, requiring about 1.24 seconds to be executed. We can notice very similar data also for the text-to-speech functions.

Information Request Interaction - Incomplete					
Node			State Machine		
Function	**Instruction**	**Hits**	**Time**	**Time per hit**	**%**
EventInfoHandler.execute()	fillEventInfoFF()	2	0.000204086	0.000102043	0.00%
EventInfoHandler.execute()	setTTSRequest()	2	8.61008	4.30504	21.51%
setTTSRequest()	reply(req)	2	8.39285	4.19642	20.97%
Node			Speech-To-Text		
Function	**Instruction**	**Hits**	**Time**	**Time per hit**	**%**
main()	stt.listen()	6	30.8282	5.13804	22.72%
listen()	r.listen()	6	25.1328	4.18879	18.52%
listen()	r.recognize_google()	2	2.49584	1.24792	1.84%
Node			Text-To-Speech		
Function	**Instruction**	**Hits**	**Time**	**Time per hit**	**%**
tts_reply()	event_info_reply()	2	2.41615	1.20807	5.77%
event_info_reply()	speak()	2	2.39504	1.19752	5.72%

Table 5.3. Profile of the nodes involved during an interaction in which the user incompletely asks for information about the upcoming event

The last case is that of requesting to be guided to a particular place and its data are reported in Tab. 5.4. To best analyze this case, a complete navigation has been completed, starting from the request until the reach of the destination. Obviously, the execution time for this particular case varies depending on the distance present between the robot's current location and the goal. The *NavigationHandler* is the state responsible for initializing the relevant resources for navigating the environment. We can see from the table that the initialization of the client, i.e. *SimpleActionClient*, requires about 1 second and the request for the server, *client.wait_for_server*, is almost immediate such as sending the goal to the server through the instruction *client.sendGoal*. The server processes the request and returns the result of its operations after about 2.44 seconds. Then the control of the operation passes to the *NavigationMonitor* state. In particular, we can see that the most time consuming control in this function is the one concerning the analysis of the data coming from the laser scan, present in the function *checkLaser*; the number of times this control is performed is decisively high, as we can see from the fact that it takes about the 31% of the total execution time. This is justified by the long time required for the interaction to be completed, as this is a control carried out throughout the duration of the interaction. As for the times for the other functions, as visible from the table, they are essentially similar to those for the other interactions.

Navigation Interaction					
Node			State Machine		
Function	**Instruction**	**Hits**	**Time**	**Time per hit**	**%**
NavigationHandler.execute()	SimpleActionClient()	1	1.17783	1.17783	0.56%
NavigationHandler.execute()	client.wait_for_server	1	0.355287	0.355287	0.17%
NavigationHandler.execute()	client.sendGoal()	1	0.0307121	0.0307121	0.01%
NavigationHandler.execute()	client.wait_for_result	1	2.44381	2.44381	1.16%
NavigationHandler.execute()	setTTSRequest()	1	5.10898	5.10898	2.42%
NavigationMonitor.checkLaser()	if(any(scan > 0.0 and scan < 1.0 for scan in msg.ranges))	853765	66.2104	7.75511e-05	31.34%
NavigationMonitor.execute()	setTTSRequest()	1	5.00312	5.00312	2.37%
setTTSRequest()	reply(req)	2	9.88372	4.94186	4.68%
Node			Speech-To-Text		
Function	**Instruction**	**Hits**	**Time**	**Time per hit**	**%**
main()	stt.listen()	47	198.17	4.21639	64.46%
listen()	r.listen()	47	192.165	4.08862	62.51%
listen()	r.recognize_google()	2	2.12476	1.06238	0.69%
Node			Text-To-Speech		
Function	**Instruction**	**Hits**	**Time**	**Time per hit**	**%**
tts_reply()	navigation_reply()	2	1.59289	0.796446	0.52%
navigation_reply()	speak()	2	1.58369	0.791845	0.52%

Table 5.4. Profile of the nodes involved during during an interaction in which the user has asked to be guided to a specific destination

Chapter 6

Conclusions

In this work, the ALTER-EGO robot was equipped with a software architecture used to implement autonomous behaviors in response to specific requests. The robot has become capable of responding to simple vocal interactions, such as a greeting, by replicating both verbally and through gestures, it is able to respond to a user's requests regarding the upcoming event and ask for particular information regarding the request in case the latter is incomplete, and finally, being aware of the map of the building in which it is employed, it is able to escort a user to a specific destination, while avoiding obstacles along the path. Each type of these interactions was tested in a simulated environment by having the robot simulator interact with a user via a microphone. Each test was successful, showing the robot's ability to correctly complete each interaction in any case, even those in which the user's requests were not included in those considered correct, while also showing satisfactory execution times of the various modules. Those described above are the basic capabilities needed to be able to employ a robot within an industry such as hospitality. The scenario for which the interactions developed in this project were defined is one in which the robot is responsible for greeting guests at a conference, testified by the specificity of the interaction regarding the request for information. However, the same functionalities can be employed in other types of contexts, again concerning hospitality, by making the right modifications to the type of interactions that a certain assignment requires. To give an example, the robot could be employed in welcoming guests at a hotel, in which case the interaction concerning the request for information regarding the upcoming event should be changed to one more consonant with the context such as the schedule of services offered by the facility. Precisely to encourage the use of the developed architecture in several applications, it has been designed to be easily expandable and improvable in many ways. In terms of improvements, the nodes of the architecture have been implemented to be flexible to the addition of new use cases. In particular, for what concerns the execution of the gestures, this functionality has been implemented to be as flexible as possible, introducing a formalism through which is possible to define a complex movement with a simple sequence of fixed poses. In case one wants to add a new gesture one would only need to fill the relative file, using the implemented template, with the poses related to the gesture and and the Gesture node will reproduce the movement, following the specified instructions. Also the addition of new speech interactions

is made easy by the use of simple patterns to be matched in the user sentence. Using this approach, to introduce a new interaction of this type it will be sufficient to define the characteristics of the pattern to be recognized, specifying a certain sequence of words or linguistic properties, and the relative responses adding only to the state machine the states related to this case without affecting the whole structure. Similarly, any expansion of the basic architecture is also made easy by the fact that the architecture is composed of individual independent modules, each with its own functionality, connected to each other by specific topics. So if one wants to add a new node, one would simply define it independently and connect it to the others in a consistent manner, without changing the basic structure. For what concerns possible future updates, some are the changes suggested. The first concerns the execution of the gestures: instead of publishing each pose for a certain number of seconds, it could be improved by integrating a more comprehensive control that can compare the current pose with the desired one. Another possible improvement concerns voice interaction with a user. In this project, simple patterns were matched in the user's sentences to find specific requests. However this strategy could be limiting, because for each specific kind of interaction that one wants to implement, a series of patterns must be defined in order to be able to respond to a request even if worded in different ways. An improvement could be the use of a better information extraction system, which does not dwell on individual words but on the overall meaning of the request, such that this one, even if formulated in a different way, returns always the same result.

Bibliography

[1] M. A. Goodrich and A. C. Schultz, "Human-Robot Interaction: A Survey," *Foundations and Trends in Human-Computer Interaction*, January 2007.

[2] D. Norman, *User-centered design: new perspectives on human-computer interaction*. Erlbaum associates: Hillsdale, 1986.

[3] J. Scholtz, "Theory and Evaluation of Human Robot Interactions," in *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03)*, 2002.

[4] R. C. Arkin, M. Fujita, T. Takagi, and R. Hasegawa, "An Ethological and Emotional Basis for Human-Robot Interaction," *Robot. Autonom. Syst.*, vol. 42, no. 3-4, pp. 191–201, 2003.

[5] C. Brezeal, "Regulation and Entrainment in Human-Robot Interaction," *Int. J. Robot. Res.*, vol. 21, no. 10-11, pp. 883–902, 2002.

[6] T. Shibata, K. Wada, and K. Tanie, "Tabulation and Analysis of Questionnaire Results of Subjective Evaluation of Seal Robot in Japan, U.K., Sweden and Italy," in *Proceedings of the 2004 IEEE International Conference on Robotics Automation New Orleans, LA*, April 2004.

[7] S. All and I. Nourbaksh, "Insect Telepresence: Using robotic tele-embodiment to bring insects face-to-face with humans," *Auton. Robots*, vol. 10, no. 2, pp. 149–161, 2001.

[8] A. D. Brito, J. C. Gamez, D. H. Sosa, M. C. Santana, J. L. Navarro, J. I. Gonzalez, C. G. Artal, J. P. Perez, A. F. Martel, and H. Teiera, "Eldi: An Agent Based Musem Robot," *Syst. Sci.*, vol. 27, no. 4, pp. 119–128, 2001.

[9] S. Thrun, M. Beetz, M. Bennewitz, W. Burgard, A. B. Cremers, F. Dellaert, D. Fox, D. Hahnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz, "Probabilistic Algorithms and the Interactive Museum Tour-Guide Robot Minerva," *Journal of Robotics Research*, 25 July 2000.

[10] R. Ambrose, H. Aldridge, R. Askew, R. Burridge, W. Bluethmann, M. Diftler, C. Lovchik, D. Magruder, and F. Rehnmark, "Robonaut: Nasa's Space Humanoid," *IEEE Intell. Syst.*, vol. 15, no. 4, pp. 57–63, 2000.

[11] M. Hans, B. Graf, and R. Schraft, "Robotic Home Assistant Care-O-Bot: Past-Present-Future," in *Proceedings of the 11th IEEE Int. Workshop Robot Human Interactive Communication*, pp. 380–385, 2002.

[12] J. Pineau, M. M, M. Pollack, N. Roy, and S. Thrun, "Towards Robotic Assistants in Nursing Homes: Challenges and Results," *Robotics and Autonomous Systems: Special Issue on Socially Interactive Robots*, vol. 42, no. 3, pp. 271–281, March 2003.

[13] J. Yang and E. Chew, "The Design Model for Robotic Waitress," *International Journal of Social Robotics*, 2021.

[14] M. Raibert, K. Blankespoor, G. Nelson, and R. Playter, "Bigdog, the Rough-Terrain Quadruped Robot," in *Proceedings of the 17th World Congress The International Federation of Automatic Control Seoul, Korea*, 2008.

[15] F. Negrello et al., "Humanoids at Work: The WALK-MAN Robot in a Postearthquake Scenario," *IEEE Robot. Autom. Mag.*, vol. 25, no. 3, pp. 8–22, 2018.

[16] G. Nelson et al., "Petman: A Humanoid Robot for Testing Chemical Protective Clothing," *J. Robot. Soc. Jpn.*, vol. 30, no. 4, pp. 372–377, 2012.

[17] G. Lentini, A. Settimi, D. Caporale, M. Garabini, G. Grioli, L. Pallottino, M. G. Catalano, and A. Bicchi, "Alter-Ego: A Mobile Robot With a Functionally Anthropomorphic Upper Body Designed for Physical Interaction," *IEEE Robotics Automation Magazine*, vol. 26, pp. 94–107, December 2019.

[18] G. Zambella and al., "Dynamic Whole-Body Control of Unstable Wheeled Humanoid Robots," *IEEE Robot. Autom. Lett.*, vol. 4, no. 4, pp. 3489–3496, 2019.

[19] M. Honnibal, I. Montani, S. Van Landeghem, and A. Boyd, "spacy: Industrial-strenght Natural Language Processing in Python," 2020.

[20] A. Zhang, "Speech Recognition (Version 3.8) [Software]. Available from https://github.com/uberi/speech$_r ecognitionreadme.,''$ 2017.