6CCS3PRJ Final Year Project
Web interface to interact with the TIAGo robot - Controller

Final Project Report

Author: Bhavik Gilbert

Supervisor: Gerard Canal

Student ID: 21004990

Program of Study: BSc Computer Science

July 19, 2024

Abstract

To the best of my knowledge, currently, there are not many accessible, simple, and reliable ways to interact with robots developed using the Robot Operating System (ROS). From what I've seen, most existing methods require either knowledge of how to create packages in the Robot Operating System (ROS), how to interact with topics in the command line using the Robot Operating System (ROS), or access to a physical controller, which may be prone to problems such as joy-con drift or an unstable Bluetooth connection.

This project aims to create an easy to use, reliable and quickly accessible web interface to control and interact with a robot developed using the Robot Operating System (ROS). This would allow users to control their robot with a more accessible device such as a smartphone or laptop, connected over a more stable LAN connection, with a friendlier web interface. This project will focus on developing the platform for a single model of robot, TIAGo, which could then be easily expanded upon to work for other kinds of robots developed using the Robot Operating System (ROS).

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Bhavik Gilbert

July 19, 2024

Acknowledgements

I would like to give a thanks to my project supervisor Dr.Gerard Canal for his guidance, support and supervision throughout the project.
I would also like to thank Sahitya Sakthivel, Matthew Palmer, and Reshma Dhillon, for their help testing the system.

Contents

Acronyms

API Application Programming Interface. 25

CORs Cross-origin resource sharing. 15, 22, 32, 40, 78

CSS Cascading Style Sheets. 36, 37, 47

HTML HyperText Markup Language. 52

HTTP Hypertext Transfer Protocol. 14, 16, 32, 40, 66, 81

IO Input Output. 13, 14

IP Internet Protocol. 67

ISO Information Commissioner's Office. 15, 28, 39, 64, 79

LAN Local Area Network. 5, 32, 81

LIDAR Light Detection and Ranging. 49, 50

LOA Levels of Operation. 16, 82

MRI Magnetic Resonance Imaging. 11

npm Node Package Manager. 21, 37, 60

PECR Privacy and Electronic Communications Regulations. 15, 28, 39, 64, 79

RGB-D Red Green Blue Depth. 10, 67, 83

ROS Robot Operating System. 1, 4–6, 8–11, 20–25, 27, 28, 30, 33, 35, 38, 45, 47, 48, 59, 63, 66, 67, 72, 73, 80–83, 94, 95

SAE Society of Automotive Engineers. 16, 17, 90

SOP Same-origin Policy. 14, 15, 32

UI User Interface. 1, 21, 23, 27, 28, 34, 39, 66, 90

UK United Kingdom. 15, 20, 64, 79

WAN Wide Area Network. 32, 81

WSGI Web Server Gateway Interface. 22

Chapter 1

Introduction

Robotics is a research field which in the coming years has the potential to see major growth in uses outside of the lab environment. To push this effort forward, this project is the creation of an easy-to-use, reliable, and quickly accessible all-in-one web interface to control and interact with robots developed using the Robot Operating System (ROS). The goal is to make an application that anyone could pick up on the go and use with relative ease to control their robot, irrespective of robotics experience, to increase the number of usable scenarios for ROS robots. Due to the varied nature of ROS robots, this project will be developed for and tested on a single model of robot, TIAGo, which supports the entire feature set of the application. Despite this, the application's design will enable use with other ROS robots that share at least some subset of akin functionality.

1.1 Project Motivations

This project is motivated by a lack of simple and reliable ways to control robots built using ROS. As an open-source platform, it helped to open robotic development and usage to a wider audience. Though this is the case, akin to the early days of computers, the interface provided is not especially human-friendly, being terminal-based. Though it is generally used by professionals, who are willing to learn or are used to operating in such environments, it can be somewhat slow to setup or unintuitive to use when just trying to perform a basic task such as manually moving the robot around or speech operation. Alongside this, it can be quite daunting for users who don't want to develop software and only want to operate the robot. Furthermore, the requirement of a terminal effectively forces the operator to use a desktop, which can lead

to further inconvenience on the operator's part, being that users spent just over 6 % more time on smartphones than desktops in 2017, which is up from the around -30 % difference in 2015 [1], that has likely only risen since if the trend has continued.

Beyond the traditional terminal-based operation, there are few and far between intuitive methods of operation. For those that are, they tend to be bound either by hardware or software to a given type of robot or have a single functionality, giving them minimal use with other types of robots. TIAGo is a research robot made by Pal Robotics, which we will be focusing on during this project, though the final software produced will offer some form of support to all Robot Operating System (ROS) based robots with akin functionality. Some existing segregated pieces of interface software allow users to interact with TIAGo, which we discuss in detail below in Chapter 2. Alongside this, for movement, TIAGo offers a direct pass-through to the motors for control using a physical controller, which though may seem like it removes some of the need for this system with TIAGo, succumbs to some flaws of controllers, including the unstable nature of a Bluetooth connection in comparison to a high-speed Local Area Network (LAN), and joy-con drift. Joy-con drift is when debris builds up on the actuators in joysticks, resulting in false or unbalanced inputs.

Given all of this, the three core motivations of this project are to reduce the amount of effort needed for professionals to perform basic operations when using a Robot Operating System (ROS) robot, to enable layman users to control Robot Operating System (ROS) robots with little hardship and to allow operators to control Robot Operating System (ROS) robots using a wider variety of devices, such as phones and tablets. Some potential use cases for this are testing robot physical functionality, testing robot software functionality, robot demonstrations and layman robot use, personal and professional.

Moving onto some personal motivations, having learned, and developed using ROS left me thinking about its expansive set of potential use cases in the robotics industry. At the same time, though development felt natural, non-autonomous use felt padded as it had too many unnecessary steps. Though there are existing implementations of control using physical controllers, they succumb to their physical flaws such as joy-con drift or an unstable connection, which can result in very costly damage in real-world situations.

Furthermore, I would like to say that I have spent quite a bit of time designing and developing different web applications, enough so to say that I find the overall process of developing and designing web applications enjoyable. From the design of a user experience including the user interface to the implementation of a well-fitted and efficient backend, the process is one in which I am enthusiastic to take part.

As a full-stack software developer, I highly enjoy problem-solving in many ways, but more than that, I like to provide value. This project covers an area which could help bridge a gap between what we have now and the seamless use of ROS.

1.2 Project Aims

The main aim of this project is to be able to interact with TIAGo remotely via a web application and using this, be able to control the movement of its base and speech. The secondary aim would be to provide this capability in a way that ensures the safe use of TIAGo as described below in Section 1.2.3.

1.2.1 Base Movement

The aims for base movement relate to the control of the movement of TIAGo's base while using this application.

- Users can manually control the linear motion of TIAGo's base.
- Users can manually control the rotation of TIAGo's base.
- Users can select pre-defined movement actions for TIAGO to perform.

1.2.2 Speech

The aims for speech relate to controlling TIAGo's speech while using this application.

- Users can provide messages for TIAGo to speak out using this application.
- Users can select pre-defined messages for TIAGo to speak out using this application.

1.2.3 Safety

The aims for safety relate to the safe control of TIAGo, to minimise the chance of damage while using this application.

- The system shifts some of the burden of safety of operating TIAGo away from the user .
- Only one user can provide an input to TIAGo at once.

Chapter 2

Background

Before talking about the design and development process of the system, we'll cover some important background information and literature which were considered during the design and development process. This covers the topics of robotics, controllers, web security and shared autonomy, going into some research and placing its relevance in the project where appropriate, alongside sharing some thoughts.

2.1 Robotics

2.1.1 Robot Operating System (ROS)

ROS is an open-source operating system released in 2007. Its core philosophical values are peer-to-peer, tools-based, multi-lingual, thin, free and open-source. ROS can be seen as a network-based approach to robotics development, taking on a modular approach.

"A system built using ROS consists of several processes, potentially on several different hosts, connected at runtime in a peer-to-peer topology" [2]. This approach opens many possibilities in terms of scalability and efficiency by allowing for the running of tasks on many different hosts, whose outputs can be used by one another, as long as they run on the same network.

During its design, the creators leaned towards a lightweight approach to reduce complexity, resulting in them adopting a micro-kernel for the system. The idea was to have several small tools run the ROS components, rather than a monolithic system. This was to help improve the reusability of code. The reason they chose this approach is that they found that other systems

at the time using a monolithic approach, such as drivers, have several reusable code components, but due to being highly coupled to middleware, they are effectively unusable outside of that specific purpose.

2.1.1.1 ROS Features

A couple of ROS features will be brought up throughout this paper. These include rostopics, play motions, publishers, subscribers, action clients and actions.

A rostopic, otherwise referred to as a topic, can be thought of as named storage where data can be written to and read throughout a ROS network, and is how different nodes on a ROS network asynchronously communicate with one another [3]. Each topic on a ROS network will have a different name, akin to variables in programming, with each rostopic being expected to store a value for differing purposes.

A subscriber is a type of ROS node which listens to a given rostopic, and executes a given function when the value at that rostopic changes. A publisher is another type of ROS node which writes a value to be stored at a given rostopic [4].

An action client is used to request services from an action server, which can be used to request a robot to perform certain actions, via providing goals, which is a message telling the action server what you want to robot to do [5]. Using an action client allows finer control over actions, as alongside being able to request to perform them, they can also be used to cancel actions, get feedback on actions that are being performed, and get the result of whether an action was completed or not.

A play motion is a file containing recorded inputs [6] which can be loaded into the ROS master's parameter server. These will commonly be executed and managed via an action client.

2.1.2 TIAGo

"TIAGo is the mobile manipulator of PAL Robotics that has been designed in a modular way" [7]. It is a humanoid service robot modelled on a modular architecture and designed to assist in research, developed by PAL Robotics.

Figure 2.1: TIAGo Modular Robot [8]

TIAGo is equipped with many modules including a RGB-D camera, laser range-finder, differential drive base, microphone and speaker, amongst others [8]. A visual of TIAGo and some of its features are displayed above in Figure 2.1 Its modular components support ROS, allowing developers to communicate and operate it over a ROS network.

2.1.2.1 TIAGo Base

The TIAGo Base is the bottom module of a TIAGo robot, which can be mounted by several other components besides the current TIAGo combination shown earlier. It is an autonomous mobile robot for indoor use. The TIAGo Base can come in a few different configurations including, a two wheel differential move base, a set of omnidirectional mecanum wheels and two LIDAR sensors [9].

2.1.3 Pal Robotics Web User Interface

The creators of TIAGo, Pal Robotics, have developed a paid web platform for TIAGo to help create motions, program for and control base movement [10].

Both the Pal Robotics web interface and this project have a virtual joystick. It also offers a view of the 2D map of the environment that TIAGo is currently in, alongside a single joystick to control motion, with linear and angular speed text boxes.

Three core aspects differentiate this project from the web interface made by Pal Robotics.

1. The joystick aspect of the project does crossover with existing work, but due to the nature of it being bundled within the TIAGo Web User Interface suite, it is inaccessible for use with other ROS robots. This makes it difficult, if not impossible, to extend for use with other ROS robots. Though this project aims to develop a system for TIAGo, it aims to be made in an easily extendable way, and therefore support other types of ROS robots that are not TIAGo.

2. On top of the joystick, this application aims to have the means to control the speech of TIAGo, alongside including a set of saved, pre-defined motions for users to select from.

3. This project aims to develop a system that ensures the safe use of TIAGo. This means there will be some safety features built in to reduce the burden of safe use on the end user, which is not something the TIAGo Web User interface supports as it is aimed at developers and researchers.

2.2 Controllers

Controllers are devices that translate muscle movements into electronic inputs. They are a common way in which users interact with a multitude of systems.

2.2.1 Controller Design

Studies into controller design have become more important over the last few decades, with the expanded use cases of controllers in more critical systems, some examples being the operation of medical imaging technology such as Magnetic Resonance Imaging (MRI) scanning [11],

alongside the operation of drones [12]. Controller design expands into many areas, including ergonomics, interface architecture, and symbolism.

2.2.1.1 Controller Ergonomics

Ergonomics is one of the most widely discussed topics regarding controller design. Though studies on the optimal ergonomics of physical controllers may not be directly relevant to a virtual web controller, I feel it is important to discuss the environment existing design solutions were modelled for.

Most controllers currently adopt an arrangement that has inputs at either end of the controller, making them easier to reach, resulting in the same or akin input types on either side of the controller. Alongside this, they follow a bent-grip design, where the ends of the controller bulge on either end to provide a surface for the user to grip onto [13]. Modern controllers are highly optimised for both one-handed and two-handed use.

2.2.1.2 Web Controllers

In web applications, a common approach to creating functional interactive software systems is using the model view controller architecture [14]. The view is used to display information of the current state to the user. The controller is used to process user inputs. The model stores the information of the current state that is displayed by the view, alongside containing the logic on what to do with and change that information based on user interaction. Combining these allows for the creation of a functional interactive software model, such as a virtual joystick.

Moving onto web controller interface design, due to the nature of use between the differing device types that can access websites, they tend to be less optimised for a particular form factor. Despite this, the offset design from the ergonomically refined controllers is commonplace when using virtual web controllers. Not much research has been focused on this space, making it hard to pinpoint the definite reasoning for why such design language has carried over. However, the conventional approach to controller design is still regularly adopted when designing web applications. An example is the Pal Robotics Web User Interface described in Subsection 2.1.3.

2.2.1.3 Controller Symbolism

With the growth in the use cases for controllers over the decades, semiotic [1] studies about them have taken place to understand and find out how to utilise them. The study of semiotics relates to the study of signs and symbols. Given the limited available space on a controller, virtual or physical, the goal to keep controllers simple and intuitive, alongside the preface to ensure the meaning of each input is correctly conveyed to the user, results in the symbolism used being significant.

When determining what symbols to use, contiguity and conventionality are commonly considered [15]. Contiguity aims to use the environment that the use of the controller puts you in within the symbol to provide context clues for what action a given input may perform. This establishes a visual and tactile reference between the controller and the action. This is generally difficult to do in a virtual setting. Conventionality aims to use common knowledge shared between people within the symbol to convey what action a given input will perform. When using conventionality to determine symbols, further considerations need to be made. Some symbols can be considered universal, while others can have differing meanings depending on the environment, the system is in, or the background of the user. It can't be ensured that all users will interpret a given symbol the way you want them to. A mix between the two methodologies is generally used to determine symbolism.

Following these methodologies, in the current day, controllers have tended to what could be called their final form, with most multi-purpose controllers now all adopting an interchangeable set of symbols and inputs, which can be considered widely known.

2.2.2 Controller Input Output (IO)

Currently, the most explored methods of IO "apply to (1) finite-dimensional, (2) linear, (3) time-invariant, and (4) continuous-time" [16]. From these, the two most used IO types are linear and continuous. Linear IO can be one of two finite states, on or off, a common example being a button. Continuous IO can take on a variable number of inputs between a finite set of states, a common example being a slider.

Generally, for a given output, the same input type is used, but there are certain situations

[1] Semiotics is the study of signs and symbols, and their use or interpretation.

where designers may decide to give the user a different kind of input for the intended output. Such IO models are examples of hybrid control systems.

2.2.2.1 Hybrid Control Systems

Hybrid control systems aim to solve 4 problems with controllers, optimal control, hierarchical control, distributed multi-agent control, and least restrictive controllers for specifications [17]. Optimal control aims for a control model that minimises the cost incurred for making an action. Hierarchical control refers to the breakdown of action flow that can occur on a controller, to ensure that controllers can guarantee performance for critical actions. Distributed multi-agent control is a way in which optimal control problems are broken down and analysed so that they can be solved by a group of agents with defined communication and information architecture. The least restrictive controller for specifications is a generalisation for a collection of properties that controller models should aim not to restrict, such as safety and liveliness. The features of hybrid control systems result in them being ideal for use in high-confidence systems, those being systems that are expected to work with a high level of confidence, with them necessitating reliability, correctness and off-loading of faulty components during operation.

2.3 Web Security

As a project with network-based elements, a proponent of web security is involved, even if the project is designed for use over a LAN connection. The importance of this area will become more apparent later in Sections 4.1 and 4.2 when talking about my intended solution to the problem area proposed by this project, which is clarified in Section 3.1. Sessions, cookies, Hypertext Transfer Protocol (HTTP) and Same-origin Policy (SOP) are four current security measures accounted for in existing threat models.

2.3.1 Cookies

Cookies are a form of data storage that is stored in the browser. They are shared across all ports and protocols for a domain but can also be set for the parent of a domain, with them being shared between a parent domain and all subsequent subdomains [18]. They are generally used to provide secure data storage for data used over a relatively short period, which helps to maintain confidentiality.

2.3.1.1 Cookie Legislation

In the United Kingdom (UK), the use of cookies in applications must abide by the Information Commissioner's Office (ISO) [19] policies, with cookies falling under Privacy and Electronic Communications Regulations (PECR) [20].

2.3.2 Sessions

"A web session is a semi-permanent information exchange between a browser and a web server involving multiple requests and responses" [18]. Sessions to ensure that services aren't provided to unauthorised users or an insecure environment.

They are typically an assortment of seemingly random characters that expire after a relatively short period, which can help a server identify a client when they request access to a service. This is generally achieved using a cryptographic hash function. An example of this is a collision-resistant hash function, which is a one-way function that takes an input of an arbitrary length and outputs a value of a fixed length, alongside being resistant to there being two different inputs with the same output [21]. Typical session models involve the server distributing a session token to a client on their initial connection request after validating the client, which can then be sent with future requests to gain access to server services, which helps to maintain confidentiality, authentication, and integrity.

2.3.3 Cross-origin resource sharing (CORs)

Same-origin Policy (SOP) is a security policy that is implemented in almost all major web browsers. "It enforces a strict separation between content provided by unrelated sites, which is crucial to ensure their confidentiality and integrity" [18]. This protects processes that are on the same domain but have different origins from interacting with one another, maintaining confidentiality.

CORs is a security policy that exposes processes on a given origin to all other processes on its domain. As opposed to SOP, it is used in the case that two processes in the same domain need to communicate with one another. When using CORs, confidentiality is pushed to the runtime environment, being the domain, rather than the system itself.

2.3.4 Hypertext Transfer Protocol (HTTP)

HTTP is a protocol that programs can use to communicate over a network, most used over the World Wide Web. At a high level, the HTTP protocol works with a client sending a request to a server with the server sending a response back to the client. The most used HTTP request methods are GET and POST. The GET method was designed to serve content that remains the same for all clients for a given URL. "The POST method was designed to send input data to the server" [22]. Though they can effectively perform the same task, the GET request is less complex and less secure, and therefore faster to process, while the POST request is more complex and more secure, and therefore slower to process.

2.4 Shared Autonomy

The concept of shared autonomy is to keep the human controller in the loop, with the team of a human and a machine jointly maintaining situational awareness to keep in control. Though this project focuses on TIAGo, we will cover vehicular research for this field as they are akin to how TIAGo's base operates at the software input level. Within the vehicle space, "the traditional approach to highly automated vehicles is to skip consideration of the human all-together and focus on perfecting the mapping, perception, planning and other problems characterized by the exceptional performance requirement" [23], which is classified as full autonomy.

In the area of shared autonomy, commonly, some arbitrary levelling system referred to as the Levels of Operation (LOA), is used to define the relationship of responsibility and control between the machine and human, otherwise known as the taxonomise of automation. Though there are many variations of the LOA that can be found, they mostly follow the basis that the lowest level puts the human fully in control, with the highest level putting the machine fully in control, with the levels in between reflecting some ratio of shared control, with lower levels giving the human more control and higher levels giving the machine more control [24]. Throughout vehicular shared autonomy research, a popular scale used for the LOA is one made by the Society of Automotive Engineers (SAE), which recognises 6 levels from 0 to 5, that is described further below in Figure 2.2 [25]. We will use this system when referring to the LOA throughout the rest of this paper.

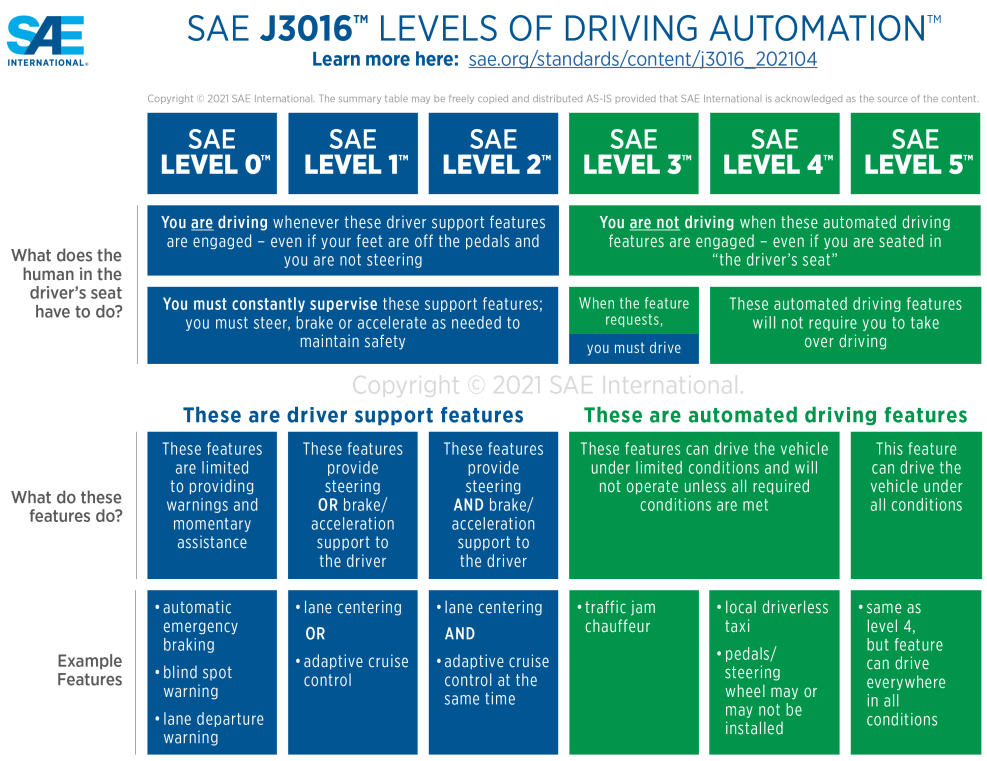

Figure 2.2: Society of Automotive Engineers (SAE) Levels [26]

One approach to shared autonomy is the human-centred autonomous vehicle approach. In its current usage, it is described as a Level 2 system, where responsibility remains with the driver, but depending on human factors such as driver state, driving style and prior joint experience of the human and machine, several control mechanisms such as steering, acceleration and deceleration can be controlled by the system rather than the human. In its most basic form, shared autonomy can work as a universal system where machines only aid in balancing out human inputs in a logical way, due to the lack of precision humans provided with non-linear inputs.

Current limitations of such a system come because of deep personalisation. This is the concept that all aspects of the system's operation should be drawn from all the previous experience it has built up with the driver, which should make every system highly unique. A way to get such data would be human sensing. In the case of driving, it "refers to multi-modal estimation of overall physical and functional characteristics of the driver including level of distraction, fatigue, attentional allocation and capacity, cognitive load, emotional state, and activity" [23].

Gaining such data not only helps to build up the personalisation of the system but can also be used to gauge shared perception. The traditional fully autonomous approach exists to remove the task of perception control, as in its current state, calculations can be very unreliable and unpredictable due to the many factors involved. Removing humans from the calculations makes it much easier to solve this. As a result of this, in shared autonomy, the task of perception control has moved from being about full control of driver perception to supporting the driver with important external information they may not have access to, while also ensuring they know the limitations of such systems so that they don't become over-reliant.

Using the concepts built up in shared autonomy, even without user personalisation, this system can make simplistic but effective approaches to interpret user inputs, to maximise intended results, even with imperfect inputs. More specifically, these concepts are used for crash prevention and movement input smoothing mechanisms in this project.

Chapter 3

Requirements and Specification

With the aims of the system in mind, a set of requirements has been created. Following that, an explanation of all of the technologies used throughout this system is provided, including the specification of the TIAGo robot, the frontend technologies used, and the backend technologies used.

3.1 Requirements

3.1.1 Functional Requirements

As this system is developed with a focus on TIAGo, the functionality will be based on its use cases. Though this is the case, in implementation, open-ended approaches should be used to enable functionality to be easily extended for other robots.

The functional requirements for this system are:

- **FR1** - The system must provide a way to communicate with TIAGo via the web interface.
- **FR2** - The system must allow users to control the linear and angular movement of TIAGo's base.
- **FR3** - The system must allow users to send custom messages for TIAGo's to say.
- **FR4** - The system must provide a set of pre-defined movements to select from for TIAGo to perform.
- **FR5** - The system must provide a set of pre-defined messages to select from for TIAGo to say.

- **FR6** - The system must stop inputs that are deemed to jeopardise the safety of TIAGo.
- **FR7** - The system must ensure only one client can control TIAGo for a given server of the application at once.
- **FR8** - The system must help the user to provide consistent base movement inputs.
- **FR9** - The system must provide feedback to the user, based on the outcome of their inputs.

3.1.2 Non Functional Requirements

Given the aims of this project, the system must not only be functional but provide a good user experience. This results in some requirements stemming from user interaction rather than exact system functionality.

The non-functional requirements for this system are:

- **NFR1** - The system must be simple to setup.
- **NFR2** - The system must be intuitive and easy to learn.
- **NFR3** - The system must allow for simple switching between usage of features.
- **NFR4** - The system must ensure the safe operation of TIAGo.
- **NFR5** - The system must comply with UK data regulations.

3.2 Specification

3.2.1 Robot Specification

This subsection refers to any specific hardware or software onboard the model of TIAGo that the system is tested on. Either a physical TIAGo robot or a ROS container with the code to simulate a physical TIAGo robot and environment can be used.
TIAGo specs [8]:

- Height: 110-145cm
- Footprint: ø54cm
- Arm Payload: 3Kg (without end-effector)

- Battery autonomy: 4-5h (1 battery)/8-10h (2 batteries)
- Mounting points: On the head, laptop tray and mobile base
- OS: Ubuntu LTS, Real-Time OS

3.2.2 Frontend Specification

The frontend refers to the client-side portion of this project, the client side being what is run in the browser. Here, a Nodejs based solution using Reactjs will be implemented to create the User Interface (UI).

3.2.2.1 Nodejs

Nodejs is "an asynchronous event-driven JavaScript runtime" environment [27]. It is a commonly used runtime environment that is often paired with Node Package Manager (npm), which manages the installation, deletion, and version-control of packages within a Nodejs application [28]. Nodejs has a large community of developers with several existing tools which can be easily imported and integrated into applications.

3.2.2.2 Reactjs

Reactjs is a "library for web and native user interfaces" created by Meta [29]. It enables the development of user interfaces from components, with a built-in testing suite. In conjunction with Nodejs and npm, it is a fast and efficient tool to create and test a UI.

3.2.3 Backend Specification

The backend refers to the server side portion of this project, the server side being what is run on the web server where the application is hosted. The system will be running a Python3 backend, using some frameworks and packages including roscore, rospy and flask.

3.2.3.1 Roscore

"Roscore is a collection of nodes and programs that are pre-requisites of a ROS-based system", and will start up a ROS Master, a ROS Parameter Server and a rosout logging node when activated [30]. It can be run in either ROS Melodic or ROS Noetic. ROS was chosen over ROS2 due to previous experience with the technology. The role of roscore in this project is to establish part of the communication pipeline between TIAGo and the web server.

3.2.3.2 Rospy

"Rospy is a pure Python client library for ROS. The rospy client API enables Python programmers to quickly interface with ROS Topics, Services, and Parameters" [31]. It allows for the quick and simple implementation of communication with roscore.

3.2.3.3 Flask

Flask is a Web Server Gateway Interface (WSGI) micro-framework for Python, that can be used to set up a server [32]. The flask server's main purpose will be to act as a bridge between the browser and the roscore. It was chosen for two main reasons. The first reason is that it allows for simple but robust communication between it and roscore using rospy, alongside easy communication with the browser using CORs. The second reason is that it has several libraries built for it which could be useful regarding one of the components of safe the use of the robot, being that only one user can control a given robot using this application for a given server at once.

Alongside these reasons, using Flask gives the system a gateway to a Python environment, with Python having a catalogue of frameworks and libraries available to it that can be used throughout this system. Using alternative methods that rely on existing javascript tools to connect to the ROS servers such as rosnodejs [33] and roslibjs [34] wouldn't give the system access to a catalogue of frameworks and libraries in the same way having access to a Python environment does when using flask. This enables me to have more control of the system design, allowing for a more open design approach, and making it easier to design the system in a way that can be expanded for other ROS robots besides TIAGo in the future.

Chapter 4

Design

Using the research from the background, the system has been designed and moulded around its requirements. Here we'll have a high-level understanding of the system's design, and how it will expect to function, going through the data communication models considered, the potential use cases, the layout of the core UI components, the incorporated security measures and the third-party software included in the system. Each area will touch on why design decisions were made.

4.1 Data Communication Model

Given the available technologies, two data communication models were considered. Both models rely on using some intermediary system to allow the client browser to communicate with the ROS Master.

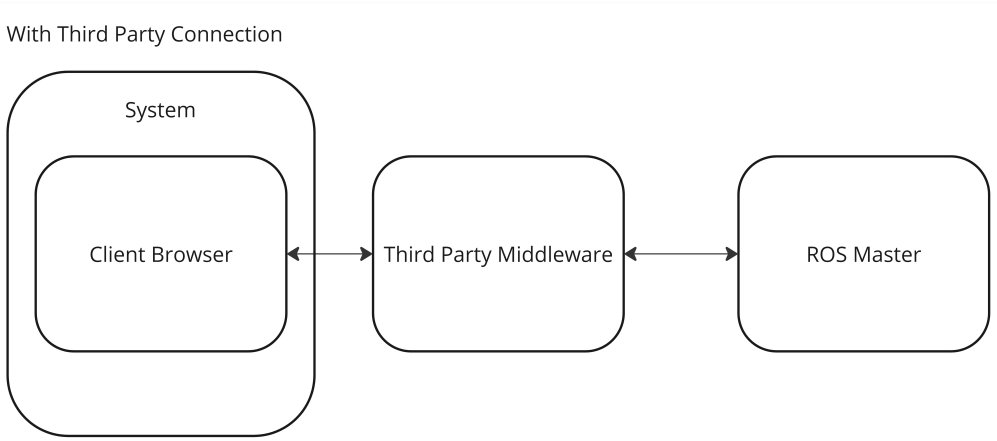

Figure 4.1: Data Communication Model 1 - With Third Part Connection

The first data communication model involves the client browser sending data through third-party middleware to communicate with the ROS Master, as shown in Figure 4.1.
The benefits of this solution are that it results in less development required, a lower maintenance workload and requires fewer system resources to operate.
The drawbacks of this solution are that it requires the installation of extra packages, creating an added reliance on the system that can't be controlled while restricting communication to the provided formats.

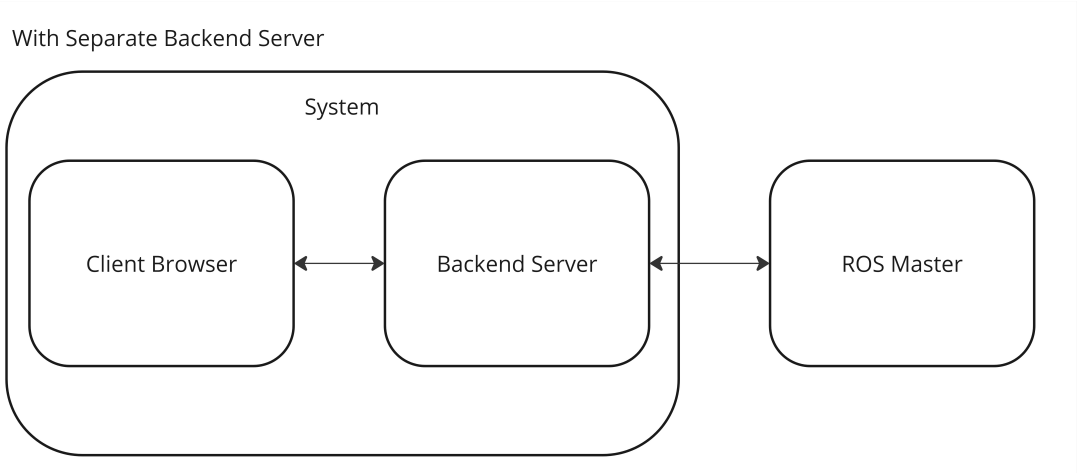

Figure 4.2: Data Communication Model 2 - With Separate Backend Server

The second data communication model involves using a backend server alongside the client browser to communicate with the ROS Master, as shown above in Figure 4.2.
The benefits of this solution are that it makes it easier to keep the system lean, makes the code base more expandable and increases the system's reliability in the long run.
The drawbacks of this solution are that it increases the development workload, increases the maintenance workload, and requires more system resources to operate.

The chosen data communication model for the system is the one incorporating a separate backend server. It provides a leaner, more reliable and more expandable solution to the problem of communicating with ROS Master. Though it may increase local operation resources and development time, it's a trade-off worth taking, as this results in less client side compute and storage being used, which increases the number of devices and scenarios in which the system can be used.

It also adds the potential for the server to act as a base for, to the best of my knowledge,

the first web-based rospy interface Application Programming Interface (API). Existing solutions make use of rosbridge, which provides direct JSON API to ROS functionality [35], meaning that no further data processing occurs once the data is sent. This opens the potential for an extended version of the server to act as a web interface suite that not only provides ROS functionality to non-ROS projects, but also takes on the burden of safe inputs alongside other data processing capabilities.

4.2 Use Cases

The use cases help to identify the different interactions that can occur between the actor and the system. In the use case diagram, the user and TIAGo will be modelled as actors. It describes how the system should function given a set of actions for each actor and defines what tasks the system is required to perform, as shown below in Figure 4.3.

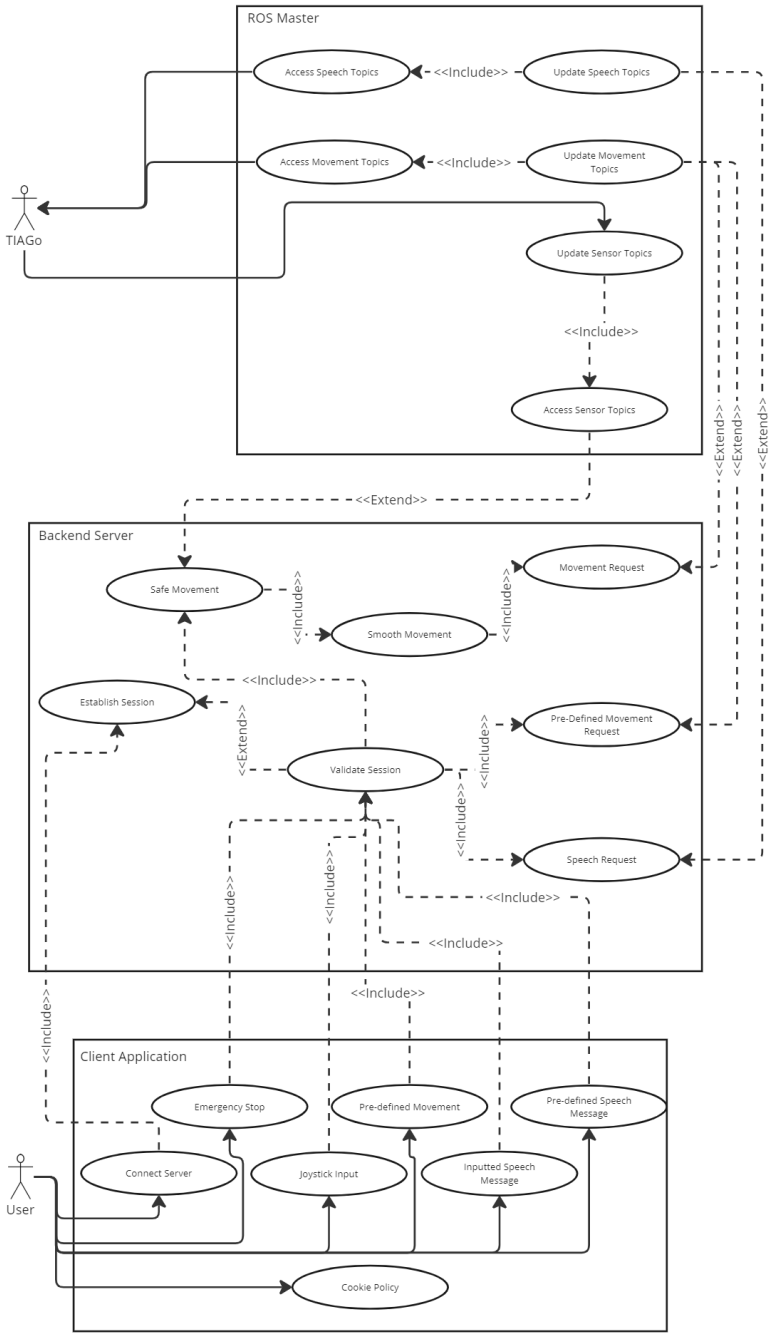

Figure 4.3: System Use Case Diagram

To explain what is happening in Figure 4.3, on starting the client application, the system will be required to connect to the server and establish a session with the backend server. Once established, the user can perform five main operations. All of these operations start by sending a request to the backend server and having the client's session validated and established if not already done so.

The joystick input request or emergency stop request causes the backend server to check all relevant sensor topics on the ROS Master to ensure that the inputted movement is safe to do. An example of a safe movement is one where the movement doesn't have TIAGo crashing into an object. If deemed safe, the input movement will be compared with previous inputs, and if similar, will be average to smooth the movement. The backend server then sends a request to the ROS Master to change movement by updating the given movement topic values to the movement values.

The pre-defined movement input request causes the backend server to send a request to the relevant movement topics to perform the inputted action.

The inputted speech message input request or pre-defined speech message input request causes the backend server to send a request to the ROS Master to change the message to TIAGo to sat by updating the given speech topic values to the message values.

4.3 User Interface (UI)

4.3.1 Layout

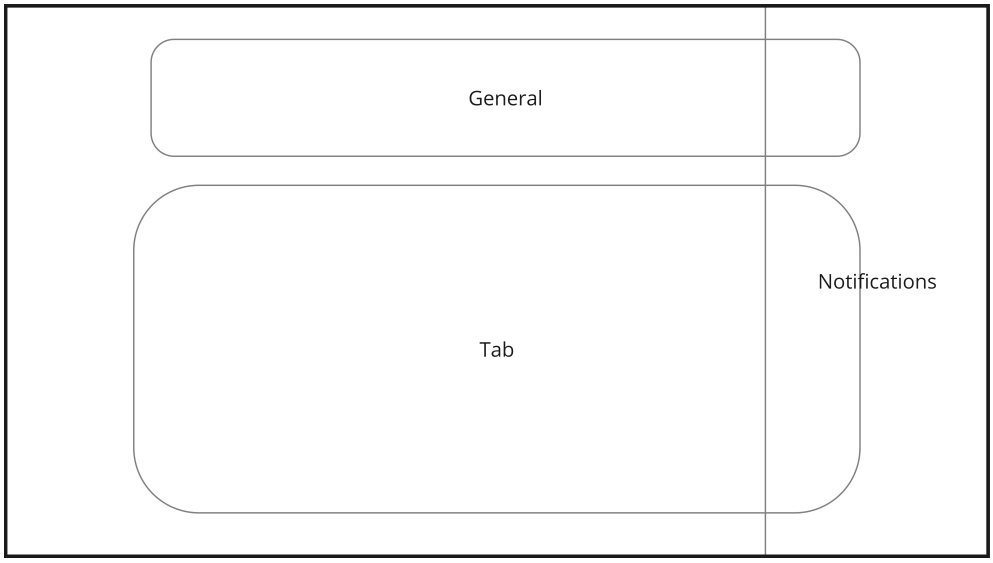

Figure 4.4: General UI Layout

As shown in Figure 4.4, there are three main components to the UI, tabs, general and notifications. The general area is for key inputs and information that will always be displayed. The tab area is for the functionalities of the system that do not all need to be displayed at the same time. The notifications area is where notifications will be displayed over the other areas when a notification is sent out.

4.3.2 Cookie Policy

As shown in Figure 4.2, before accessing the client's functionality, the user will be shown the client's cookie policy and is required to agree before continuing to use the rest of the application. This is done to comply with ISO PECR policies.

4.3.3 Server Connection

To choose a server URL, a textbox is provided. By default, the current domain is inputted with the default port of the backend server. A submit button is also provided to check the client's connection to the server at the inputted URL. These UI components are available in the general area of the page, as shown in Figure 4.4.

4.3.4 Topic Selection

To choose the rostopic to update, a selection field with set options is provided. The set options will be the topics found on ROS Master. The selection field will support search, to make it easier to find and select topics. Alongside that, a refresh button will be provided to update the list of rostopics with the most up-to-date set of rostopics. These UI components are available in the general area of the page, as shown in Figure 4.4.

4.3.5 Tab Structure

Within the tab area, there are two main tabs, the movement tab, and the speech tab. Within those, the user can navigate between sub-tabs.

4.3.5.1 Movement Tab

The movement tab is aimed at containing tabs relating to controlling the movement of TIAGo. The sub-tabs for the movement tab are the joystick tab and the play motion tab.

4.3.5.2 Speech Tab

The speech tab is aimed at containing tabs relating to controlling the speech of TIAGo. The sub-tabs for the speech tab are the manual speech sub-tab, pre-defined speech sub-tab and speech history sub-tab.

4.3.6 Joystick Sub-Tab

The joystick enables users to input movement commands, by default for TIAGo's base. Its inputs are translated into linear and angular velocity. Three designs were under consideration.

1. One joystick with vertical and horizontal movement which will correlate to linear and angular velocity.
2. Two joysticks, one with vertical movement, and another with horizontal movement, with one correlating to linear velocity and the other to angular velocity.
3. One joystick with vertical movement which will correlate to linear velocity alongside 2 rotation buttons, to represent clockwise and anti-clockwise angular velocity.

Option 1, being one joystick alone, is what was decided. It is the most natural to control across platforms (mobile, tablet, laptop, desktop), which support multi-touch inputs and single-touch inputs.

Alongside this, numerical fields to dictate the percentage of the max speed to set the linear and angular velocity are provided. A sticky joystick slider to toggle the joystick providing continuous inputs is displayed. These are displayed above the joystick component.

4.3.7 Manual Speech Sub-Tab

The manual speech sub-tab includes a text box where the user can write the message to be spoken, a search selector field with options of languages for what language the message is in, and a submit button to tell the client to send the message to the current topic. With the correct topic selected, the robot will speak out the message.

4.3.8 Pre-defined Input Sub-Tabs

For sub-tabs with pre-defined inputs, such as play motion, pre-defined speech and speech history, a grid of text-based cards describing the data it represents is displayed. Alongside this, a search field will be provided to speed up finding a given input.

4.3.8.1 Play Motion Sub-Tab

On the play motion sub-tab, input data is displayed as the name of the play motion. Similar to the rostopics, as the play motions are available via the ROS network, a refresh button is provided to update the list of available play motions. Alongside this, due to the slow and careful nature of play motion actions, after playing a play motion, a cancel button will appear in place of the play button for that play motion, which will then turn back to the play button when that play motion ends or is cancelled. Furthermore, activating further play motions via the system when another play motion is active will be rejected until the initial play motion terminates, either via competition or cancellation.

4.3.8.2 Pre-defined Speech Sub-Tab

On the pre-defined speech sub-tab, input data is displayed in the form of the text that will be sent for TIAGo to say if selected, using the native character set of the language it is in.

4.3.8.3 Speech History Sub-Tab

On the speech history sub-tab, input data is displayed in the form of the message inputted, the topic it was sent to, and the time it was sent, with the most recent messages displayed at the top. A repeat button is provided with every message, which sends the original message to TIAGo using the same topic the selected message did originally. Speech history is updated anytime a speech message is sent through the system to TIAGo successfully, either via the manual input using the manual speech sub-tab, selected via the pre-defined speech sub-tab, or repeated via the speech history sub-tab.

4.3.9 Emergency Stop

The emergency stop will stop sending movement inputs from the client and cancel any play motions activated by the system. It will come in the form of a button. Due to the importance of the button, it is available in the general area of the page shown in Figure 4.4.

4.3.10 Notification

Throughout the usage of the system, users will be kept in the loop about the outcome of their inputs so that they can adjust what they do accordingly if need be. Examples would include the client connecting or disconnecting from the server, whether a predefined input was sent to TIAGo, if a joystick input fails and other akin functionality. Notifications are displayed for a short time over the right side of the screen, placed in order of the oldest notification, from top to bottom. They can be viewed in the notification area of the page, as shown in Figure 4.4.

4.4 Base Movement

The approaches used here are based on shared autonomy, aimed at providing a better user experience by abstracting certain processes to reduce how much they need to focus on. Here, we aim to reduce the need to focus on the pinpoint accuracy of joystick inputs and the safety of the environment around the robot they are operating.

4.4.1 Safe Base Movement

As mentioned above in Subsection 2.1.2, TIAGo has several modular components, which include sensors like a laser range-finder and an RGB-D camera. To help meet requirement **FR6**, after a movement input is taken from the joystick, the system will use readings from sensors onboard TIAGo to determine whether that given input is unsafe. An unsafe input in this case causes TIAGo to collide with another object. Using this, the server will then determine whether to allow that input, disallowing any manual movement inputs that are deemed unsafe.

4.4.2 Steady Base Movement

Aimed at requirement **FR8**, if a movement input given is similar to the previous movement inputs, an average of the new movement input and previous movement inputs will be used in place of the new movement input when requesting TIAGo to move in a given direction. This will avoid jittery movement caused by an unstable human hand when using the joystick, and result in the smooth and steady movement of TIAGo, while still allowing for changes in direction.

4.5 Security

4.5.1 Sessions

Traditionally, sessions are used to authenticate a client browser requesting services from a server. In this system, sessions will be used to ensure that only one client can connect to a given server at once. Under the assumption that each server is connected to a unique TIAGo, this will allow the system to meet the safety requirement **FR7**. This will work by having the server only distribute out one session token at a given time and using the notion of the client returning the token to have the server destroy it or having the session token destroyed after a given period, before distributing a new session token.

4.5.2 Cross-origin resource sharing (CORs)

As mentioned in the specification, CORs is used in this project on the backend server to allow the client browser to interact with it even though they're on different origins on the same domain. The reason for using this over SOP, which is deemed as the more secure protocol, is to enable the use of a separate client and server running on the same origin, allowing the system to take advantage of the different features of the React and Python environments listed in Section 3.2.

Though CORs has been used, in the case of this project, it can be deemed secure enough. The first reason for this system will be run over a LAN connection, and as CORs pushes security onto its domain, the LAN that the system is running on will provide more security than the less secure WAN, better ensuring that only authorised users can access the server, thereby reducing the potential for malicious attacks and protecting confidentiality. The second is by explicitly requiring all requests sent to the server via the HTTP POST method. Though slower than using HTTP GET, it reduces the chance of accidental calls to the server via other processes in the same domain, thereby reducing the potential for accidental attacks and protecting integrity. The third is by using sessions, which requires the client to have been recently authenticated by the server before accessing services, which reduces the potential for both accidental and malicious attacks, thereby protecting integrity and confidentiality. Given these conditions, for its use cases, this system should be secure enough to not require using SOP.

4.6 Third-Party Content

This project uses some third-party content to help meet the requirements in the allotted time and allow me to focus development efforts on other system components.

4.6.1 Rospy

As mentioned in Subsubsection 3.2.3.2, "Rospy is a pure Python client library for ROS. The rospy client API enables Python programmers to quickly interface with ROS Topics, Services, and Parameters" [31], being an open-source application that allows for the quick and simple implementation of communication with roscore. This is used on the server side to communicate with the TIAGo robot, with the functionality being spoken about above in Section 4.1.
Used under the BSD License.

4.6.2 Rostopic

The rostopic Python package gives live information about rostopics, such as publishers and subscribers [36]. In this project, it is a part of the process to get a list of rostopics available on the ROS master.
Used under the BSD License.

4.6.3 Rosgraph

The rosgraph Python package gives access to the currently running ROS master and performs low-level functions [37]. In this project, it is a part of the process to get a list of rostopics available on the ROS master.
Used under the BSD License.

4.6.4 BCrypt

BCrypt is a Python library used to hash input strings using a cryptographic hash, typically passwords [38]. Cryptographic hashes are one-way functions that generate a unique fixed-sized output for an arbitrarily sized input [39]. This will be used in the process of generating a session token for this system, with the functionality of the session token being spoken about above in Subsection 4.5.1
Used under the Apache Software License (Apache-2.0).

4.6.5 React-Joystick-Component

The react joystick component is an open-source React component that displays a functional joystick in the UI, supporting several typical joystick events such as movement [40]. It can be used in place of a self-made joystick to get user input. This is used on the client side of the application to enable the user to provide manual movement inputs to TIAGo, with the functionality being spoken about above in Subsection 4.3.6.
Used under the MIT License.

4.6.6 React-Select

The react-select component is an open-source React component that displays a selector input with search functionality [41]. Akin to the typical HTML selector input, it allows users to input a set of options but adds in the additional functionality of live search of the options. This is used on the client side of the application to provide a simpler and more intuitive way for users to select options in pre-defined lists, with an example of use being to select a rostopic, with the functionality being mentioned above in Subsection 4.3.4.
Used under the MIT License.

4.6.7 React-Toastify

The react-toastify component is an open-source React component that displays notifications [42]. It can be used to display non-invasive notifications of different styles depending on the type of notification on the UI for the user to see. This is used on the client side to provide the user with feedback on their inputs, with the functionality being talked about above in Subsection 4.3.10.
Used under the MIT License.

Chapter 5

Implementation and Testing

Based on the design, the system has been developed into a functional application. This has involved the incorporation of software development processes, to ensure that the system is functional at the time of development, and remains functional in the future, even if adapted or extended. Beyond this, we'll cover the project structure, alongside getting into the details of how each bit of functionality has been brought to life.

From this point onward, any figures starting with a letter can be found in the appendix.

5.1 Project Management Methodology

Due to the time constraints, an agile approach was taken when managing and developing this project. This enabled the system to be built up in parts in a short amount of time, focusing on the core feature set and requirements first, helping to ensure a working solution was developed. With this project's primary goals focusing on the development of features for TIAGo, it left much room for extension including adding support for other ROS robots alongside additional control operations outside of the initial proposal. Due to this large potential scope, managing the project in such a manner allowed for the flexible extension of the system using modular programming to better meet the system's initial requirements while also enabling it to become a better solution to the initial problem proposed in Section 1.1. Alongside this, to aid in making incremental changes and safeguard against problematic changes, version control is being incorporated through Git.

5.2 Code Organisation

The source folder contains 2 folders, the **client and server folders**.

5.2.1 Client Code Organisation

The **client folder** is the source for the client side of the application, which is written in JavaScript and Cascading Style Sheets (CSS) using React. Its file structure is as shown below in Figure 5.1.

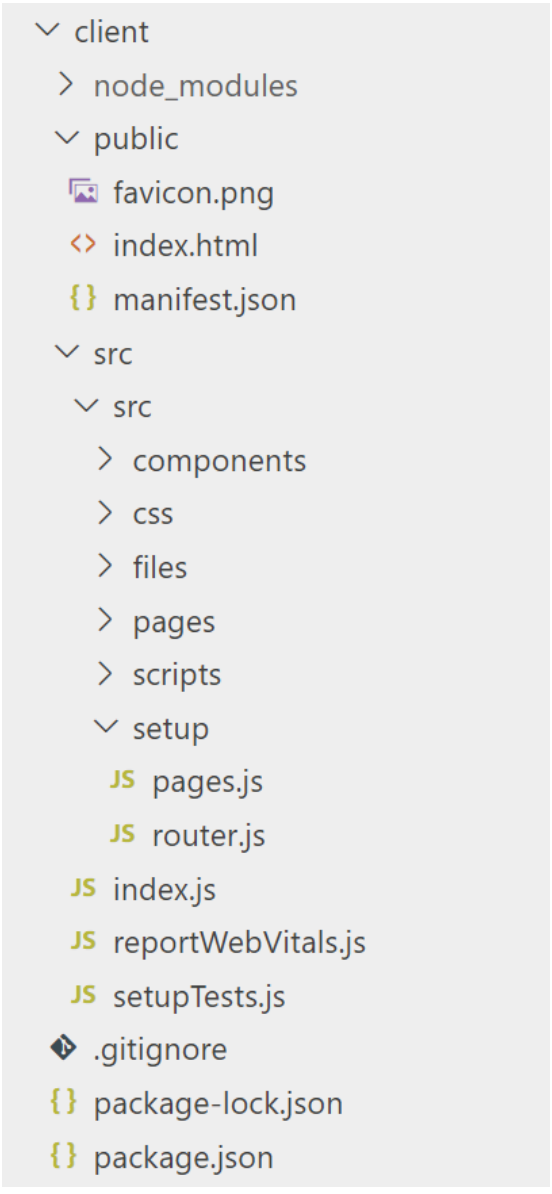

Figure 5.1: Client Folder Structure

.gitignore file: Stores instructions to not save locally generated files in the **client folder** onto the repository.

public folder: Contains files that are publicly accessible over the network from the project.

node_modules folder: Contains packages imported via npm, with exact package versions stored in **package-lock.json**.

public folder: Contains files that are publicly accessible over the network from the project.

outer src folder: Contains the inner **src** folder and **index.js** file which is run on startup.

inner src folder: The bulk of the client's code, containing the **components** folder, **css** folder, **files** folder, **pages** folder, **scripts** folder and **setup** folder.

components folder: Contains files that store different React components. React components are modular visual units and can have multiple instances on the same web page.

css folder: Contains the CSS files for the client. CSS is a way to modify the look of a website.

files folder: Contains non-code files used for stored data.

pages folder: Contains files that store the visuals to be loaded at each sub-directory. An example of a sub-directory is protocol://sub-domain.domain/**sub-directory**.

scripts folder: Contains files that store processing and logic code.

setup folder: Contains files that define constants and classes used to setup and load the web pages.

test files: Each **.js** file is accompanied with a **.test.js** file in the same directory, containing the unit tests for that file.

5.2.2 Server Code Organisation

The **server** folder is the source for the server side of the application, written in Python. Its file structure is as shown below in Figure 5.2.

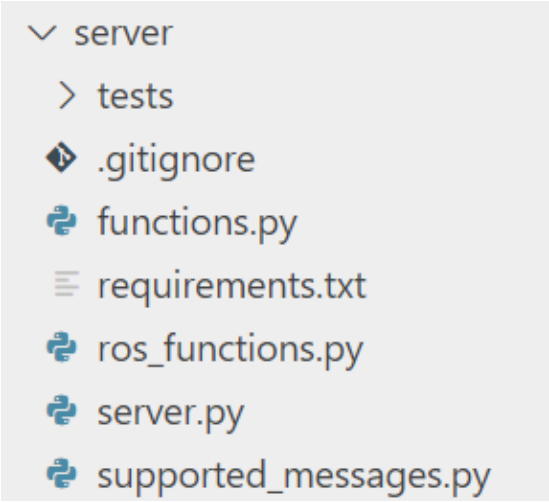

Figure 5.2: Server Folder Structure

.gitignore file: Stores instructions to not save locally generated files in the **server folder** onto the repository.

requirements.txt: Stores all packages and versions installed in Python3 using the PIP package manager.

server.py: Stores code directly related to running the server, including the server's main function that starts the server, sub-address routing, and routing decorator functions. A routing function accepts requests sent to a set sub-address, generating, and returning a response for requests. A routing decorator function pre-processes requests before passing them to a routing function.

ros_functions.py: Stores code that communicates with the ROS network.

functions.py: Stores code that parses data or performs arithmetic or logical operations.

supported_messages.py: Stores imported message types supported by the system.

tests: Contains the unit test files for the server.

5.2.3 Naming Conventions

5.2.3.1 Server

On the server, the **snake_case** naming convention was used. **CONSTANTS** are denoted in full capitals. Functions or variables that aren't used in other files than the one it is defined in, known as **_private_member_values**, are represented with an underscore at the start.

5.2.3.2 Client

On the client, the **camelCase** naming convention was used.

5.3 System Functionality

5.3.1 Cookies

5.3.1.1 Accessing Cookies

As spoken about in Subsection 4.5.1, a session token will be used to verify a client, alongside the action ID spoken about in Subsection 5.3.14, which requires client side storage. This is implemented through cookies, or more specifically, session storage. Traditionally, cookies are stored browser-wide, while session storage is stored per tab. This means that only one browser tab can be connected to the server at a given time, ensuring that a user doesn't try to input multiple commands simultaneously via different browser tabs.

5.3.1.2 Cookie Policy

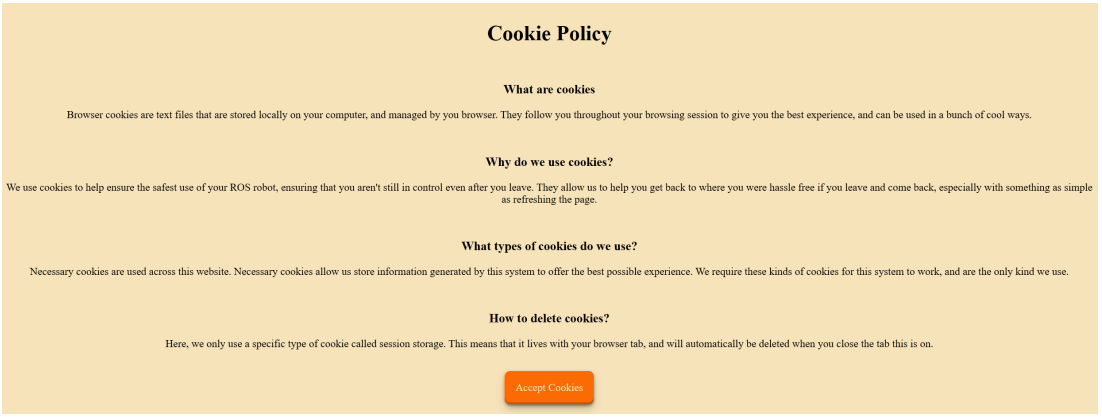

Figure 5.3: Webpage Cookie Policy

As mentioned in Subsection 4.3.2, in the aim to comply with ISO PECR, a cookie policy is presented to the user when accessing the client and requires them to accept before proceeding onto the rest of the application. This is done via checking for a *cookiesAccepted* session cookie, displaying the cookie policy when this is not found, and displaying the rest of the application when it is, using React states. React states enable UI components to change based on variable values. On accepting the policy, the *cookiesAccepted* session cookie is stored, therefore not showing the policy again while the cookie is stored. This can be seen in Figure 5.3.

5.3.2 Client Server Communication

5.3.2.1 Receiving Requests

As mentioned in Subsection 4.5.2, the client and server of this system will be communicating via the POST HTTP protocol to send and receive requests from the client and server to help maintain security, due to the need to use CORs. This is enforced on the server side by only allowing routing functions to accept requests sent via the POST protocol, thereby automatically rejecting any requests sent via any other protocol, most commonly, the HTTP GET protocol. Routing functions maintain the same definition given in Subsection 5.2.2.

5.3.2.2 Sending Requests

To enable the client to interact with the server by sending requests and receiving responses, the *sendRequest* function shown in Figure C.43 was made. It configures a request payload with the inputted data to send to the server at an inputted address, sends the request and waits for a response, returning a response once received. A max timeout of 750 milliseconds is set so that if the network connection is poor or the server takes too long to respond, the request is treated as aborted, to ensure a predictable control experience. This number was chosen after undergoing rigorous manual testing in multiple network conditions, to be assumed as the upper bound of time which all functionality on the system can handle being delayed without disrupting the control experience.

The *sendRequestWithError* function shown in Figure C.42, which uses the *sendRequest* function, parses the response generated from the request, to check whether it could successfully connect to the server, returning the response alongside an appropriate error message, which can be displayed to the user if needed. Connection in this case means whether the request was granted access to the requested services after it successfully arrived at the server, with the case of the request not reaching the server being handled by there being no response. This function also handles adding the client's copy of the session key to the request data before it is fed into the *sendRequest* function and storing the response's updated copy of the session key on the client, where required. It does this using browser cookies to interact with the stored session key, using the *getCookie* and *setCookie* functions.

5.3.2.3 Sessions

5.3.2.3.1 Defining Sessions

As mentioned in Subsubsection 4.5.1, only one session on the server can be active at any given time. This is achieved using global variables to store information on the current session, and global constants to store data relating to the rules that determine an active session, as seen in Figure C.1. Some key global values are the *session_key* variable, which stores the active key, the *session_opened_time* variable, which stores the timestamp the current session was last accessed, and the *SESSION_MAX_LIMIT_SECS* constant, which represents the maximum time a session is active for between accesses. An active session is a session where less than *SESSION_MAX_LIMIT_SECS* seconds have passed since the session was last accessed. Due to the asynchronous nature of client requests, when a new session is opened, the previous key, otherwise referred to as the handover key, will still be accepted to access services for a short time.

The *valid_session* function shown in Figure C.2 determines whether a session key allows access to server services. Outside of the session being active, a key may also be deemed valid if either the current server session is inactive, or the inputted key is the handover key shortly after a new session was started. In each of these cases, the function will return the current session data, alongside if that session is valid or invalid. If the key was active, it updates the active key's last accessed time, and removes the handover key, assuming the client has successfully started using the new key. If the current session is inactive, a new key is generated using the *generate_key* function, which is used to update the current active key, setting the previous key as the handover key, and updating the active session's last accessed time. If the handover key is used or an invalid key is provided, no changes are made to the information of the active session returned.

As shown in Figure C.3, the *generate_key* function generates a key based on using a collision-resistant hashing algorithm on the current time concatenated with a randomly generated string, called a salt. Time is used to ensure consecutively generated keys are different, given collision-resistance holds up, which along with the salt, makes it difficult to guess, without having to perform arduous calculations to generate the key.

5.3.2.3.2 Enforcing Sessions

Sessions are enforced on the server using the *session_required* routing decorator function shown in Figure C.4. Routing decorator functions maintain the same definition given in Subsec-

tion 5.2.2. For routing functions where it is attached, it pre-processes the request, using the *valid_session* function to determine whether the provided key allows access to the server's services, alongside updating the server's session details with the updated session data returned. If the request can be correctly associated with a session, then the response containing a boolean flag *connected* with the value *True* is passed onto the routing function it was addressed to, otherwise, a response containing a boolean flag *connected* with the value *False* is sent back straight back to the client, bypassing the addressed routing function. The *connected* flag is used to let the client know whether it was able to access the address it sent the request to. As a result of how the *valid_session* function works, the *session_required* routing decorator function can validate and start new sessions.

5.3.3 Connection Management

Based on the sessions mentioned above, users can connect and disconnect from the server. Separate *connect* and *disconnect* routing functions, alongside the *session_required* routing decorator function, were made for this purpose.

As shown below in Figures 5.4 and 5.5, the client enables users to try to manually connect and disconnect from a server, with the server address provided in a text field that users can change at any time. On load, the default address provided will be the address used to access the client swapping out the client's port to the server's port, 5000. When connecting and disconnecting from the server using these buttons, notifications will be displayed to notify the user of the outcome.

5.3.3.1 Connect

5.3.3.1.1 Server

The *connect* routing function shown in Figure C.4 is used to try to either connect a client to the server or extend existing sessions. It does this by using the *session_required* routing decorator, and then forwarding the response it passes to the *connect* routing function onto the client with no changes. This works as the decorator already can validate and start new sessions and as such, requires a routing path to give access to the logic used. As a result, all routing functions using the decorator enable the client to connect to the server and bypass the *connect* routing function, which is useful in cases where sessions become inactive between inputs, to reduce the number of requests that need to be sent to the server.

5.3.3.1.2 Client

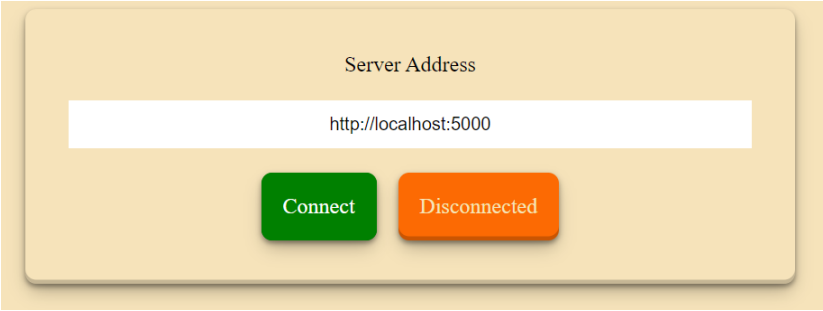

Figure 5.4: Webpage Connect Button

As shown in Figure C.44, the *connectServer* function makes use of the *sendRequestWithError* function to send a request to connect to the server at the connect address and then parses the response provided to generate an output message. The output message is then displayed as a notification. As shown in Figure 5.4, alongside the notification, the connect button turns green to inform users that they're connected to the server.

5.3.3.2 Disconnect

5.3.3.2.1 Server

The *disconnect* routing function shown in Figure C.6 is used to try to disconnect a client from the server. It does this by checking that the client is currently connected using the *session_required* routing decorator, then resetting the session variables to their default values, and sending the response with an additional boolean flag *disconnected* with the value *True* to the client. The *disconnected* flag lets the client know whether it disconnected from the server. Clients are also effectively disconnected from the server when their session key becomes inactive, with the session key being overwritten on the server when a new session is started.

5.3.3.2.2 Client

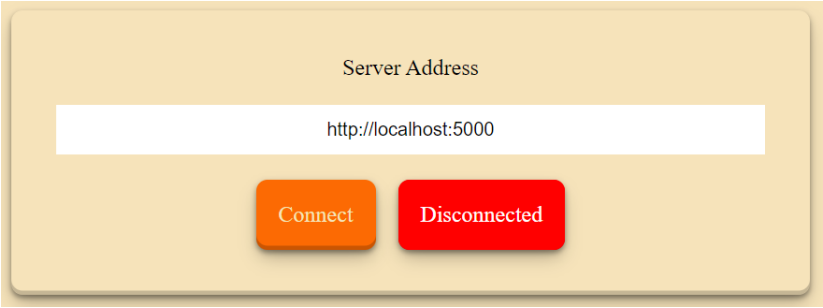

Figure 5.5: Webpage Disconnect Button

As shown in Figure C.45, the *disonnectServer* function makes use of the *sendRequestWithError* function to send a request to disconnect from the server at the disconnect address, delete the stored session key using the *deleteCookie* function, and parse the response provided to generate an output message. The output message is then displayed as a notification. As shown in Figure 5.5, alongside the notification, the disconnect button turned red to inform users that they're disconnected from the server.

5.3.4 Rostopic Selection

5.3.4.1 Finding Available Rostopics

The server can compute an up-to-date list of available rostopics using the rostopic and rosgraph libraries. As shown in Figure C.12, we initialise some global values, including the constant *_MASTER*, which stores the master on which rostopics are stored, obtained via the rosgraph library, and the variable *_rostopic_list*, which represents the most up to date list of rostopics known to the server.

The *update_rostopic_list* function shown in Figure C.8 is used to update the global list of known rostopics. It does this by getting a multidimensional list of the different publisher and subscriber topics on the master, via the rostopic library, then combining and flattening those lists using the *flatten_iterables_to_list* function shown in Figure C.11, and finally removing duplicate entries using the *unique_list* function shown in Figure C.10. This gives the list of currently available rostopics.

5.3.4.2 Selecting Rostopics

Figure 5.6: Webpage Topic Selection Field Default

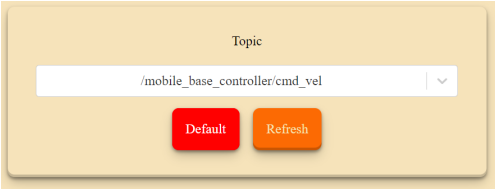

Figure 5.7: Webpage Topic Selection Field Not Default

For any given input, a rostopic can be selected for that input to be written to. As shown in Figure 5.6 and mentioned in Subsection 4.3.4, a searchable option selection field is provided, with the selected option displayed. At the time of input, the currently selected rostopic is used. Furthermore, each main tab mentioned in Subsection 4.3.5 is associated with a default rostopic. When on a tab, if the default rostopic is not selected, the default button turns red to inform the user, as shown in Figure 5.7. This enables the user to select whatever rostopic they want to write to, while not forcing them to keep track of the default rostopics for each tab. When switching between tabs, if the default rostopic for the previous tab was selected, then the default rostopic of the new tab is automatically selected. This removes the need to remember to change rostopics when switching tabs if using the default rostopic while enabling the user to select custom rostopics, and not requiring them to change back to the custom topic when switching tabs.

5.3.4.3 Requesting Available Rostopics

5.3.4.3.1 Server

The *rostopic_list* routing function shown in Figure C.7 sends a response with the flag *rostopicList* containing the list of currently available rostopics. It does this by using the *get_rostopic_list* function shown in Figure C.9, which updates the current copy of *_rostopic_list* using the *update_rostopic_list* function, and then returns this updated list. The *get_rostopic_list* function and the *update_rostopic_list* function are separated to enable the opportunity to get the server's *_rostopic_list* variable without updating the list through the use of a boolean parameter, which can take an unknown amount of time due to needing to communicate with the ROS master.

5.3.4.3.2 Client

The user can update the client's selection of rostopics by pressing the refresh button. This

indirectly calls the *getRostopicList* function shown in Figure C.46. This sends a request to the server at the rostopic list address, to get an up-to-date copy of available rostopics, and then parses the response sent back to generate an output message, returning the list and the output message. This list will then be used to update the option list, with the output message displayed as a notification. When the currently selected topic is not in the updated list, the first item in the list is selected by default.

5.3.5 Tabs

5.3.5.1 Tab Structure

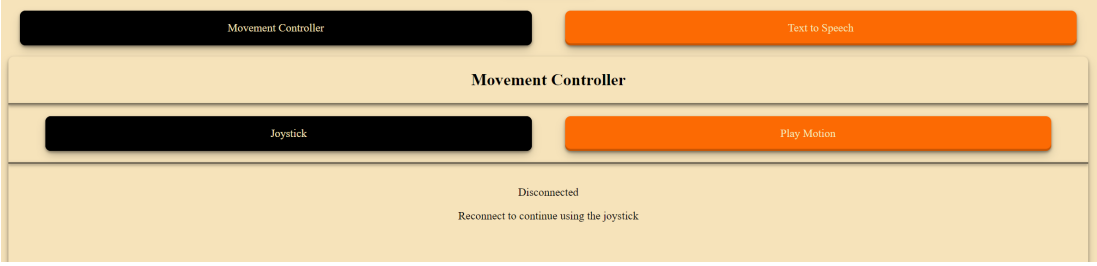

Figure 5.8: Webpage Joystick Tab Disconnected

Tabs are structured as described in Figure 4.4, with the main tab selectors at the top in the general area, and the content of that main tab below that in the tab area. The layout of the main tab content is the sub-tab selectors at the top and the content of the sub-tab below that. The tab area adjusts to the size of its contents, to signify what is part of the tab, while keeping the visuals of empty space to a minimum.

When connected to the server, the contents of the sub-tab will display as normal, allowing users to provide inputs that communicate with the server. As shown in Figure 5.8, when disconnected, the inputs in that tab are replaced with a message prompting the user to connect to the server.

5.3.5.2 Tab Selection

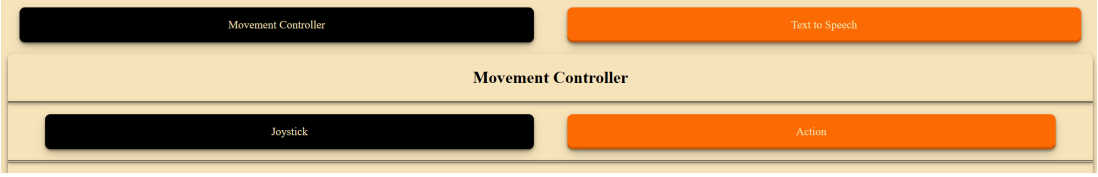

Figure 5.9: Webpage Movement Controller Tab Selection

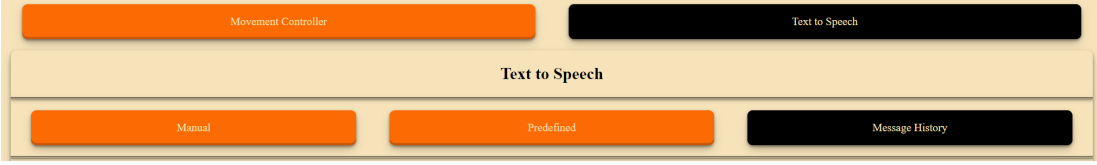

Figure 5.10: Webpage Text to Speech Tab Selection

Tab selection is operated via Cascading Style Sheets (CSS), which is used to design the layout of elements on a web page. Tab selection works by initially not displaying any tabs, and then displaying only the selected tab. Each time a tab is selected, the previous tab stops being displayed, and the new tab is displayed. In the case of both the main tabs and sub tabs as seen in Figures 5.9 and 5.10, each tab selector is a button, with the button for the currently selected tab being highlighted using a different colour.

5.3.6 Supported ROS Messages

To enable the system to support a variety of ROS messages while enforcing system security, the server has a defined set of supported ROS messages which are imported into *supported_messages.py*. This implementation is easily extendable to all ROS modules. All functionality described below can be seen in Figure C.16.

The *is_supported_ros_messages* function is used to check if a ROS message module is supported, based on its string name. This is done by obtaining a list of the names of the supported ROS message modules using the *get_supported_ros_messages* function and checking if the message module name given is in the list.

The *get_supported_ros_messages* function retrieves and returns the list of supported module names by getting a list of the names of all modules in the *supported_messages.py* file and filtering out all default modules made by Python.

The *get_ros_module* function gets and returns a supported ROS module using its name. This is done by checking if the module is supported using the *is_supported_ros_messages* function, then getting the module from the list of system modules via its name attribute.

5.3.7 Formatting ROS Message Data

ROS message data, the data in the input ROS message, is created as an object on client. When sent to the server in a request, it is converted into a Python dictionary. The dictionary's key represents the attribute, with sub-attributes represented by a **full.stop**, and the value represents the value stored at that attribute. Once an object of the ROS module class defined in the request is created using the *get_ros_module* function, this message data is loaded into the object. As shown in Figure C.30, this is done using the *dictionary_fill_object* function. Due to multiple levels of attributes, each key is split by the full stop, and traversed down the object's attribute tree, with the value assigned at the last sub-attribute in the key. Doing this for every key allows the ROS message data in the request to be parsed and converted into the desired ROS message.

5.3.8 Publisher

The server's *publish* routing function shown in Figure C.27 attempts to publish a message to a topic, which is given in a request. It returns a response containing a boolean flag *published* based on whether the response data is published successfully, alongside any flags added to the response in the *session_required* decorator function. It does this by reading the data from the request, rejecting any invalid rostopics using the *get_rostopic_list* function, without updating the list of rostopics, alongside rejecting invalid message types for the requested rostopic. This is done using *get_ros_module* function to get the request's message type, and the *get_topic_message_type* function, shown in Figure C.28, to get the rostopic's message type, and then checks that they are the same. For requests that aren't rejected, its parsed data is passed onto the *ros_publish_raw_data* function, to publish the message.

As shown in Figure C.29, the *ros_publish_raw_data* function creates a message object for the inputted message class and loads it with the inputted message data using the *dictionary_fill_object* function, which then passes this data onto the *ros_publish* function, which is also shown in Figure C.29. This creates a publisher using the rospy library and publishes the message to the publisher.

5.3.9 Base Movement Publisher

Based on the *publish* routing function, the server's *base_movement_publish* routing function acts in the similar role of a routing decorator, as it takes a request, generates a response, and passes that onto the publish function, with the difference being that the request would be addressed to the *base_movement_publish* routing function.

As shown in Figure C.31, this function publishes movement requests. It performs movement safety analysis and dampening before publishing, which is explained in more detail below. Using the *safe_base_movement* function, it determines whether an inputted movement may cause a crash via a slowdown multiplier. If this is zero, the requested movement is set to zero, if not, the requested movement is dampened using the *smoothen_base_movement_input* function, with the multiplier applied to slow down movement when close to and moving toward an object. It then passes the request data to the *publish* routing function to publish the message and returns the response generated. The data type used for movement messages is *geometry_msgs/Twist*.

5.3.9.1 Crash Prevention

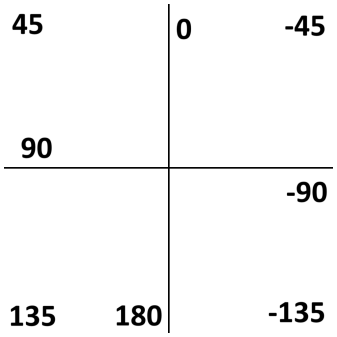

Figure 5.11: Forward Vertical Angle Axis

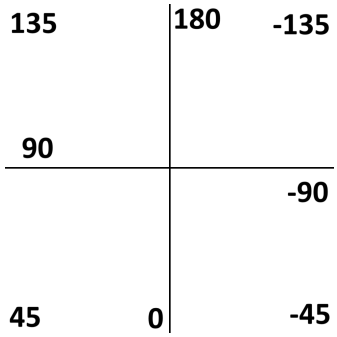

Figure 5.12: Backward Vertical Angle Axis

The system uses the Light Detection and Ranging (LIDAR) and ultrasonic sensors onboard TIAGo for crash prevention. On TIAGo, the laser sensor is at the front, while the ultrasonic sensor is at the back.

To have access to an updated copy of these scans, subscribers were setup using rospy, with a callback that updates the global variables with the current scans, which are shown in Figures C.17, and C.18. Using this scan data, in the *safe_move_base* function shown in Figure C.19, it

is determined whether a given movement would cause TIAGo to move too close to a detected object, causing the potential to crash. This is done by storing a local copy of the current global scan data to ensure the scans don't change while being checked. It then computes the angle of movement along the vertical axis, as shown in Figure 5.11, which is how laser scans calculate their angles. Using the techniques mentioned below, the *safe_move_base* function can return a decimal slowdown value between zero and one, based on the safety of the input vector. To ensure that only relatively up-to-date scans are used, scans are cleared on the server if they are older than a fixed period, via the *clear_old_scans* function shown in Figure C.23.

With the extensibility of the system in mind, a warning will be provided on the server's terminal when the server is started if either of the scan topics is unavailable and will not use the missing scan data when the *safe_move_base* function is called. At the start of this function, the scanner topics are checked for and can be subscribed to if the subscriber fails on server startup. If both scanners are missing, the *safe_move_base* function will always return true.

For both types of scans, the *slowdown_rate* function shown in Figure C.21 is used to determine an appropriate slowdown multiplier. It performs a linear equation on the closest object's distance to get a value for the multiplier. This starts slowdown at 1m and stops movement at 0.5m.

5.3.9.1.1 LIDAR Sensor

The LIDAR scanner is located at the front of TIAGo, as shown in Figure 2.1.2. There is a list of different distances in laser scans, representing the distances from the nearest object at incrementing angles within the sensor scanning arc. The server determines a range of angles to check, based on the movement's angle and a fixed angle buffer around it. The fixed buffer range is used to help better incorporate objects with thinner floor footprints into the safety calculations. It then obtains the detected object distances from the laser scan within that range and then retrieves the minimum distance from them, to represent the closest object in the movement direction. Using this, it can determine and return an appropriate slowdown multiplier.

5.3.9.1.2 Sonar Scanner

There are three sonar scanners at the back of TIAGo, as shown in Figure 2.1.2. Sonar range scans have a single value, representing the distance of the closest detected object. The angle

for the laser scans is transformed for use with the ultrasonic scans. It does this by inverting the horizontal axis of the previously calculated angle, as shown in Figure 5.12. This angle is then offset for each scanner's orientation. As this varies between robot models, and scans having no information on the scanner's orientation, the sonar scanner specific transformation offsets for TIAGo are stored in a dictionary, associated by the frame_id of that scanner's scans, as seen in Figure C.17. These angles are then compared to the ranges checked by the ultrasonic sensors, to determine if the movement's direction is in their search space. If it is, the sonar scan's value is used to determine and return an appropriate slowdown multiplier.

5.3.9.2 Smoothed Base Movement Input

To reduce inconsistency between movement inputs caused by human error when using the joystick, via the *smoothen_base_movement_input* function shown in Figure C.25, movement inputs are dampened based on the previous inputs. The *_base_movement_input_history* dictionary stores the history of inputted base movements for each rostopic, which has a maximum size, to ensure new inputs have a noticeable impact on the dampened movement. The *clear_old_history_base_movement_input* function shown in Figure C.26 removes inputs from the history that are older than a fixed period. This ensures only recent history is considered when smoothing movements.

In the *smoothen_base_movement_input* function, if there is no history for that rostopic or the current movement input is not similar to the previous movement input, the history for that rostopic is replaced with the current movement input, and the inputted movement is returned. Here, the similarity of movements is based on the movement direction's angle. Otherwise, the current movement input is added to the history for that rostopic, if over the maximum history list size, the oldest movement is removed, and the mean of that history is returned. Mean here is a mechanism to dampen the input based on the last few inputs. As the dampening is bypassed when large directional changes are made between inputs, it enables a sequence of inputs in a similar direction to have a lower variance, thereby reducing the impact of human error on base movement control.

5.3.10 Emergency Stop

Emergency stop exists to force stop all currently active movement inputs caused by that instance of the client. This includes joystick movements and play motions and is realised through

an emergency stop event.

An emergency stop button triggers an emergency stop event, of which there are two, one above the main tab selector buttons in the general area, and another in the joystick tab. Alongside that, all tab selection buttons and movement controller sub-tab selection buttons also act as emergency stop buttons, to ensure users are not trying to perform multiple tasks at once, as this can increase the chances of unsafe operation. Besides this, manually disconnecting from the server alongside closing or refreshing the page triggers an emergency stop event.

When an emergency stop event is triggered, the *emergencyStopEvent* function is called, which calls the *stopMovement* function shown in Figure C.47. This function clears all movement input intervals, sends a movement request to move with zero velocity, and cancels any current play motion actions. Movement intervals here refer to automatically repeating movement input requests. The *emergencyStopEvent* function then displays a notification message relevant to the success of the *stopMovement* function. The specifics of how to create these inputs are spoken about over the next few subsections.

5.3.11 Joystick Sub-Tab

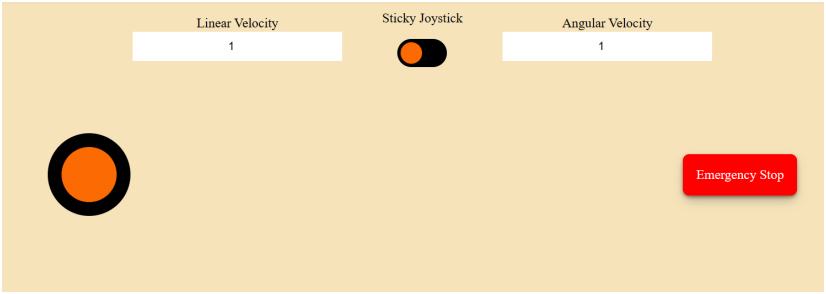

Figure 5.13: Webpage Joystick Sub-Tab

As shown in Figure 5.13, the joystick sub-tab has five elements. The linear and angular velocity number fields define the base number of wheel rotations to be used as the multiplier for the joystick vector input. The velocity values are limited between negative one and one, enforced by the HyperText Markup Language (HTML) number field and the joystick input event function mentioned below. The sticky joystick toggle is used to toggle the feature of the joystick holding in place and sending repeated inputs, even when not being held by the user. The joystick acts as the actuator for a movement input, as a continuous input with values ranging from -1 to 1

on both the horizontal and vertical axis, with the vertical axis representing linear velocity, and the horizontal axis representing angular velocity. The emergency stop button is as described in Subsection 5.3.10.

The decision to offset the joystick and emergency stop button was made with conventional controller design and handheld ergonomics in mind. By offsetting them, it should make operation more comfortable for the user and reduce the time taken to react to changing conditions, due to the clear separation of tasks. The other elements were moved up and out of the way to avoid accidental inputs during operations and increase reaction time, by reducing visually available inputs. An option to swap the positions of offset inputs was considered for accessibility but was avoided so as not to clutter the screen on smaller devices, and likely unnecessary based on most modern controller designs, such as Xbox and Nintendo Switch controllers.

5.3.11.1 Joystick

Anytime the joystick is moved, a joystick event function is called with a movement vector corresponding to the joystick movement, alongside a type, relating to the form of interaction. The start is when the joystick is first pressed. Move is when the joystick is moved. Stop is when the user lets go of the joystick. This event function first collects the current address, topic, linear and angular velocity, and sticky joystick values. It then bounds the vector velocities within a safe range and feeds it into the *joystickMoveInput* function shown in Figure C.49.

First, this function prepares the data to be sent to the server by calculating the vertical and horizontal velocities by multiplying the joystick vector input by their value multipliers. It then calls the *joystickMoveAction* function, which calls the *rosBaseMovementPublish* function to send the input to the server at the base movement publish address, and parses the output, generating and returning an output message in the process. This is passed up to the joystick event, which displays the output message to the user as a notification.

Due to how the joystick works, each movement only corresponds to one joystick event. To work around this, once the joystick vector input request is successful, it is set to be requested at a fixed interval, only when *stickyJoystick* is active, or the joystick input is not of type stop. This enables the joystick to work as intended with a sticky joystick set, alongside the user holding the joystick still in one direction. The intervals are represented as a list, as shown in

Figure C.48, to allow multiple intervals to exist simultaneously. This is required due to the asynchronous nature of these function calls, caused by the inconsistent time taken for the server to respond to a request. The intervals can be cleared using the *clearJoystickMoveActionIntervals* function, which stops all automated movement inputs. Intervals are cleared before any new joystick input request is sent in the *joystickMoveAction* function.

This subsection only describes the client side operations of the joystick's base movement inputs. The server side operations can be found in Section 5.3.9, which describes the general functionality of the *base_move_publisher* server address.

5.3.12 Search Bar

Search bars can be found on the play motion sub-tab, pre-defined speech sub-tab, and speech history sub-tab, which displays individual items in a list to the user. A text field, filter button or buttons, and reset button are provided. On pressing the filter button, the text field value will be used to filter all items in the list, only displaying those including the filtered value, case insensitive. The reset button clears the text field and displays all items in the list.

5.3.13 Text to Speech Tab

Text-to-speech inputs are made on the sub-tabs of the text-to-speech tab. The inputs are comprised of two parts, the message to be spoken, and the language it is in. The *ttsAction* function shown in Figure C.55 is called to send the message to the server. This function formats the input data and sends it to the *rosPublish* function shown in Figure C.56, which sends the input to the server in a request to the published address and parses the output, generating and returning an output message. The output message is then displayed to the user as a notification. The data type used for text-to-speech messages is *pal_interaction_msgs/TtsActionGoal*.

This subsection only describes the client side operations of text-to-speech. The server side operations can be found in Subsection 5.3.8, which describes the general functionality of the *publish* server address.

5.3.13.1 Manual Speech Sub-Tab

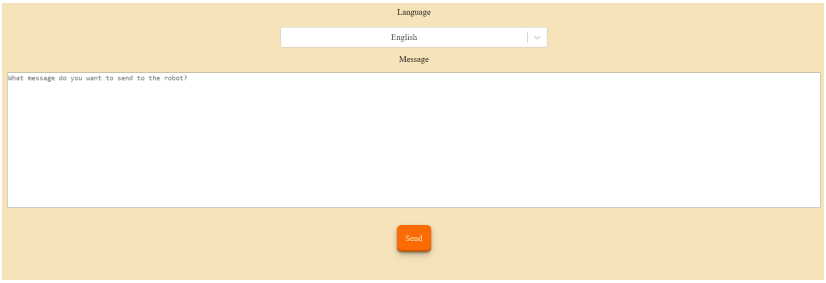

Figure 5.14: Webpage Manual Speech Sub-Tab

As shown in Figure 5.14, the manual speech sub-tab has three elements, a searchable option field for language selection, a text field for the input message and a send button. When the send button is pressed, the current language and message values from their input fields are taken and sent to the *ttsAction* function to send them as a request to the server.

5.3.13.2 Pre-defined Speech Sub-Tab

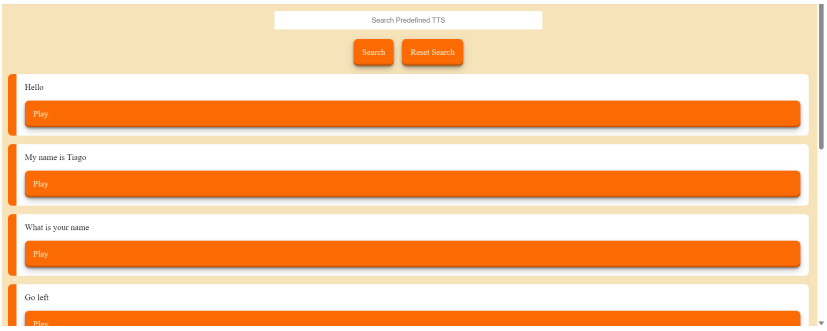

Figure 5.15: Webpage Pre-defined Speech Sub-Tab

As shown in Figure 5.15, pre-defined speech inputs are displayed as the message along with a play button, which when pressed calls the *ttsAction* function to send the message and default language to the server.

The list of pre-defined speech inputs is generated by reading the **predefinedTTS.txt** file located in the **files folder**. As shown in Figure C.57, the *readPredefinedTTSTextFile* function reads the file line by line, storing it as a list of strings. This function is called when the page is loaded, with the list then being used to display the inputs. As they're generated from a text file, the pre-defined speech inputs can be altered quickly and simply for use in different

scenarios.

5.3.13.3 Speech History Sub-Tab

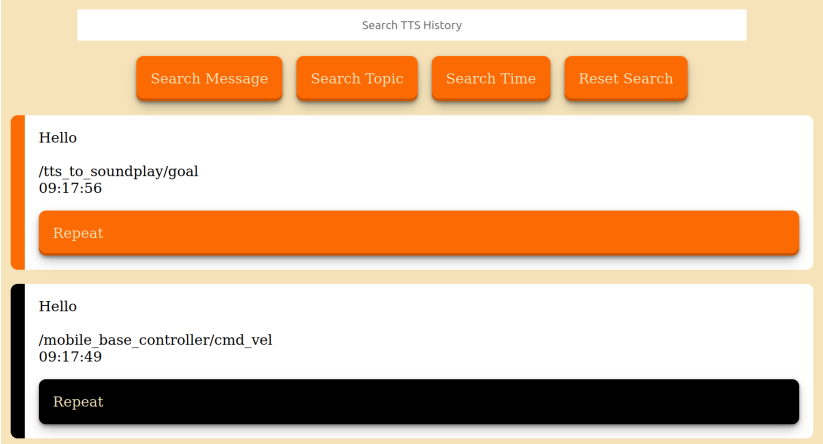

Figure 5.16: Webpage Speech History Sub-Tab

The speech history tab displays the previous successful speech requests. As shown in Figure 5.16, the messages are displayed in order of most recent, showing the message sent, the topic it was sent to, and the time it was requested. Message requests that were sent to the currently selected topic are marked in orange, with the other requests highlighted in black. Pressing the repeat button calls the *ttsAction* function to send the message to the request's original topic, irrespective of the selected topic.

5.3.14 Play Motion Sub-Tab

Figure 5.17: Webpage Play Motion Sub-Tab

The system supports control over one action at any given time. This is done via the global variable shown in Figure C.33 which stores the client for the currently active action, with a key generated to identify it. The key ensures that only the client that activated the action can stop it. This is upheld using the *validate_ros_action_id* function shown in Figure C.34, which returns a boolean based on whether the ID inputted matches the current action client's ID. When the current action ends, its action client is removed from the server via resetting the *action_client* global variable to its default value using the *clear_ros_action_client* function as shown in Figure C.35. The data type used for play motion messages is *play_motion_msgs/PlayMotionActionGoal*.

As shown in Figure 5.17 play motions are displayed as their name along with a play button. This is generated from a list, displaying the individual items. How the list of play motions is obtained is explained in Subsubsection 5.3.18.

5.3.15 Start Action

5.3.15.1 Server

The *action_start* routing function is used to create an action client and start an action, based on a request, returning a response with the *action* flag, stating whether the action was possible, and the *actionID* flag that is added if the action was started successfully. As seen in Figure C.36,

it reads the request data and checks if the current action has ended, using the *ros_action_ended* function. If the current action has ended, it gets the action class and goal class via their names using the *get_ros_module* function and calls the *ros_start_action_client_raw_data* function seen in Figure C.37, with the parsed data from the request. This then creates a goal object, and loads it with the goal data, using the *dictionary_fill_object* function. This is then passed into the *ros_start_action_client* function, which starts a simple action client, sends the goal to the client, generates an ID for the client using the *generate_key* function, and updates the global copy of the current action client. Finally, it outputs the generated action ID to add it to the response and send it to the client.

5.3.15.2 Client

By pressing the start button shown in Figure 5.17, a request to perform the given play motion is sent. Pressing it calls the *rosStartAction* function shown in Figure C.37, which sends the input as a request to the server at the action start address and parses the response to generate an output message. This function also stores the *actionID* for the play motion locally, if it is started successfully, in browser storage using the *setCookie* function. The output message is then displayed as a notification.

If the action is started successfully, the currently active play motion is set, which causes the play button to turn into a cancel button. Alongside that, an interval is started to call the *rosActionEnded* function to check if the action has ended, at a fixed rate. The interval stops itself when it finds that the action has ended, alongside clearing the currently active play motion.

5.3.16 Cancel Action

5.3.16.1 Server

The *action_cancel* routing function shown in Figure C.38 takes a request to cancel a given action and sends a response with the flag *actionCancel*, which is a boolean stating whether the action was cancelled. This is done using the *cancel_ros_action_client* function shown in Figure C.39, which first validates that the given action ID from the request matches the one on the server. If it does, it cancels the current action and clears the globally stored action client using the *clear_ros_action_client* function.

5.3.16.2 Client

Pressing the cancel button will call the *rosCancelAction* function shown in Figure C.52, which sends a request to the server at the action cancel address to stop the play motion, alongside parsing the response to generate an output message. This function will also delete the local copy of the *actionID* for the play motion, if it is stopped successfully, using the *deleteCookie* function. The output message is then displayed as a notification.

5.3.17 Action Ended

5.3.17.1 Server

The *action_ended* routing function takes a request and returns a response with the *actionEnded* flag, which is a boolean stating whether the action matching an action ID has ended. As shown in Figure C.40, this is done using the *ros_action_ended* function, by passing the action ID found in the request. As seen in Figure C.41, this function assumes that if the inputted action ID does not match the one for the client stored on the server, that action has ended. This assumption is made due to the enforcement of only one action being possible at a time, through a given server. Otherwise, the action client on the server is checked to see if the action has ended. If it has ended, the globally stored action client is cleared using the *clear_ros_action_client* function. After, it returns a boolean for whether the globally stored action client is empty.

5.3.17.2 Client

To check when a play motion has ended, the *rosActionEnded* shown in Figure C.53 can be used. Using the *actionID* as an input, it sends a request to the server at the action ended address, parsing the response, and generating an output message. Alongside the output message, this function returns a boolean for whether the action associated with the inputted *actionID* has ended, which is taken from the response.

5.3.18 Play Motion List

5.3.18.1 Server

The server can compute an up-to-date list of available play motions by using the rospy library. The *play_motion_list* routing function shown in Figure C.14 sends a response with the flag *playMotionList* containing the list of currently available play motions on the ROS master. It does this by using the *get_play_motion_list* function shown in Figure C.13, which uses the ropsy

libraries' *get_param* function to get a dictionary where the keys are all the available play motions and returns the list of keys from that dictionary.

5.3.18.2 Client

The local copy of the play motion list can be updated by pressing the update list button. When pressed, it calls the *getPlayMotionList* function shown in Figure C.54. This function sends a request to the server at the get play motion list address, to get an up-to-date copy of the play motion list and parses the message to generate an output message. The output message is then displayed as a notification.

5.3.19 Notifications

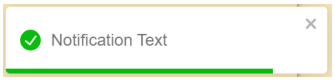

Figure 5.18: Webpage Success Notification

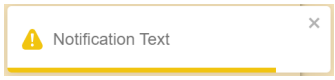

Figure 5.19: Webpage Warning Notification

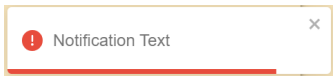

Figure 5.20: Webpage Error Notification

As described in Subsection 4.6.7, notifications throughout the client side are handled via the React-Toastify library, which has been imported into the project through npm. Each notification appears for five seconds before being dismissed but can be manually dismissed by pressing it. If multiple notifications are received, they stack below one another, unbounded by the page, bouncing up once previous notifications are dismissed. The time left before automatic dismissal is displayed as a bar at the bottom of the notification. Notifications used come in the form of success, error, and warning messages, which can be differentiated by the colour of the time bar, as shown in Figures 5.18, 5.19 and 5.20.

5.4 Changes From Design

After performing a usability test, talked about in Section 7.3, changes were made to the system to reflect the findings.

5.4.1 Control Centre Tab

Figure 5.21: Webpage Control Centre Tab

When using only the predefined inputs and default rostopics, switching between tabs can disrupt the flow of control. The control centre tab, shown in Figure 5.21, is a space where the default form of all provided inputs is available, which allows for more natural chaining of inputs, thereby increasing the freedom of control. This works by displaying instances of the React functional components for the joystick, play motion, and predefined text-to-speech sub-tabs in one place, thereby not duplicating code while maintaining the functionality from the sub-tabs. The original sub-tabs remain available for when users want to customise the rostopic they are inputting to or focus on one type of input.

5.4.2 Larger Joystick

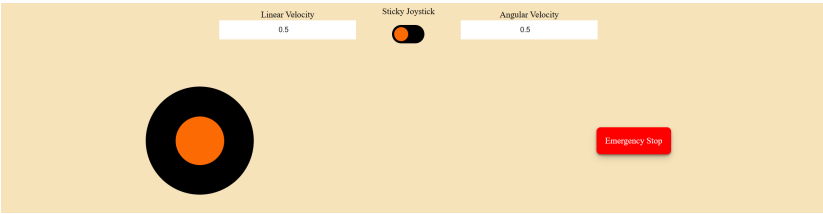

Figure 5.22: Webpage Joystick Sub-Tab with Larger Joystick

To increase the feel of control for users, the size of the joystick was increased by 33%, seen between Figures 5.13 and 5.22. This enables finer control in tighter spaces.

5.4.3 Shorter Stopping Distance

Previously, the stopping distance was 0.5m. Though this worked well for open environments, in practice it heavily limits control in tight spaces. To address this, the stopping distance has been decreased to 0.25m, with the slowdown distance increasing to 1.5m from 1m, shown in Figure C.22, to smooth the speed transition to enable more fluid control in tighter spaces.

5.4.4 Base Movement Rotation

Rotational inputs have been made to bypass the original crash prevention and movement dampening functionality, shown in Figures C.20 and C.32. Instead, inputs with forward movement angles, as shown in Figure 5.11, around ninety or negative ninety degrees are slowed down to a quarter of the speed, with the vertical component of the movement vector being nullified. A range around the angles was used to make it easier to activate while being reasonably sized so as not to remove many useful movements. Assuming TIAGo is in the home position, rotations keep TIAGo's footprint the same, and should not put it in a position to crash, based on the footprints of the surrounding objects.

5.4.5 Keyboard Events

When using the search bar or custom text-to-speech, keyboard events such as enter to search or send and tab to reset search were added.

5.5 Software Testing

5.5.1 Software Testing Methodology

Unit testing is implemented to provide a more robust and certain set of tests to ensure that the system functions according to the requirements set out in Section 3.1. Alongside this, a set of manual tests covering the full user flow shown in Figure 4.2 will be completed.

5.5.2 Client Software Testing

The client side of the system will be unit tested using the React testing library [43], which enables the testing of all the client's code, from frontend components, to request calls, to individual scripts.

5.5.3 Server Software Testing

The server side of the system will be unit tested using the Python unittest package [44], which enables testing of all the server's code, from computing server responses, to ROS communication, to logic and arithmetic.

Chapter 6

Legal, Social, Ethical and Professional Issues

When designing and developing this system, aspects outside of raw functionality were considered, to ensure that it not only meets the general aims of the project but also minimises the number of potential blockers for its use. This includes ensuring it meets legal legislation, and is socially and ethically acceptable, both in what was made, and how it was made.

6.1 British Computer Society Legislation

All parties involved in the project were aware of the standards set out in the Code of Conduct & Code of Good Practice issued by the British Computer Society and did their best to abide by them when engaging with this project.

6.2 UK Data Regulations

Within this project, the service makes use of session storage, which is a form of cookies. Though this is only used to store non-personal data about communication between the client and server, in the UK, this requires that the system is compliant with ISO policies [19], with cookies falling under PECR [20]. In the aim to abide by this, following the guidelines from the ISO alongside the UK government [45], users are required to accept the use of cookies on the client before accessing the functionality of the web page. This can be seen in Subsection 5.3.1.2. As all cookies used are required for the system's functionality, no option is provided to proceed without

accepting. This ensures that all users are readily aware of its use and accept this before using the system. Alongside this, an effort has been made to keep the use of cookies to a minimum.

6.3 Shared Autonomy

Though automation has existed for a long time, an example being a clock, autonomy is still not completely socially acceptable, especially regarding Artificial Intelligence. The biggest ethical and social problems regarding Artificial Intelligence generally come from the inability to completely explain why they make the decisions they do [46]. Even when autonomous systems are not controlled via conventional black-box Artificial Intelligence, the stigma against them still exists. Compared to fully autonomous machines, shared autonomy is generally socially accepted, being found in some of the most safety-critical systems in the world, such as car cruise control, and aeroplane autopilot. This likely comes about due to a human still being in control, with them being able to, at the very least, explain why they didn't take control. Though this is the case, no technology is perfect, and the times they fail can create negative sentiment throughout society. Considering these social and ethical circumstances, the decisions were made on what tasks shared autonomy was applied to, and what level of control to provide the system.

6.4 Third-Party Content

All third-party content used throughout this project is appropriately cited and accredited where mentioned. A collation of all third-party software used alongside the licences they are used under is provided in Subsection 4.6.

Chapter 7

Results and Evaluation

Due to the time given and the scope of the project, some limitations have been made to enable the completion of the system. Based on the initial aims and requirements of the system, here, we'll evaluate the finished product to see how successful it is at fulfilling the needs it was made for and to see if the best approach was taken.

7.1 Overview of Software Product

This system can be broken down into five functional components, the joystick for base movement, play motion actions, manual text-to-speech, pre-defined text-to-speech, and text-to-speech history. This functionality is brought together by two separate systems, the client side React application, and the server side Python application, each of which functions individually and is not directly tied to the other. This means that either part could be applied to other functionality given the adequate integration of some tertiary system.

The server side can take HTTP requests and communicate with the ROS environment to use the request data, perform a control operation, or retrieve data. It can also parse, alter, or reject requests where fit, to ensure the safe operation of the connected robot. The client side can make requests to the server to perform many operations, providing users with a UI with a variety of continuous and linear inputs, to provide a simple and intuitive control experience. Core operations include movement inputs via a virtual joystick, emergency stop via a button, starting or cancelling play motion actions via buttons, and playing pre-defined, custom, or previously used text-to-speech messages via buttons. Notifications are provided to the user based

on the outcome of their inputs.

7.2 System Limitations

As most do, this system has some limitations which were taken on to allow it to meet the requirements in the limited time for this project.

- Due to needing network requests to be sent between the server and client, the client needs to be able to correctly identify the server's Internet Protocol (IP) address and port number. If they are not running on the same IP address, this requires the user to know and manually input the server's IP address and port number. This has been addressed by having the server print its IP address and port number to the terminal on startup, which should make it easier for the user.

- As the ROS environment that the server is run in catches commands for scripts to end themselves, the only way to close the server is to force close it. This results in the port the server was on not being freed, even though the server is closed. This requires the user to manually free the port after closing the server, requiring an extra terminal command.

- The assumptions made about there being one ROS robot per ROS environment means that a single server cannot support control of multiple ROS robots at a time, though this can be worked around running multiple servers in a ROS environment.

- The assumption that each ROS robot would be associated with a single server, running multiple servers in the same ROS environment allows for multiple clients to send inputs to the same ROS robot at the same time.

- As scans have a limited range and only collect information close to the ground, it is unable to prevent crashes against objects in their blind spots, alongside being poor at preventing crashes against objects that have a small floor footprint, such as a table. This further applies when TIAGo is not in his home position.

- This system's crash prevention only applies to base movement inputs, and though play motion should check that no collisions will occur using the RGB-D camera, it leaves the user liable for safety depending on the precautions of the play motion action used.

7.3 Usability Evaluation

A usability test was performed to evaluate the effectiveness of the system.

7.3.1 Test Protocol

7.3.1.1 Objectives

The main objective of this test was to see how easy it was to use for new users, across functionalities. A secondary objective was to see how usable the system is in tight spaces.

7.3.1.2 Task Environment

Figure 7.1: Usability Test Environment

The test will occur in a makeshift closed space environment, shown in Figure 7.1, followed by interaction with a second participant.

7.3.1.3 Tasks

1. Connect to the server.
2. Drive TIAGo through the provided course.
3. Try to crash into TIAGo into the designated person at the end of the course.
4. Make TIAGo wave to the designated person.
5. Make TIAGo say hello and a message of your choosing to the designated person.
6. Disconnect from the server.

7.3.1.4 Test Measures

Two main measures will be used for this test, the time taken, the number of times assistance was required, and the feedback received from the after test questionnaire.

7.3.1.5 Participants

Two participants who had not seen or used the system were recruited.

7.3.1.6 Questionnaire

For all these questions, rank them one to five, with one being you strongly disagree and five being you strongly agree. Optional further comments were gathered after this questionnaire.

1. It was easy to drive around TIAGo.
2. It was easy to make TIAGo wave.
3. It was easy to make TIAGo speak.
4. The system was easy to use.
5. Your hands were in a natural position while operating TIAGo.
6. You felt it was a mentally taxing experience to operate TIAGo.

7.3.2 Control Results

Table 7.1: Usability Test Physical Controller Results

Participant Number	Course Time (s)	Assistance Count
1	43	0
2	54	0

7.3.3 Before Changes Results

Table 7.2: Usability Test Before Changes Results

Participant Number	Course Time (s)	Wave Time (s)	Talk Time (s)	Q1 Score	Q2 Score	Q3 Score	Q4 Score	Q5 Score	Q6 Score	Assistance Count
1	96	44	7	3	5	5	4	4	3	2
2	218	13	20	3	5	5	4	5	5	3

Looking at Table 7.2, we can see that the users generally liked the visual design and comfortability of the system. Functionally, they found it easy to control actions and speech, alongside finding the joystick mechanism intuitive, though the time taken between functionality and finding given options could be improved. Though it prevented them from crashing into another person, from the feedback, we can see that within the enclosed environment, the safety mechanisms were intrusive and prevented any sort of control, hindering movement, with one participant saying they felt stuck, and that it was hard to drive in the provided course. This is further supported by the control, seen in Table 7.1, with the time taken to navigate the course being significantly better. Some assistance was required due to the limited control in the course, alongside missing expected functionality, such as keyboard events.

7.3.4 After Changes Results

Table 7.3: Usability Test After Changes Results

Participant Number	Course Time (s)	Wave Time (s)	Talk Time (s)	Q1 Score	Q2 Score	Q3 Score	Q4 Score	Q5 Score	Q6 Score	Assistance Count
1	46	14	12	5	5	5	5	5	1	0
2	50	7	4	4	5	5	5	5	2	0

Some noticeable improvements were made in the results from Table 7.2 to 7.3 due to the changes made. The time taken to control actions and speech drastically improved. With participant one using the separated tabs, and participant two using the control centre, this overall speed up shows the success of the control centre and keyboard events. Whether due to one or more of the changes, the navigation time greatly improved as well, with the times being comparable to the control times shown in Table 7.1, while remaining to keep them from crashing into a person. They also found themselves not requiring any assistance this time around. For the feedback, it can be said that their opinion of the system improved, with one participant saying they find it more fun and less stressful to control. From this, it could be gathered that the system is in a state where it is easy to use while reducing the burden of control for even new

users.

7.4 Evaluation of Data Communication

Selection	Download speed	Upload speed	Minimum latency
GPRS	50 Kbps	20 Kbps	500
Regular 2G	250 Kbps	50 Kbps	300
Good 2G	450 Kbps	150 Kbps	150
Regular 3G	750 Kbps	250 Kbps	100
Good 3G	1.5 Mbps	750 Kbps	40
Regular 4G/LTE	4 Mbps	3 Mbps	20
DSL	2 Mbps	1 Mbps	5
Wi-Fi	30 Mbps	15 Mbps	2
Offline	0 Mbps	0 Mbps	5

Figure 7.2: Firefox Network Throttling Speeds

To evaluate the effectiveness of the chosen data communication method, some tests involving sending requests from the client to the server were performed, under different network conditions, measuring the time and request timeout rate. The results are recorded in Table 7.4. To record the response time and throttle the network, the Firefox debugging suite was used, with the corresponding speeds shown in Figure 7.2 [47]. When testing a given network address, it was setup that all expected operations would occur to test real-time performance. As set on the system, the timeout time used is 750ms.

Table 7.4: Communication Speed Metrics

Request Path	Network Throttling	Request Count	Mean Latency (ms)	Timeout Rate (%)
connect	GPRS	10	736.6	0
disconnect	GPRS	10	507.1	0
rostopic_list	GPRS	10	2557	100
rostopic_list	Regular 2G	10	322.1	0
play_motion_list	GPRS	10	516.7	0
publish	GPRS	10	512.8	0
base_movement_publish	GPRS	10	622.7	20
base_movement_publish	Regular 2G	10	312.8	0
base_movement_publish	Good 2G	10	198.8	0
base_movement_publish	4G/LTE	10	23.6	0
base_movement_publish	Wi-Fi	10	5.1	0
action_start	GPRS	10	941.3	80
action_start	Regular 2G	10	606	20
action_start	Good 2G	10	419.7	0
action_cancel	GPRS	10	513.7	0

Looking at Table 7.4, some evaluation can be made about the effectiveness of the separate client server approach for ROS environment communication. Based on the timeout rate most functionality is fully operational at the slowest GPRS speeds, with all functionalities fully operational at good 2G speeds. According to recordings taken by speedtest.net, the world's median mobile latency is 27ms and broadband latency is 9ms as of February 2024 [48]. This would mean most of the world has better than good 3G network performance on mobile and better than regular 4G/LTE network performance on broadband. This puts the minimum network requirements for full system functionality due to this communication method alongside computation below what most of the world has access to.

For most people, the average rate at which their eye perceives the world around them sits at around 30 frames per second, which translates to their view updating roughly every 33ms [49]. Based on this, when on most broadband networks, using the joystick to request at the base_movement_publish address path, which is shown to have a mean latency of 23.6ms, will result in most users being able to send more inputs than they can physically see. This means that in most situations, the system will be able to provide a truly seamless control experience.

When considering the data communication method for this system, the other standout option was using a third-party library such as roslibjs. Based on unthrottled speeds, others have found that such a solution would take 3ms to communicate between the ROS environment and the client [50]. Though the network conditions cannot be perfectly compared when throttled to Wi-Fi network performance, both communication and processing of data such as crash prevention and movement dampening for the base_movement_publish address path results in a mean of a 5.1ms delay between input and response. Due to the unknown differences in network conditions and additional data processing being included in our results, it cannot be definitively decided which communication method is fastest, but it can be argued that both are similar in terms of speed of communication. Tying this with the other benefits mentioned in Section 4.1, the chosen approach to data communication comes out as the better option for this project.

7.5 Evaluation of Control Mechanisms

7.5.1 Simplicity of Control

Across the application, a mixture of linear and continuous input types has been used. Due to the simplicity of linear inputs, they appear most commonly, with continuous inputs being used for complex operations to provide a more intuitive control experience. An example of this is the joystick. In cases such as for starting and cancelling play motion actions, inputs are intuitively brought into and removed from view, to reduce the amount of information the user must consider and reduce the amount of information the user must remember. Text has been opted for in place of symbols, to ensure that users don't struggle to understand what a given input does. Humans are shown to function best when they aren't required to draw information from memory, but instead from the task environment they are in, which can thereby increase reaction time [51]. Such design decisions help to provide a simple and safe control experience.

7.5.2 Comfortability of Control

For ease of use in landscape mode, following conventional controller design, some core inputs which may need to be used reflexively have been offset, to reduce its distance from the edge of the screen. To prove this, measurements from the closest edge of the input to the left and right edges of the screen were taken, measured as percentages of the screen width.

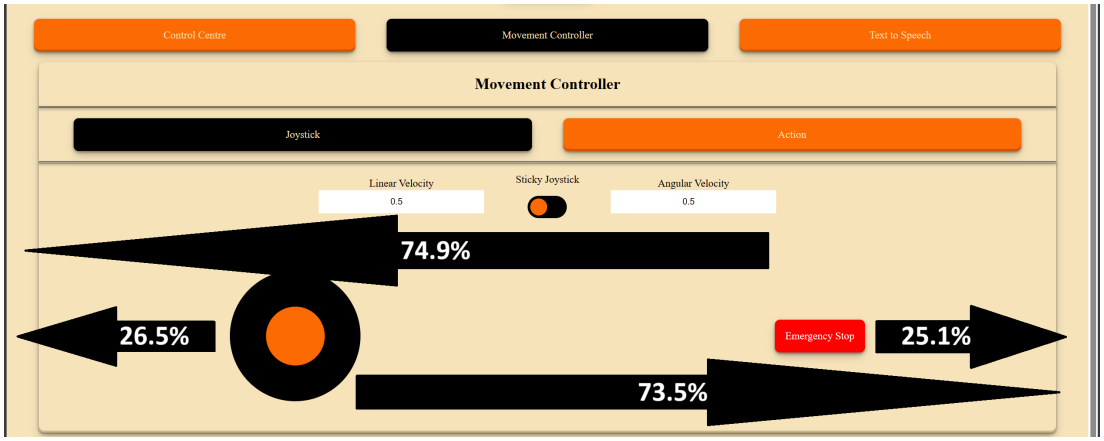

Figure 7.3: Joystick With Offset (Measurements From Input's Midpoint)

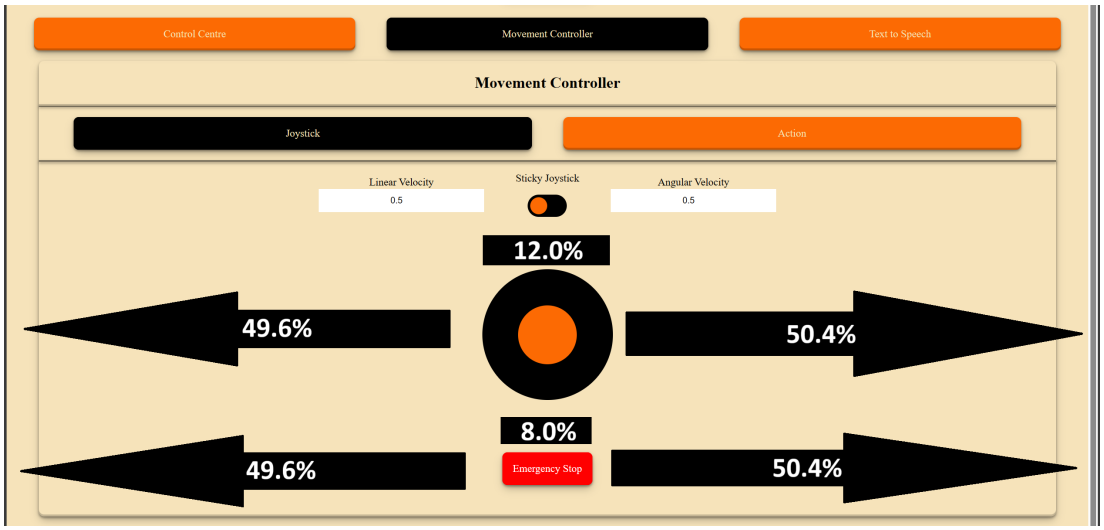

Figure 7.4: Joystick Without Offset (Measurements from Input's Midpoint)

As shown in Figures 7.3 and 7.4, offsetting the joystick and emergency stop button in landscape orientation reduces the distance to reach it by over 20% of the screen width. Doing this reduces the travel distance to interact with the inputs, alongside promoting a comfortable and simple interaction environment. This is especially useful for on-the-go devices, such as mobiles and tablets, where when in landscape orientation, the user should be able to keep each hand in place over either of the inputs, rather than forcing switching back and forth when needed. Furthermore, beyond comfort, it enables users with smaller hands to use this functionality without having to strain themselves. In cases of one-handed operation, where to make use of both inputs, the offset would force users to either stretch 70% of the screen with or swap the side their hand is on, portrait mode offers this functionality without the offset. Conventionally,

one-handed control is more likely to be used in a portrait orientation, both on mobile [52], and on tablet [53], where the device is held while in use. Design decisions around offsetting inputs have therefore helped to increase the comfort of use. Alongside this, per Fitt's Law of ergonomics and rapid movement [54], it improves safety for the robot while being controlled using the joystick, by reducing the travel distance in emergencies.

7.5.3 Feedback

Notifications are a simple but effective way that users are informed about the outcomes of their inputs. Though effective, it can lead to users not always heeding warnings. Generally, adding something distinctive, such as using different colours, helps to create more interest in the text, while also conveying the type of message being presented [55]. With notifications having an associated colour, it increases the chance feedback given is noticed.

Beyond notifications, depending on the outcome, other changes are made to the interface to provide feedback. Examples of this include the colouring of the connected or disconnected button when connected or disconnected, the colouring of the default rostopic button when the default rostopic is not selected, the highlighting of selected tabs and sub-tabs, and the removal of control elements from the view when disconnected. Through multiple mediums, feedback can aid the user through control, keeping them informed, and best able to make real-time decisions. As a result, these feedback mechanisms help promote the safe control of the robot.

7.6 Evaluation of Safety and Simplicity Measures

All results obtained in this section were recorded in a simulated environment. This environment can be found using the command ***roslaunch tiago_gazebo tiago_gazebo.launch public_sim:=true end_effector:=pal-gripper ft_sensor:=false world:=tutorial_office***, in a terminal where the TIAGo container has been loaded.

7.6.1 Crash Prevention

To evaluate the effectiveness of crash prevention, some tests moving towards different object types with different input vectors were done using the joystick, to see how the system performed in different scenarios, measuring the crash and slow-down performance across these inputs. The results are recorded in Table 7.5, with the angle represented as shown in Figure 5.11. All objects

tested were stationary.

Table 7.5: Crash Prevention Metrics

Object	Approx Forward Vertical Vector Angle (±°)	Velocity (m/s)	Times Crashed	Times Slowed Down	Crash Rate (%)
Wall	0	1	0/5	5/5	0
Wall	45	1	0/5	5/5	0
Wall	180	1	0/5	5/5	0
Person	0	1	0/5	5/5	0
Person	45	1	0/5	5/5	0
Person	180	1	0/5	5/5	0
Table Leg	0	1	0/5	5/5	0
Table Leg	45	1	3/5	5/5	60
Table Leg	45	0.5	2/5	5/5	40
Table Leg	180	1	0/5	5/5	0
Table Top	0	0.5	5/5	0/5	100
Table Top	45	0.5	5/5	0/5	100
Table Top	180	0.5	5/5	0/5	100

Looking at Table 7.5, a few points can be noticed. Against surface surfaces of different sizes, when moving approximately in a cardinal direction at the fastest supported speed, the system stops the robot from crashing in all attempts. When against a surface with a small footprint, if moving approximately in a non-cardinal direction, the system struggles to stop the robot before it crashes. In all cases mentioned, even when the robot isn't completely stopped, it slows it down. This shows that in all cases with stationary objects where the floor footprint is present, some level of crash prevention is provided. As expected, in cases where no floor footprint is present, neither slowing down nor stopping the robot occurred, regardless of the movement direction.

When it doesn't completely stop but slows down, it reduces the chance of damage from a collision, alongside providing more time for the user to react to stop the collision from occurring. In cases like this, the collision likely occurs due to the changing position of the sensor, which depending on the robot's orientation, speed, and approach, can result in the sensor not being the closest point to the object, causing the reading to be slightly off from reality. A solution would be to increase the stopping distance and angle range checked. In practice, it could reduce freedom of operability for the user in tight spaces, thereby reducing functionality, hence some balance had to be achieved.

7.6.2 Smooth Base Movement

To evaluate the effectiveness of base movement smoothing, some tests involving changing direction over different timescales were done using the joystick, with the average differences between inputs being measured. By using the joystick, results may not trend perfectly but will show experimental results rather than theoretical results. The results are recorded in Table 7.6.

Table 7.6: Base Movement Smoothing Metrics

Approx Movement Angle Range (±°)	Approx Transition Time (s)	No Dampening Mean Difference (%)	With Dampening Mean Difference (%)	Improvement Ratio (%)
0 to 45	2	61	24	154
0 to 45	5	49	30	63
0 to 90	2	60	46	30
0 to 90	5	60	23	161
0 to 135	2	30	30	0
0 to 135	5	266	80	232
0 to 180	2	193	193	0
0 to 180	5	33	21	57

Even after using the joystick for the test, some general trends can be derived from the data in Table 7.6. To start with, excluding the 0 to 45 degree range, all ranges were able to reduce the average difference between movement vectors given more time transitioning between movement directions. This shows that the effect of small imperfections is being reduced, while still allowing for larger sweeping movements. This is well represented via the 0 to 135 and 0 to 180 degree angle ranges at 2 seconds, where the movements were too large for any dampening to occur. Going back to the 0 to 45 degree range, though the data shows that the dampening improvement was larger with less time, it also shows that the inputs provided were more stable, which likely comes down to user error, demonstrating exactly why such a feature was included. From looking at the data, there are no points at which the movement vectors become more sporadic, and along with the improvements it has brought, some level of base movement input smoothing has been achieved.

7.7 Requirements Evaluation

Here, requirements are matched up with the parts of the application through which they were met.

7.7.1 Functional Requirements Evaluation

- **FR1** - The system must provide a way to communicate with TIAGo via the web interface: Using CORs requests and the client's server text field, the client can communicate with the server, and through the use of the rospy library, the server can communicate with TIAGo, enabling the user to communicate with TIAGo through the client's web interface. More details are provided in Section 7.4.

- **FR2** - The system must allow users to control the linear and angular movement of TIAGo's base: Users can control the linear and angular movement of TIAGo's base using the client's joystick alongside the linear and angular velocity text fields to dictate its velocity, and through the topic selection field, can choose the relevant topic to send it to.

- **FR3** - The system must allow users to send custom messages for TIAGo's to say: Users can create custom messages through the language selection field and message text field and can send it to TIAGo to say using the topic selection field and send button.

- **FR4** - The system must provide a set of pre-defined movements to select from for TIAGo to perform: Pre-defined movement inputs are provided in the form of play motions and can be selected to start through the play button and stopped using the cancel button.

- **FR5** - The system must provide a set of pre-defined messages to select from for TIAGo to say: Pre-defined speech inputs are provided from a client side file and can be selected to start through the play button.

- **FR6** - The system must stop inputs that are deemed to jeopardise the safety of TIAGo: Base motion inputs are stopped and turned into a stop input if the input movement direction is towards an object and that object is close to TIAGo and is spoken about in more detail in Subsection 7.6.1. Play motion inputs are restricted to one at a time, both client and server side through the user of an action ID, which is generated on the server, and stored client side using a form of browser storage.

- **FR7** - The system must ensure only one client can control TIAGo for a given server of the application at once: This is done using of a session key, which is only distributed to one client at a time, that is required to access all input services on the server and is stored on the client using a form of browser storage.

- **FR8** - The system must help the user to provide consistent base movement inputs: When users provide a safe movement input, given it is similar to the previous input, it is averaged with the last few inputs. This dampens the movement input sent to TIAGo resulting in a more consistent set of inputs. More details are provided in Subsection 7.6.2.

- **FR9** - The system must provide feedback to the user, based on the outcome of their inputs: The system provides feedback through using notifications, colour, and presence to keep the user aware of the outcomes of their inputs. More details are provided in Subsection 7.5.3.

7.7.2 Non Functional Requirements Evaluation

- **NFR1** - The system must be simple to setup: Given the source code and required package managers, all required software packages can be installed and started up using only a few terminal commands, which are displayed in the user guide in appendix Chapter B.

- **NFR2** - The system must be intuitive and easy to learn: The system provides input feedback by using notifications to keep the user aware of their inputs, alongside opting for text over symbols to make it clearer what a given element does. More details are provided in Section 7.5.

- **NFR3** - The system must allow for simple switching between usage of features: The use of tabs and sub-tabs, simple selection process, seamless switching, intuitive grouping of features via tabs and sub-tabs, and pointing out the selected tab by highlighting the selection button makes switching between features a simple process.

- **NFR4** - The system must ensure the safe operation of TIAGo: The safe operation of TIAGo is enforced for the reasons listed in **FR6**, **FR7**, **FR8** and **FR9** alongside the automatic emergency stop when switching tabs or unloading the browser window.

- **NFR5** - The system must comply with UK data regulation: To comply with ISO PECR policies when using cookies, a mandatory cookie policy has been provided, shown in Subsubsection 5.3.1.2. Users are required to agree with this policy before accessing the application, thereby ensuring that users are made aware that cookies are in use, what kind, and how they are being used.

7.8 Goal Evaluation

The goal of this project was to make a simple to use application that allows users to control their TIAGo robot on the go, which anyone could use once setup, irrespective of technical experience. Up to this point. a functional application has been created, with the different parts now having been evaluated. Here, we will briefly lay these out succinctly to see if this goal was achieved.

By achieving all nine functional requirements, some application that allows users to control a TIAGo robot has been developed. Through the completion of requirement **NFR5**, we ensure that what was made is legal to use. The fulfilment of requirements **NFR2** and **NFR3** means that the system must be simple to use, in a way that even non-technical users could use it. Reaching requirement **NRF4** means that the system has mechanisms to ensure that the inputs provided are safe for TIAGo, thereby not requiring the user to know the difference between safe and unsafe inputs. Finally, beyond our goal, the realising of requirement **NFR1** means that even the setup of the system has been kept simple, thereby reducing the barrier to entry for using the system as well as setting it up. Given all of this, it should be safe to say that this project accomplished creating a system that can aid the use of ROS robotics inside, and outside of a lab environment.

Chapter 8

Conclusion and Future Work

This project was about developing a web-based application to act as an interface for users to interact with ROS robots, centred around the TIAGo model of the robot. The goal was to make a system that once setup, even non-technical users could navigate and operate with little to no instruction. Throughout the process, there have been several insights that have come about.

8.1 Key Findings

Even though research into controller design has mostly been focused on physical hardware, many of the properties can be directly applied to a virtual setting with the same effect, to varying degrees of success. This can be especially seen in the use of offsetting inputs, which as seen from the results, makes them more reachable for users.

Via using a hybrid control system, integrating both linear and continuous inputs, an intuitive control mechanism can be made, which is closer to how humans visualise actions than text.

As the theory suggests, the HTTP proves to be effective on a LAN, even though it is mostly used on the world wide web, which is a WAN.

Though not the traditional or intended means of interaction, running a web server in a ROS environment alongside using rospy is an effective way to enable non ROS applications to com-

municate with the different parts of a ROS environment.

Research on shared autonomy for cars can also be applied on a smaller scale to other types of robots. This breeds the opportunity for joint research and knowledge sharing between the research areas. Though the LOA for each project may be different, and the exact system used to define the LOA isn't concretely defined, this shows that even if they may have differing goals, it shows that growth in one of these research areas can support the other.

8.2 Future Work

As I see it, this system could go down either of two directions. The first is continuing to expand the functionality of the client and server as a pair, to offer support for more kinds of robot operations. The second is to take the server and turn it into a standalone software package, expanding on it to support more rospy operations, creating a web-based rospy interface, and adding more data processing operations.

Going down the first route, such work could be done by:

- Adding a model of the robot moving through the environment to the interface, gives the users a better understanding of what the robot sees.
- Including views from the robot's camera and environment, which could enable safe remote control.
- Allowing for the cancellation of text-to-speech inputs after they are started.
- Adding other input methods paired with custom messages for the selected topic for expanded operational control.
- Enabling the recording and local storing of rosbags via the web page. A rosbag is a recording of the inputs sent to a topic.

Going down the second route, such work could be done by:

- Supporting creating custom ROS subscribers, to allow applications to read data from rostopics via the rospy interface.
- Supporting the creation of custom ROS action servers, enabling applications to have more ways to interact with the ros environment.

- Creating more data pre-processing options for various kinds of available operations.

Across both routes, such work could be done by:

- Improving crash prevention for objects with a disproportionately small floor footprint using other sensors, such as the RGB-D camera, to identify surrounding objects better.
- Using the onboard sensors to add some crash prevention features for play motions.
- Enabling the running of locally stored scripts in the ROS environment via the server or client.

Bibliography

[1] C. Bröhl, P. Rasche, J. Jablonski, S. Theis, M. Wille, and A. Mertens, "Desktop pc, tablet pc, or smartphone? an analysis of use preferences in daily activities for different technology generations of a worldwide sample," in *Human Aspects of IT for the Aged Population. Acceptance, Communication and Participation: 4th International Conference, ITAP 2018, Held as Part of HCI International 2018, Las Vegas, NV, USA, July 15–20, 2018, Proceedings, Part I 4*, pp. 3–20, Springer, 2018.

[2] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, p. 5, Kobe, Japan, 2009.

[3] K. Conley, "Topics - ros wiki." [Online] `http://wiki.ros.org/Topics`, 02 2019. (Accessed on 03/01/2024).

[4] K. Conley, "rospy/overview/publishers and subscribers - ros wiki." [Online] `http://wiki.ros.org/rospy/Overview/Publishers%20and%20Subscribers`, 03 2019. (Accessed on 03/01/2024).

[5] S. Glaser, "actionlib_tutorials/tutorials/writing a simple action client (python) - ros wiki." [Online] `http://wiki.ros.org/actionlib_tutorials/Tutorials/Writing%20a%20Simple%20Action%20Client%20%28Python%29`, 03 2023. (Accessed on 03/01/2024).

[6] P. Mathieu and A. R. Tsouroukdissian, "play_motion - ros wiki." [Online] `https://wiki.ros.org/play_motion`, 02 2014. (Accessed on 01/24/2024).

[7] J. Pages, L. Marchionni, and F. Ferro, "Tiago: the modular robot that adapts to different research needs," in *International workshop on robot modularity, IROS*, vol. 290, 2016.

[8] Pal Robotics, "Tiago - mobile manipulator robot." [Online] `https://pal-robotics.com/robots/tiago/`. (Accessed on 11/29/2023).

[9] Pal Robotics, "Tiago base - autonomous mobile robot - pal robotics." [Online] https://pal-robotics.com/robots/tiago-base/. (Accessed on 11/29/2023).

[10] Pal Robotics, "Tiago web user interface - pal robotics." [Online] https://pal-robotics.com/tiago-web-user-interface/. (Accessed on 11/29/2023).

[11] Y. Harel, A. Cyr, J. Boyle, B. Pinsard, J. Bernard, M.-F. Fourcade, H. Aggarwal, A. F. Ponce, B. Thirion, K. Jerbi, *et al.*, "Open design of a reproducible videogame controller for mri and meg," *Plos one*, vol. 18, no. 11, p. e0290158, 2023.

[12] L. V. Santana, A. S. Brandão, M. Sarcinelli-Filho, and R. Carelli, "A trajectory tracking and 3d positioning controller for the ar. drone quadrotor," in *2014 international conference on unmanned aircraft systems (ICUAS)*, pp. 756–767, IEEE, 2014.

[13] M. W. Lee, M. H. Yun, E. S. Jung, and A. Freivalds, "High touch: ergonomics in a conceptual design process—case studies of a remote controller and personal telephones," *International Journal of Industrial Ergonomics*, vol. 19, no. 3, pp. 239–248, 1997.

[14] A. Leff and J. T. Rayfield, "Web-application development using the model/view/controller design pattern," in *Proceedings fifth ieee international enterprise distributed object computing conference*, pp. 118–127, IEEE, 2001.

[15] J. Blomberg, "The semiotics of the game controller," *Game Studies*, vol. 18, no. 2, pp. 311–323, 2018.

[16] M. Van De Wal and B. De Jager, "A review of methods for input/output selection," *Automatica*, vol. 37, no. 4, pp. 487–510, 2001.

[17] C. J. Tomlin, J. Lygeros, and S. S. Sastry, "A game theoretic approach to controller design for hybrid systems," *Proceedings of the IEEE*, vol. 88, no. 7, pp. 949–970, 2000.

[18] P. Huntington, D. Nicholas, and H. R. Jamali, "Website usage metrics: A re-assessment of session data," *Information Processing & Management*, vol. 44, no. 1, pp. 358–372, 2008.

[19] UK Government, "Cookie law comes into force - gov.uk." [Online] https://www.gov.uk/government/news/cookie-law-comes-into-force, 05 2012. (Accessed on 02/26/2024).

[20] Information Commissioner's Office, "Guidance on the use of cookies and similar technologies — ico." [Online] https://ico.org.uk/for-organisations/direct-marketing-and-privacy-and-electronic-communications/guide-to-pecr/

guidance-on-the-use-of-cookies-and-similar-technologies/. (Accessed on 02/26/2024).

[21] B. Preneel, *Analysis and design of cryptographic hash functions.* PhD thesis, Citeseer, 1993.

[22] D. Gourley and B. Totty, *HTTP: the definitive guide.* " O'Reilly Media, Inc.", 2002.

[23] L. Fridman, "Human-centered autonomous vehicle systems: Principles of effective shared autonomy," *arXiv preprint arXiv:1810.01835*, 2018.

[24] M. Vagia, A. A. Transeth, and S. A. Fjerdingen, "A literature review on the levels of automation during the years. what are the different taxonomies that have been proposed?," *Applied ergonomics*, vol. 53, pp. 190–202, 2016.

[25] D. Hopkins and T. Schwanen, "Talking about automated vehicles: What do levels of automation do?," *Technology in Society*, vol. 64, p. 101488, 2021.

[26] SAE International, "Sae levels of driving automation™ refined for clarity and international audience." [Online] https://www.sae.org/blog/sae-j3016-update, 05 2021. (Accessed on 01/26/2024).

[27] OpenJS Foundation, "About node.js — node.js." [Online] https://nodejs.org/en/about. (Accessed on 12/04/2023).

[28] Github, "npm — home." [Online] https://www.npmjs.com/. (Accessed on 12/04/2023).

[29] OpenJS Foundation, "React." [Online] https://react.dev/. (Accessed on 12/04/2023).

[30] K. Conley, "roscore - ros wiki." [Online] http://wiki.ros.org/roscore, 09 2019. (Accessed on 12/04/2023).

[31] K. Conley, D. Thomas, and J. Perron, "rospy - ros wiki." [Online] http://wiki.ros.org/rospy, 11 2017. (Accessed on 12/04/2023).

[32] M. Grinberg, *Flask web development: developing web applications with python.* " O'Reilly Media, Inc.", 2018.

[33] C. Smith and I. McMahon, "rosnodejs - ros wiki." [Online] https://wiki.ros.org/rosnodejs, 05 2017. (Accessed on 01/24/2024).

[34] R. Toris, "roslibjs - ros wiki." [Online] https://wiki.ros.org/roslibjs, 10 2023. (Accessed on 01/24/2024).

[35] J. Mace, "rosbridge_suite - ros wiki." [Online] http://wiki.ros.org/rosbridge_suite, 09 2022. (Accessed on 03/05/2024).

[36] K. Conley, D. Thomas, and J. Perron, "rostopic - ros wiki." [Online] http://wiki.ros.org/rostopic, 10 2022. (Accessed on 03/05/2024).

[37] K. Conley, D. Thomas, and J. Perron, "rosgraph - ros wiki." [Online] http://wiki.ros.org/rosgraph, 02 2015. (Accessed on 03/05/2024).

[38] The Python Cryptographic Authority, "bcrypt · pypi." [Online] https://pypi.org/project/bcrypt/, 12 2023. (Accessed on 01/24/2024).

[39] B. Preneel, "Cryptographic hash functions," *European Transactions on Telecommunications*, vol. 5, no. 4, pp. 431–448, 1994.

[40] J. Martin, "react-joystick-component - npm." [Online] https://www.npmjs.com/package/react-joystick-component, 05 2023. (Accessed on 12/14/2023).

[41] J. Watson, "React select." [Online] https://react-select.com/home, 2022. (Accessed on 12/14/2023).

[42] F. Khadra, "react-toastify - npm." [Online] https://www.npmjs.com/package/react-toastify, 05 2023. (Accessed on 12/14/2023).

[43] S. Moreno, "React testing library — testing library." [Online] https://testing-library.com/docs/react-testing-library/intro/, 08 2022. (Accessed on 02/13/2024).

[44] S. Purcell, F. L. Drake Jr., and R. Hettinger, "unittest — unit testing framework — python 3.12.2 documentation." [Online] https://docs.python.org/3/library/unittest.html, 02 2024. (Accessed on 02/13/2024).

[45] UK Government, "Working with cookies and similar technologies - service manual - gov.uk." [Online] https://www.gov.uk/service-manual/technology/working-with-cookies-and-similar-technologies, 02 2021. (Accessed on 02/26/2024).

[46] F. Xu, H. Uszkoreit, Y. Du, W. Fan, D. Zhao, and J. Zhu, "Explainable ai: A brief survey on history, research areas, approaches and challenges," in *Natural Language Processing and*

Chinese Computing: 8th CCF International Conference, NLPCC 2019, Dunhuang, China, October 9–14, 2019, Proceedings, Part II 8, pp. 563–574, Springer, 2019.

[47] Firefox, "Throttling — firefox source docs documentation." [Online] `https://firefox-source-docs.mozilla.org/devtools-user/network_monitor/throttling/index.html`. (Accessed on 03/16/2024).

[48] OOKLA SpeedTest, "Speedtest global index – internet speed around the world – speedtest global index." [Online] `https://www.speedtest.net/global-index`, 02 2024. (Accessed on 03/16/2024).

[49] A. Al-Rahayfeh and M. Faezipour, "Enhanced frame rate for real-time eye tracking using circular hough transform," in *2013 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, pp. 1–6, IEEE, 2013.

[50] M. Penmetcha, S. S. Kannan, and B.-C. Min, "Smart cloud: Scalable cloud robotic architecture for web-powered multi-robot applications," in *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 2397–2402, IEEE, 2020.

[51] T. Halverson and A. J. Hornof, "A computational model of "active vision" for visual search in human–computer interaction," *Human–Computer Interaction*, vol. 26, no. 4, pp. 285–314, 2011.

[52] K. Seipp and K. Devlin, "Landscape vs portrait mode: Which is faster to use on your smart phone?," in *Proceedings of the 15th international conference on Human-computer interaction with mobile devices and services*, pp. 534–539, 2013.

[53] A. Pereira, T. Miller, Y.-M. Huang, D. Odell, and D. Rempel, "Holding a tablet computer with one hand: effect of tablet design features on biomechanics and subjective usability among users with small hands," *Ergonomics*, vol. 56, no. 9, pp. 1363–1375, 2013.

[54] I. S. MacKenzie, "Fitts' law as a research and design tool in human-computer interaction," *Human-computer interaction*, vol. 7, no. 1, pp. 91–139, 1992.

[55] E. Keyes, "Typography, color, and information structure," *Technical communication*, pp. 638–654, 1993.

Appendix A

Extra Information

List of Figures

Appendix B

User Guide

B.1 Installation

These are the steps to installing all required packages. By this point, you are expected to have a copy of the source code, alongside having installed a version of ROS Melodic or Noetic, Python 3, node package manager, and React 16, 17 or 18.

B.1.1 Client

To install client side packages, from inside the **client** folder, open a terminal and write the following command.

```
npm install --legacy-peer-deps
```

B.1.2 Server

To install client side packages, from inside the **server** folder, open a terminal and write the following command.

```
python3 -m pip install -r requirements.txt
```

B.2 Start up

To use the application, an instance of the server and client need to be started separately.

B.2.1 Client

To start up the client, from inside the **server** folder, open a terminal and write the following command.

```
npm start
```

B.2.2 Server

To start up the server, the terminals must be loaded into a ROS container, and roscore must be active. If roscore is already active on a separate host, skip to the instructions in terminal 2, and set the ROS_IP and ROS_MASTER_URI of the host before the last command in terminal 2. When loading a container, either singularity or apptainer may be used depending on your version on linux.

B.2.2.1 Terminal 1

Open a terminal and write the following command.

```
singularity <container name>
roscore
```

B.2.2.2 Terminal 2

Open a terminal and write the following command.

```
singularity <container name>
# if roscore is on a different host, set ROS_IP and ROS_MASTER_URI
# navigate to server folder
python3 server.py
```

B.3 Testing

This software application comes with separate unit tests for the client and server.

B.3.1 Client

To run the React client unit tests, from inside the **client** folder, open a terminal and write the following command.

```
npm test
```

B.3.2 Server

To run the Python server unit tests, the terminals must be loaded into a ROS container, and a fresh roscore environment must be opened, with no other applications running in it. When loading a container, either singularity or apptainer may be used depending on your version on Linux. From inside the **server** folder, open two terminals and write the following commands.

B.3.2.1 Terminal 1

Open a terminal and write the following command.

```
singularity <container name>
roscore
```

B.3.2.2 Terminal 2

Open a terminal and write the following command.

```
singularity <container name>
python3 tests/__init__.py
```