FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Flexible and Modular Middleware for Industrial Robotic Arm Integration

João Luís Marques de Sousa

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Gil Manuel Gonçalves

August 21, 2022

© João Luís Marques de Sousa, 2022

Resumo

Com os processos de fabrico a tornarem-se progressivamente mais complexos e precisos, as estratégias de programação de robôs devem ser atualizadas. Até à data, existem dois tipos de interacções homem-máquina: programaçao *online*, que se baseia principalmente no *teach pendant*, e a programação *offline*, baseada em ambientes de desenvolvimento de software. Na área de robótica, a integração de diferentes manipuladores robóticos é um dos desafio que as indústrias enfrentam atualmente, dado o elevado nível de heterogeneidade de hardware e software.

Geralmente, cada fabricante de robôs tem o seu próprio software, apenas adequado aos seus produtos. Dado que isto não contribui para melhorar a integração dos manipuladores robóticos nos processos de fabrico, esta dissertação visa estudar e desenvolver uma camada de middleware capaz de os integrar de forma flexível e modular. Além disso, é também integrada com uma Interface Gráfica, de fácil utilização, principalmente dedicada a utilizadores menos experientes. Utilizando a interface, qualquer operador com capacidades de programação reduzidas pode beneficiar da camada de middleware.

Inicialmente é apresentado um estudo teórico sobre middlewares para sistemas robóticos e, de seguida, a solução proposta é detalhada. O middleware é baseado em ligações TCP/IP entre o PC e o controlador do robô. Através destas ligações, é possível receber e interpretar o estado actual do robô e também enviar comandos com a acção pretendida. A camada middleware é composta por três módulos principais (Robot Control, Communication Module, e Data Parser), onde a modularidade e escalabilidade são consideradas propriedades fundamentais. As funcionalidades dos três modulos são também utilizadas pela interface gráfica para programar e controlar o robô.

Os robôs Universal Robots UR5 e Techman TM12 foram utilizados para testar e validar a adaptabilidade da camada de middleware para diferentes robôs. Embora ambos sejam robôs colaborativos, estes diferem em vários aspectos. No entanto, o middleware gere essas diferenças e fornece ao utilizador um conjunto de funcionalidades simples e uniforme.

Foi realizada uma vasta gama de testes para avaliar o desempenho e eficácia da solução proposta, assim como as suas características-chave. Além disso, para avaliar a opinião do utilizador em relação à interface desenvolvida, foram realizados testes de usabilidade a 13 utilizadores diferentes. Os participantes tinham conhecimentos bastante distintos na area da robótica, o que permitiu validar a simplicidade da solução.

Abstract

With manufacturing processes becoming progressively more complex and precise, robot programming approaches should be updated to cope with new industrial robot technologies. To date, there are two types of human-robot interactions: online programming, primarily relying on the teach pendant, and offline programming based on a software development environment. The current challenge in the robotics field that industries face is the integration of different robotic manipulators in an industrial environment due to the high level of hardware and software heterogeneity.

Usually, each robot manufacturer has its own unique software environment, only suitable for its products. As this does not help improve integration in manufacturing processes, this thesis aims to study and develop a middleware layer capable of integrating different robotic manipulators in a flexible and modular way. Furthermore, it is integrated with a user-friendly Graphical User Interface, mainly dedicated to less experienced users. Using the GUI, any operator with reduced programming skills should still take advantage of the middleware layer.

This work begins with a theoretical study about middleware for robotic systems, and then the proposed solution is introduced. The middleware API is based on TCP/IP connections between the PC and the robot controller. Through these connections, we can receive and interpret the robot's current state and also send commands with the user's intended action. It comprises three main modules (Robot Control, Communication Module, and Data Parser) where modularity and scalability are fundamental properties. The GUI also uses the middleware functionalities from the three modules to program and control the robot.

The Universal Robots UR5 and Techman TM12 robots were used as a case study to test and validate the adaptability of the middleware layer for different robots. Even though both are collaborative robots, they differ in several aspects, but the middleware manages those differences and provides the user with a general and straightforward set of functionalities.

A wide range of experiments was conducted to evaluate the performance of the proposed middleware solution and its key features. Moreover, to evaluate user acceptance of the developed GUI, an usability testing method was carried out with 13 different users with two distinct robotics profiles: experts and non-experts, allowing us to validate the simplicity of our solution.

Agradecimentos

A entrega desta dissertação representa o final de uma etapa muito importante da minha vida. Por isso, quero agradecer a todas as pessoas que de certa forma partilharam este percurso comigo.

Ao Professor Gil Gonçalves, orientador na FEUP, e à Carla Pereira, orientadora na Controlar, por toda a disponibilidade e compreensão durante a realização da dissertação.

À Liliana Antão, um especial agradecimento, não só pela orientação durante o semestre, mas também pela grande paciência para aguentar um estudante stressado. Queria também agradecer a constante positividade que em vários momentos me deu uma motivação extra.

Aos membros do laboratório DIGI2, por me ajudarem e participarem nos testes realizados à interface gráfica.

Aos restantes membros do departamento I&DI da Controlar, pelo apoio durante estes últimos meses e por todas as pausas para lanchar onde aprendia cada vez mais sobre o ambiente empresarial.

À Controlar, pela oportunidade proporcionada e pelo excelente ambiente de trabalho.

À minha mãe e ao meu pai, por todas as oportunidades que me ofereceram, pela grande paciência em conviver comigo em alturas de stress e, acima de tudo, por serem o exemplo que quero seguir.

A toda restante família sem exceção, pela constante preocupação e disponibilidade para fazer tudo que tivesse ao seu alcance, se soubessem que me iria fazer sentir melhor. Obrigado pela força e motivação nos momentos menos bons deste percurso e por celebrarem as conquistas ao meu lado.

Aos meus amigos e colegas da FEUP, que me acompanharam nestes 5 anos recheados de gargalhadas, noitadas de estudo e grandes convívios. Agradeço do fundo do coração por tornarem esta jornada inesquecível. Um especial agradecimento ao Ribeiro e Vale pelos momentos hilariantes que passamos dentro e fora dos queijos.

Por fim, aos meus amigos fora da universidade, que tão importantes foram para me proporcionar momentos de pura felicidade, que guardo com muito carinho.

João Sousa

"Life really does begin at forty.
Up until then, you are just doing research."

Carl Gustav Jungl

Contents

1 Introduction **1**
- 1.1 About Controlar - Innovating Industry . . . 1
- 1.2 Context . . . 2
- 1.3 Motivation . . . 2
- 1.4 Problem Definition . . . 3
- 1.5 Objectives . . . 4
- 1.6 Research Questions . . . 4
- 1.7 Document Structure . . . 5

2 Literature Review **7**
- 2.1 Background . . . 7
  - 2.1.1 Industrial Robot's General Architecture . . . 7
  - 2.1.2 Robot Programming . . . 8
  - 2.1.3 Middleware for Robotic Applications . . . 8
  - 2.1.4 Middleware Communication Infrastructure . . . 9
- 2.2 Related Work . . . 10
  - 2.2.1 Robotics Middleware Solutions . . . 11
  - 2.2.2 Robotics Middleware Comparison . . . 16
  - 2.2.3 Other Middleware-based Approaches . . . 17
  - 2.2.4 Summary . . . 22

3 Flexible and Modular Middleware for Robotics **23**
- 3.1 Proposed Solution . . . 23
  - 3.1.1 Initial Assessment . . . 23
  - 3.1.2 Solution Overview . . . 24
- 3.2 Case Study . . . 26
  - 3.2.1 Universal Robots UR5 . . . 26
  - 3.2.2 Techman TM12 . . . 29
- 3.3 Summary . . . 31

4 Middleware Solution Development **33**
- 4.1 Specifications and Requirements . . . 33
- 4.2 Data Parser . . . 34
  - 4.2.1 Universal Robots . . . 36
  - 4.2.2 Techman . . . 38
- 4.3 Communication Module . . . 40
- 4.4 Robot Control . . . 41
- 4.5 Graphical User Interface (GUI) . . . 48

4.6 Summary 54

5 Experiments and Results **55**
5.1 Data Parsing and Communication 55
5.2 Robot Motion 57
5.3 GUI Usability Testing 64
5.4 Summary 68

6 Conclusions and Future Work **69**
6.1 Conclusions 69
6.2 Future Work 71

References **73**

List of Figures

2.1 Industrial Robot generic architecture (extracted from [1]) . . . 8
2.2 Middleware layers . . . 10
2.3 Generic ROS architecture flowchart . . . 11
2.4 Miro abstraction layers (extracted from [2]) . . . 13
2.5 Orca components running on different operating systems (extracted from [3]) . . . 14
2.6 MARIE's Mediator Interoperability Layer (MIL) (extracted from [4]) . . . 15
2.7 OPC UA model of *SkillType* (extracted from [5]) . . . 18
2.8 OPC UA server connections (extracted from [6]) . . . 19
2.9 PMSS architecture (extracted from [1]) . . . 21

3.1 Overall architecture of the system . . . 25
3.2 Universal Robots UR5 . . . 27
3.3 PolyScope interface, version 3.4 . . . 27
3.4 axis-angle with magnitude representation . . . 28
3.5 Techman TM12 . . . 29
3.6 TMflow interface, version 1.84 . . . 30
3.7 Techman's Robot stick . . . 30

4.1 Listen node project example . . . 34
4.2 Tool Center Point offset value . . . 35
4.3 Universal Robots Secondary Client package structure . . . 36
4.4 Portion of the Ethernet Slave Data Table . . . 39
4.5 Techman's TMSVR package structure . . . 39
4.6 Techman's TMSCT package structure . . . 41
4.7 Unit conversion example with UR5 . . . 42
4.8 Speed and acceleration conversion example with TM12 . . . 43
4.9 Manipulator coordinate systems . . . 46
4.10 QtDesigner window example . . . 49
4.11 GUI landing page . . . 50
4.12 GUI home page . . . 51
4.13 GUI Robot status page . . . 52
4.14 GUI Motion control page . . . 52
4.15 GUI Robot Settings page . . . 53

5.1 UR5 status comparison between the GUI and PolyScope . . . 56
5.2 TM12 status comparison between the GUI and TMFlow . . . 56
5.3 Robot's path executing a "Move joints" function . . . 58
5.4 Robot's path executing a "Move Linear" function . . . 59
5.5 Move Joints vs Move linear for TM12 robot . . . 60

5.6 Difference between Move Joints and Move linear when performing a square path 60
5.7 Tool Center Point relative motion example 61
5.8 Path for the precise movement sequence 62
5.9 Speed and acceleration influence in robot's motion 62
5.10 Tool Center Point configuration 63
5.11 Participants background 65
5.12 Results of the answers for the Motion Control Page 65
5.13 Difficulty rating for each task 66
5.14 Comparison of total and expected time for completing the tasks 67
5.15 Overall GUI evaluation 67

List of Tables

2.1 Attribute comparison of existing middleware 16
2.2 Additional attribute comparison of existing middleware 17

3.1 Overview of remote-control specifications 28

4.1 Standard Layout and robot brand variable comparison 36
4.2 Universal Robots Message and Package Types 37
4.3 Robot mode data Sub-Package 37
4.4 Variable type and string character association 38
4.5 Middleware functionalities 44

5.1 GUI data update frequency 57
5.2 GUI Usability Testing tasks performed by the participants 66

Abbreviations

ACE	Adaptive Communication Environment
AI	Artificial Intelligence
AMR	Autonomous Mobile Robot
API	Application Programming Interface
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
FEUP	Faculdade de Engenharia da Universidade do Porto
FTP	File Transfer Protocol
GUI	Graphical User Interface
HMI	Human Machine Interface
ICE	Internet Communication Engine
IDL	Interface Definition Language
IoT	Internet of Things
MARIE	Mobile and Autonomous Robotics Integration Environment
MIL	Mediator Interoperability Layer
ML	Machine Learning
OLP	Offline Programming
OMG	Object Management Group
OPC UA	Open Platform Communications United Architecture
OROCOS	Open Robot Control Software
PC	Personal Computer
PLC	Programmable logic controller
PMSS	Programming and Management Software System
RC	Robot Control
ROS	Robot Operating System
RPC	Remote Procedure Call
RTT	Real Time Toolkit
TCP	Tool Center Point
TCP/IP	Transmission Control Protocol/Internet Protocol
UDP	User Datagram Protocol
URDF	Unified Robotic Description Format
UTF8	UCS Transformation Format 8
XML	Extensible Markup Language

Chapter 1

Introduction

This chapter is divided into seven sections. Firstly, Section 1.1 introduces the company which presented this thesis' theme. Section 1.2 presents the context of the dissertation theme, while Section 1.3 states the arguments that brought about the thesis definition. Moreover, Section 1.4 describes the existing challenges, and Section 1.5 sets the objectives expected to achieve with this dissertation. Section 1.6 lists some research questions to review the performance of the developed solution. Lastly, Section 1.7 delineates this document structure.

1.1 About Controlar - Innovating Industry

Founded in 1995, Controlar has been growing steadily and diversifying its areas of activity, becoming a reference in the areas of Industrial Automation and Test Systems, both nationally and internationally.

Supported by skills such as hardware and software development, systems integration and development, functional and quality tests for electronic devices and robotics, the company has a significant portfolio of projects and services, with references in multinational companies and partnerships with the world's main manufacturers.

The Automation Systems department of Controlar is known to be flexible and capable of responding to different types of projects according to each customer's specific needs. Taking advantage of the interdisciplinary competencies within the group, all stages of project development are handled by Controlar. Its industrial automation solutions include the use of advanced technology, project design, programming, among others. A large number of projects include the integration of several robotic arms, which can be collaborative or not. ABB, KUKA, Universal Robots and Techman are some well-known brands that Controlar supports and integrates into the projects.

The present document describes the development of a flexible and modular middleware for industrial robotic arm integration at Controlar.

1.2 Context

Over the last decade, it is clear that the industry is going through its fourth revolution, also named Industry 4.0. However, this revolution was preceded by three other ones, which led to an improvement of productivity, efficiency, and also life quality. In the late 18th century, the first occurred and established mechanisation and steam engines to replace and improve human labour, known for being monotonous and sometimes unsafe.

The second revolution took place in the second half of the 19th century with the development of assembly lines, which eased mass production. Then, during the 1970s, automation was the dominant concept of the third industrial revolution, along with IT systems, robots, and computers. The so-called digital revolution made it possible to automate whole manufacturing processes using computers and Programmable Logic Controllers (PLCs) and further increase productivity.

Finally, the most recent and the one we are living on is Industry 4.0, which is recognised by concepts like Internet of Things (IoT), Artificial Intelligence (AI), and Machine Learning (ML).The advanced computing power and the wide access to data allow for incorporating fields of science like AI and ML into industries and setting a new concept entitled Smart Factory. Therefore, a Smart Factory is known for the exceptional ease of accessing and sharing data between machines or devices and using it to improve and optimise manufacturing processes.

However, industries are under increasing pressure from the market and, with demand rising and enhanced needs for product customisation, it is required to speed up production to remain competitive. Thus, they need to rely on industrial robots to assemble production lines capable of fulfilling the ever-changing market requirements, resulting in productivity improvements, automated processes, and increased profitability. Industrial robots are robust and precise machines capable of executing tasks that are too difficult or dangerous to be done by humans or even that require high precision.

After all, this new industrial revolution requires flexible manufacturing processes, to cope with the market demand, and simultaneously develop simple solutions, to avoid the need for an expert field operator capable of programming the industrial robots. Given the wide range of robots and the high engineer turnover rate, finding the right person for the desired job can be cumbersome.

1.3 Motivation

In conformity with the new industrial revolution, factories rely on advanced industrial robots to deploy assembly lines in order to fulfill market requirements. Doing so enables the companies to effectively meet high productivity demands and remain competitive. In fact, robots have provided advanced automation that plays a crucial role in reshaping manufacturing processes. Programming machines achieve a significant level of autonomy to execute a sequence of movements. However, as previously mentioned, markets are changing and becoming complex, demanding flexible approaches from industries.

One of the most challenging aspects of the robotics field is integrating and deploying different robotic manipulators in an industrial environment. Given the wide variety of robotic handlers with different functionalities in automation processes (such as assembling, welding, or screwing), the need for more straightforward methods to integrate these systems becomes critical. In reality, the employment of robotic handlers is growing globally, although they must contribute to flexible manufacturing processes while attending to the ever-changing market needs [7]. Due to the increasing complexity of modern industries, flexibility and scalability are two main concepts to consider in such systems [1]. The flexibility and scalability of a manufacturing system define how easy and convenient it is to adjust it accordingly to the market requirements and thus quickly answer to those changes.

Since smart factories transformed the organization of manufacturing processes, it also implies hiring qualified and specialized engineers capable of programming and hence benefiting from the wide variety of industrial robots [8]. For this reason, industries also benefit from the development of a flexible and modular framework because the impact of the high turnover rate is significantly lower. They no longer require going through the arduous process of hiring specialized engineers, as the framework provides a cross-platform and generic programming approach suitable to several robots.

1.4 Problem Definition

Since different types of robots have different frameworks and programming methodologies, there are many specialised software platforms developed by robot suppliers, such as Techmans's TM-Flow, Universal Robots' PolyScope, and ABB's RobotStudio. These commercial software platforms are specifically designed to program, communicate, and integrate robots from the correspondent manufacturer, following its established standards. Even though developing such applications requires qualified engineers to deal with this complex and challenging task, they are nowadays considered imperative. Graphic simulation of the robot movements, offline programming, motion planning, and robot configuration are some key functionalities. Nevertheless, apart from the fact that each platform is only suitable to robots from the corresponding company, they do not contribute to a flexible and generic manufacturing methodology and sometimes can be quite expensive to manage [1]. The latter primarily affects small and medium enterprises because it requires a high initial investment in software and qualified human resources [9]. Regarding the communication methods, devices from different companies may adopt several communication protocols, making the universality aspect even more complicated. Moreover, the complexity and heterogeneity of robotic software and hardware also hinder such applications' trivial development and reusability.

In short, it is notorious the lack of universality, flexibility, and simplicity regarding robot control applications. The closed source and specialised software platforms constitute a significant entry barrier for both robotic industries and researchers in the robotic field.

1.5 Objectives

The dissertation aims to study and develop a middleware layer capable of integrating different robotic manipulators in a flexible and modular way, integrated with a simple graphical interface. In other words, it is an abstraction layer between a Graphical User Interface (GUI) and the Robot Control layer (RC). It manages diversified hardware equipment and allows the user to cope with a general and straightforward set of functionalities suitable to a wide variety of robotic handlers.

Every robotic handler has a Robot Control layer responsible for the robot motion control and its design varies depending on the robot's manufacturer. Considering the existing specialised and complex software platforms, integrating distinct robotic handlers on the same production line can be arduous. The middleware layer addresses this issue by linking the user interface with the Robot Control layer. It also means that the user does not have to be concerned about what handler is being controlled nor its framework and constraints [1].

The middleware layer brings two essential concepts: transparency and modularity. The developed middleware must be transparent, meaning that it will feature a collection of functions that are common to all robot handlers, such as moving, stopping, and adding critical points to the trajectory. Furthermore, it must be flexible and modular by allowing the addition of both handler integration modules and new features for the manipulators. Lastly, it is also worth mentioning that the middleware layer must fulfil real-time constraints by guaranteeing processing times lower than the planned execution time.

To complement the middleware layer and provide a better user experience, a GUI must also be developed and aid robot control.- It is expected to be user-friendly and mainly dedicated to less experienced users, so any operator with reduced programming skills can still take advantage of the middleware layer.

1.6 Research Questions

Given the aforementioned objectives, we formalized several research questions we hope to successfully answer with this thesis work, namely:

- Is it possible, using the same middleware layer, to interact and program industrial robots of distinct types or specifications?
- Does the modular middleware layer allow simple and quick integration of handler integration modules?
- Is the middleware layer scalable so that new features for the manipulators can be incorporated?
- Does the middleware layer fulfil the existing real-time constraints?
- Considering inexperienced operators in coding, does the Graphical User Interface simplify the integration of different robots?

1.7 Document Structure

This document is divided into six chapters. Following Chapter 1, which is the Introduction, the document is organized as follows.

Chapter 2 presents the fundamental concepts necessary to understand the robotic middleware area as well as the related work found on middleware for robotic systems, which addresses the integration problem of heterogeneous robotic systems.

Furthermore, Chapter 3 presents the proposed solution details and describes the two robots chosen as a case study to test the middleware layer.

Chapter 4 starts by presenting the middleware's specifications and requirements, followed by the implementation description of the middleware layer. It ends with the presentation of the GUI and the details of its development.

Chapter 5 shows an account of the experiments performed to validate the different aspects of both the middleware layer and the GUI. It starts by presenting the results of reading the robot's status and is followed by the analysis of the robot's path for different motion types. The GUI is evaluated in terms of its usability through a usability testing method performed by several participants.

Lastly, Chapter 6 gathers the conclusions of this project and identifies useful aspects for future work.

Chapter 2

Literature Review

This chapter presents some fundamental concepts and the literature review on robotic middleware. Firstly, Section 2.1 presents crucial knowledge required to understand robotic middleware development. It starts by describing a general industrial robot architecture, followed by the main approaches to robot programming. Then, it addresses the middleware's goals and key features and outlines some standard network middlewares. Finally, Section 2.2 presents the literature review of some existing middlewares as well as the recent approaches to address the lack of universality and interoperability in robotic systems.

2.1 Background

2.1.1 Industrial Robot's General Architecture

Three main layers can generally describe the architecture of an industrial robot: the user layer, the controller layer, and the hardware control layer [1]. The user layer is at the top of the stack, and it provides the operator with the available ways to configure and program the robot to execute the desired tasks. The teaching pendant allows online programming through a simple interface, and it is often used to execute basic tasks or monitor the robot while operating. Additionally, the PC module, mostly used by skilled engineers, manages the robot settings and the simulation environment whilst supporting offline programming. Ethernet/IP, a widely used communication protocol in automation, usually establishes the communication between the user and controller layers. The layer responsible for the motion control is the controller layer, also considered the key layer. At this stack level, two main modules interact between them – the Robot Control (RC) and the soft PLC – while the first one is responsible for planning the robot movement, the second one manages the programmed tasks. Finally, the hardware layer controls the robot's motors besides handling input or output devices. The two last-mentioned layers interact via industrial Fieldbus, a set of communication protocols mainly used in industries. Fig.2.1 represents the inter-layer interactions described above.

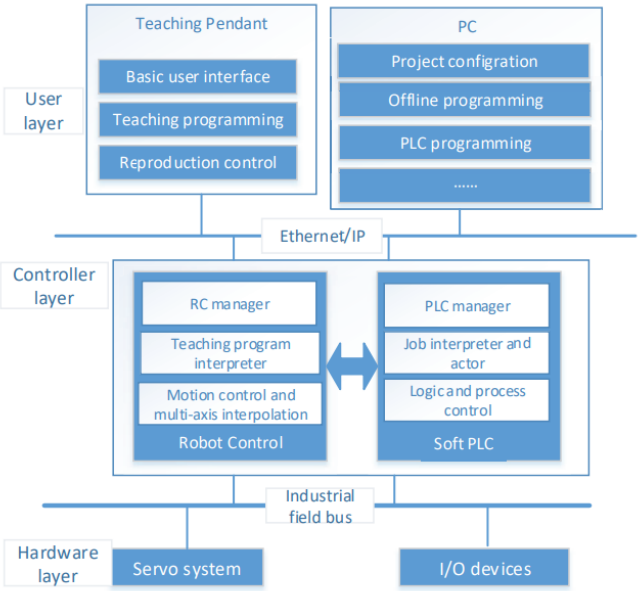


Figure 2.1: Industrial Robot generic architecture (extracted from [1])

2.1.2 Robot Programming

There are two significant ways to program industrial robots: online teaching using a teach pendant and offline programming (OLP) based on a software development environment. However, the target users are different for each type. Online programming is designed for field operators, whereas OLP is devoted to software development programmers. Additionally, online teaching requires stopping the robot and its manufacturing process just to program it. In contrast, OLP allows developing and simulating programs before testing with the real robot without being mandatory to stop it. Programming tasks using the teach pendant is frequently time-consuming and less efficient than OLP. However, as of today smaller companies have been adopting this method since some teach pendants have become more user friendly, discarding the need for a specialized operator.

Alternatively, OLP consists essentially of special integrated development environments [1]. It is helpful for more complex tasks and improves both robot integration and operator safety since programs can be simulated beforehand. However, this type of software is highly-priced, dedicated to the correspondent robot manufacturer, and requires qualified engineers to configure and program the robot. For that reason, research institutions pursue universal robot middleware to achieve simple, flexible, and modular manufacturing processes.

2.1.3 Middleware for Robotic Applications

Currently, according to industrial demands and constraints, robotic systems require a high level of flexibility and interoperability to cope with the complexity and heterogeneity of both hardware

and software. Several hardware devices can equip today's robots, allowing them to speed up manufacturing processes with cutting-edge technology. Therefore, distance, navigation, and pressure sensors are a few examples of components necessary for the proper operation of the robot, as well as ensuring the required autonomy and safety. However, even just regarding the industrial environment, there are multiple robot types to consider. An Autonomous Mobile Robot (AMR) relies on range sensors and cameras to simultaneously compute and cruise its environment, planning its trajectory near real-time. A collaborative robot is designed to operate alongside humans, yet with current advances in artificial intelligence, it can even learn and adapt its behaviour to the robot's operator. Moreover, a robotic arm is composed of a set of joints that offer various degrees of freedom resembling a human arm, and it can perform different types of tasks according to its environment.

A robot can then be described as a distributed system made of several hardware and software modules, although their features vary according to the robot manufacturer. As expected, each manufacturer builds the robot with a set of hardware components and develops a software application that allows its programming. The level of heterogeneity of these systems is furthermore increased when you also consider the variety of communication protocols available. Thus, due to the lack of robotic standards, developed frameworks that allow the integration of multiple robots are overly limited to the environment for which they were designed. In addition, considering the mentioned robotic heterogeneity, building a general and flexible framework that takes advantage of all features of multiple robots while keeping straightforward their integration is arduous [10].

Nevertheless, by having an alternative solution to the specialised software platforms and a default communication standard, it is easier to run complex manufacturing processes since it allows high integration and scalability [11]. Therefore, as illustrated in Fig.2.2, robotic middleware is an abstraction layer that lies just over the operating system layer while staying below the application layer. Its main purpose is to handle hardware diversity and thus simplify programming the robot. The addition of components should be effortless by concealing the complexity of the underlying ones. Furthermore, it improves efficiency in the robot programming task as the developer can follow a modular methodology, programming features as modules that can later be added or even to update an existing module [2]. Since robots usually have strict real-time constraints, the middleware must meet them. It should also be characterised by both user-friendliness and reliability, along with the ability to be scalable as the system grows.

2.1.4 Middleware Communication Infrastructure

The middleware communication infrastructure consists of any standards or technologies used to communicate with several devices. Every middleware uses a default or a custom communication model since it is essential to its functioning.

There are several standard network middlewares such as the Common Object Request Broker Architecture (CORBA) [12] and Internet Communication Engine (Ice) [13]. CORBA is, without a doubt, the most used in robotic middleware. Miro and OROCOS are two examples of robotic

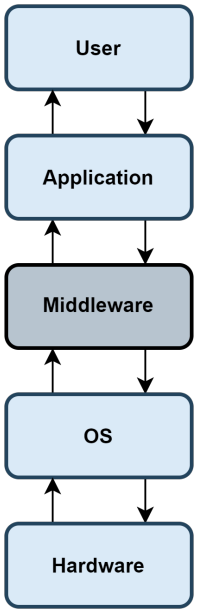


Figure 2.2: Middleware layers

middleware that rely on CORBA to establish its communication framework. CORBA is an object-oriented standard defined by the Object Management Group (OMG) that aims to help develop distributed applications. Moreover, Ice is also an open-source, object-oriented communication middleware presented by ZeroC. It is a Remote Procedure Call (RPC) framework that emerged based on the CORBA design, aiming to be less complex. Orca middleware uses Ice when establishing the interface between cross-platform components.

In addition, some middlewares do not use a standard communication infrastructure and instead rely on custom communication models based on shared memory.

2.2 Related Work

This section presents the related work found on middleware for robotic systems. It starts with the review of some existing middlewares developed throughout the years. Since there are multiple middlewares and new ones are constantly being released, Section 2.2.1 describes the most relevant ones. After that, the mentioned middlewares are compared and evaluated in Section 2.2.2. Furthermore, some recently proposed solutions to address the integration of heterogeneous robotic systems are also presented. Finally, Section 2.2.4 outlines the presented solutions and identifies some of their weaknesses.

2.2.1 Robotics Middleware Solutions

Robot operating system (ROS)

Robot Operating System (ROS) is an open-source system developed at Stanford University within the scope of the STAIR project [14], and it is currently the most popular robotic middleware due to its features, which benefit the current manufacturing processes. It follows a Peer-to-Peer architecture, allowing the integration of multiple devices into the same system. Moreover, it is Multi-lingual supporting code written in C++, Python, Octave, and LISP, and it is based on a modular structure ensuring the system's scalability. ROS also stands out for its ability of hardware abstraction, package management, and communication between processes. As a result of its popularity, this community-driven middleware contains numerous software packages with drivers and robotic algorithms. ROS's modular and flexible framework makes it compatible with other robotic middleware like YARP, OpenRTM, and OROCOS, as well as with simulation environments such as Stage and Gazebo [15].

Following the modular architecture, ROS provides four essential concepts: nodes, messages, topics, and services. Fig.2.3 illustrates the ROS architecture.

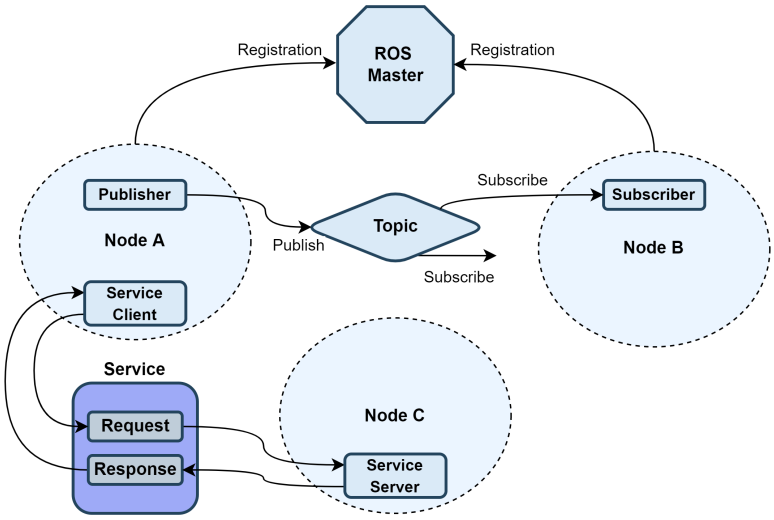


Figure 2.3: Generic ROS architecture flowchart

The publish-subscribe model is implemented through nodes, messages, and topics, while synchronous events may use the service model (client-server model) [14]. Nodes can be described as processes that can asynchronously communicate with each other, and since ROS is a modular system, it is possible to run multiple nodes simultaneously. The node-based methodology gives ROS an increased fault tolerance since the nodes crash individually. Inter-process communication happens by passing messages (over a TCP/IP transport layer), containing data arranged in a predefined structure, through topics defined by a name. Consequently, a node can obtain data by just subscribing to a suitable topic that can have multiple publishers and subscribers. However, it is

necessary to mention that the receiver must know the content structure of the message beforehand. Otherwise, it is not capable of understanding it.

Due to the increasing usability of ROS and its main features, new variants are being developed, such as ROS 2 and ROS Industrial (ROS-I). ROS 2 is still under development and, besides other main objectives, seeks to improve the multi-robot systems while ROS-I is meant to adapt ROS features to industrial purposes. Hence, robot manufacturers are aware of the benefits of blending ROS with existing technologies and already using this system in their robots.

As expected, there are some drawbacks to using ROS middleware. ROS has no Real-time capability since its primary operating system Linux is not inherently real-time. The communication framework on top of TCP/IP also does not guarantee to execute according to the real-time constraints. Therefore, ROS follows a best-effort approach. According to [16], ROS also lacks security protections, but several developed architectures already address this problem.

Open Robot Control Software (OROCOS)

Open Robot Control Software (OROCOS) was born in 2000, introduced by Herman Bruyninckx as an Open-Source alternative to specialised robot control software in the robotic research area [17]. This component-based middleware was created to address the insufficient universality and interoperability since robot manufacturers produced them with specific hardware, software, and communication protocols. OROCOS has progressively proved helpful for research and commercial purposes due to its great modularity and flexibility.

OROCOS is divided into three modules – Supporting modules, Robotic modules, and Components. The first mentioned incorporates complementary but meaningful functionalities such as 3D simulation, real-time support, and configuration tools. According to the Open-Source fundamental principles, the mentioned functionalities can be obtained by reusing code from other Open-Source projects. The second contains specific robotic algorithms and libraries assembled in modules that focus on motion planning, kinematics, control of robot hardware, and so on. Lastly, the components are CORBA user-developed objects. A set of components is what establishes robotic control applications. Considering the example of CORBA as network middleware and XML for configurations, OROCOS follows an open standards methodology. However, what stands out most is its capacity for hard real-time operations using the Real-Time Toolkit (RTT). The library provides the infrastructure to develop components that satisfy such real-time constraints, and the applications are usually programmed in C++.

Although OROCOS is cross-platform and localized for multiple programming languages (C++, Python, Simulink), its biggest drawback is the fact that there is no graphical IDE nor a simulation environment [18].

Miro

After developing specific robotic software platforms for five different robots, a research group proposed to design a generalized, reliable, and robust framework to integrate a variety of robots.

Considering the robotic heterogeneity challenges previously faced, the group developed Miro, an open-source, object-oriented robotic middleware. It provides hardware abstraction and improves the hardware devices' integration and interoperability [19].

Miro relies on the CORBA standard using the Adaptive Communication Environment (ACE) library as its communication framework. This standard allows to build software applications more efficiently as well as to achieve inter-process and cross-platform interoperability. As it is possible to see in Fig.2.4, this middleware consists of three layers: the device layer, the service layer, and the class framework. The Miro device layer, which is the platform-dependent section, provides an abstracted interface for the robotic sensors and actuators details. The Miro service layer provides service abstraction using CORBA's Interface Definition Language (IDL). Thus, sensors and actuators are represented as network-transparent CORBA objects with control functions associated with them. The Miro class framework offers a set of commonly used services for mobile robots, such as path planning, localization, and mapping. It is a layer that emphasises the importance of reusing code even for different applications or platforms.

According to Utz *et al.* [19], Miro was successfully implemented in three mobile robots with different applications and proved stable and robust. However, there are two main drawbacks when considering Miro. Due to the CORBA standard, additional components or software applications can only be written in languages or platforms that provide CORBA implementations. Furthermore, it does not provide any kind of monitoring or management interface.

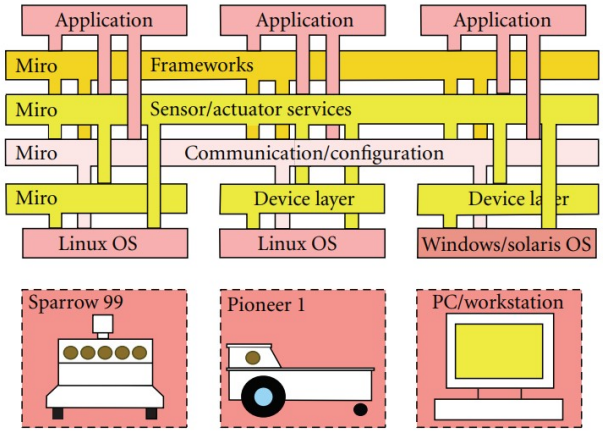


Figure 2.4: Miro abstraction layers (extracted from [2])

Orca

Makarenko *et al.* [3] stated that developing robotic software is a challenging and time-consuming task. It is also recognised to be one of the barriers to robotic advancement. Therefore, they proposed Orca, a component-based open-source software framework for a broad range of robots and distributed sensor networks. Orca highlights software reuse as the authors consider it a critical factor in making robotic areas progress, targeting both academic and commercial environments.

Thus, Orca is a general, flexible, and modular implementation framework that allows developers to build robotic software components. Component-based software enables developers to import previously built plug-in components rather than starting from scratch. Moreover, it benefits the modularity of the middleware and helps control dependencies while not imposing any design architecture. Fig.2.5 shows two components running on different operating systems and written in different programming languages. Orca also has available a set of APIs in the libOrcaIce library. This library contains several accessories and patterns commonly used when developing such robotic applications. Nevertheless, the developer can choose whether to use it or not, making it possible to build new components without using any available tools. Regarding the communication mechanism, also named network middleware, when establishing the interface between cross-platform components, the authors considered some possible options – XML-based technologies (e.g., SOAP), CORBA, and the Internet Communications Engine (Ice). However, Ice was chosen because XML-based technologies were too inefficient for low-level robotic control tasks, and CORBA was overly complex and extensive. Ice's methodology is similar to CORBA and deals with communication tasks, supporting protocols like TCP and UDP. It offers a lighter and more consistent API than CORBA while maintaining a good performance.

Similarly to Miro, Orca does not provide any kind of monitoring or management interface. This and the overhead caused by the use of Ice constitute the main disadvantages of this implementation framework.

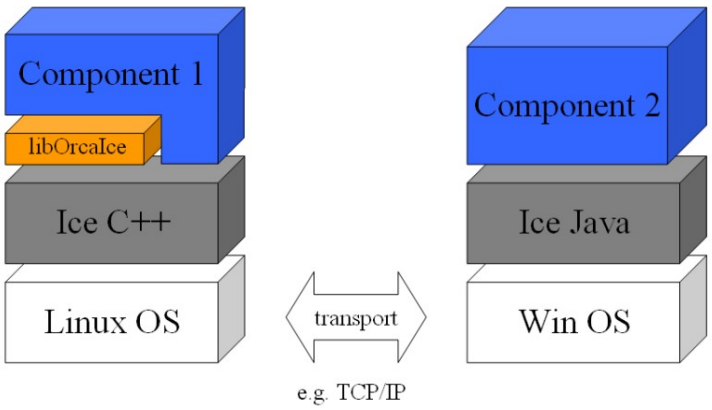


Figure 2.5: Orca components running on different operating systems (extracted from [3])

Mobile and Autonomous Robotics Integration Environment (MARIE)

Since several programming environments followed different methodologies for robotic systems improvement and integration, the majority were not compatible with each other. Different architectures and operating systems, as well as distinct communication protocols selected, were some of the incompatibility causes. Therefore, Côté et al. [20] created MARIE (Mobile and Autonomous

Robotics Integration Environment), a new open-source middleware framework. It is remarkable because it integrates new and existing components, tools, and software for robot systems.

Despite the components' heterogeneity, the distributed component-based middleware framework aims to simplify robotic software development by enhancing its reusability. As a rapid-prototyping approach, MARIE handles the integration of inherently incompatible frameworks, resulting in increased flexibility and ease of development of such software due to access to a broader set of functionalities. MARIE adopted a layered software architecture in order to achieve multiple sets of concepts and abstractions. It consists of three abstraction layers: the Core layer, the Component layer, and the Application layer. At the bottom of the stack, the core layer consists of low-level tools for communication, data handling, and I/O control. The Component layer offers the means to add and work with reusable components. At the top of the stack, the application layer is responsible for providing the tools to integrate and build robotic applications.

Moreover, MARIE relied on the Mediator Design Pattern to forge a Mediator Interoperability Layer (MIL), as presented in Fig.2.6. MIL follows the methodology of a centralized control unit denominated Mediator, providing a solution to the communication between components with heterogeneous features [4]. In other words, each application may have specific communication

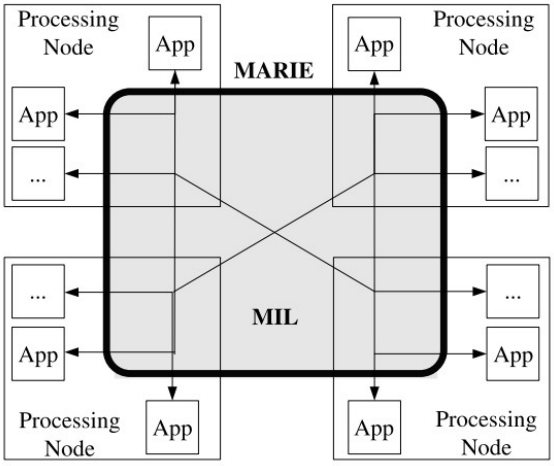


Figure 2.6: MARIE's Mediator Interoperability Layer (MIL) (extracted from [4])

protocols, but if the MIL supports it, they can interact with each other. MIL includes four functional components: Application Adapter (AA), Communication Adapters (CAs), Communication Manager (CM), and Application Manager (AM). AA operates as a proxy between the MIL and an application. The CAs are responsible for handling and translating data exchanged between AAs of applications that have communication incompatibilities. The communication links between AAs are created and managed by the CM. Lastly, AM controls and manages distributed components among the system. Assembled in the C++ programming language, MARIE uses the Adaptive Communication Environment (ACE) as its communication framework.

However, MARIE does not provide any compilation tools or deployment system, making the environment testing process more complicated and time-consuming.

2.2.2 Robotics Middleware Comparison

After describing and presenting the architecture of some significant middleware developed over the past years, this section aims to summarise their main features and briefly compare them. Both the analysis and comparison of the frameworks are based on [2], [18], [21], [15], [22], [23]. Therefore, Table 2.1 presents the described middlewares alongside their key attributes.

Table 2.1: Attribute comparison of existing middleware

Middleware	**Architecture**	**Simulator**	**Platforms**	**Communication infrastructure**	**Real-Time**
ROS	Component-based Publisher/subscriber	Yes	Linux	Message Oriented RPC services	No
OROCOS	Component-based Event-based control	No	Linux Windows	CORBA	Yes
Miro	Client-Server Event-based control	Yes	Linux Windows	ACE CORBA	No
Orca	Component-based Peer-to-peer	Yes	Linux Windows	Ice	No
MARIE	Component-based Centralized control	Yes	Linux	ACE MIL	No

According to Table 2.1, most of the frameworks are component-based but lack a standard control model. Having a simulation environment such as Stage or Gazebo is always beneficial for application development, but only OROCOS fails to provide such feature. Moreover, being cross-platform is a major middleware attribute since it improves flexibility and scalability. For example, ROS is still not wholly supported in Windows. The communication infrastructure attribute points out the standards and technologies used in each middleware. While ROS follows a custom communication protocol, other frameworks rely on standard network middleware such as CORBA, ACE, and Ice. From the mentioned middlewares, only OROCOS is suited for real-time control tasks. ROS does not provide Real-time capabilities because both the Operating System and the communication mechanism (TCP/IP) are not Real-time.

Additional attributes like management and development tools, overhead degrees, and programming languages should also be studied. That being said, Table 2.2 lists and compares those attributes.

Management tools monitor and interpret the system state during execution, thus simplifying error detection. Development tools like compilers, IDEs, and configuration files are valuable factors influencing middleware selection. In this case, among the reviewed middlewares, only ROS offers this type of tools. MARIE has a GUI that helps the user monitor the system's status. Besides that, overhead measures the consumption of hardware resources (mainly memory and CPU) by the middleware, apart from the resources needed for the operating system. ROS middleware has the most negligible overhead among the mentioned ones. Due to Ice and CORBA standards, OROCOS, Miro, and Orca have higher overhead. The same happens to MARIE because of the

Table 2.2: Additional attribute comparison of existing middleware

Middleware	Management and development tools	Overhead	Programming Languages
ROS	Yes	Small	Python C++
OROCOS	No	Large	Python C++
Miro	No	Large	CORBA
Orca	No	Large	C/C++ Pyhton Java
MARIE	Monitoring tools	Large	C++

functional components. Lastly, Miro has the stricter programming restrictions as it can only be written in languages that provide CORBA implementations. ROS is mainly programmed in C++ or Python but also supports Java, Lisp, and other languages.

2.2.3 Other Middleware-based Approaches

Using OPC UA as middleware for heterogeneous manufacturing systems

Considering all the existing middleware mentioned in the previous section and similar ones, Profanter *et al.* [5] presented a different approach to achieve a flexible and easy integration of heterogeneous robotic systems. Therefore, pursuing the same goal of simplifying the reuse of software modules for multiple devices, defined a generic skill-based model that relies on the Open Platform Communications Unified Architecture (OPC UA) communication protocol.

To promote the integration and flexibility of new devices on the production line, they must support a common communication interface. OPC UA is a platform-independent object-oriented IEC standard (IEC 62541) designed to enable data exchange and communication among multiple devices. OPC UA provides a hardware-independent communication protocol and semantic modelling capabilities. It models devices and describes their properties and functionalities. Moreover, skills-based architectures are a distinctive approach to robot programming and benefits flexible component exchange. Skills, acting as tools to process modelling, are effectively functionalities provided by a given component and also provide hardware and software abstraction. According to the authors, it is mandatory to use an adaptable underlying skill model which fulfils specific requirements. A skill must be classified by a type definition and identified by a set of base properties. Each skill can be in diverse states, and methods define the state transitions.

Most of the mentioned requirements are fulfilled using the OPC UA Specification Part 10 Programs. An OPC UA Program has a set of states (Halted, ready, running, and suspended) as well as basic methods to trigger transitions. That being the case, Profanter *et al.* introduced *SkillType*, a sub-type of the OPC UA Program. It is the base type of all system skills and, together with the *ISkillControllerType* interface, allows the user to easily search for a particular skill in the

OPC UA server. Since *SkillType* can work with different devices (robots, grippers, tools), software components can then be modular and composed of multiple skills. For example, the pick and place function can be performed using the robot movement skill combined with the gripper skill. Fig.2.7 illustrates the OPC UA model of *SkillType*.

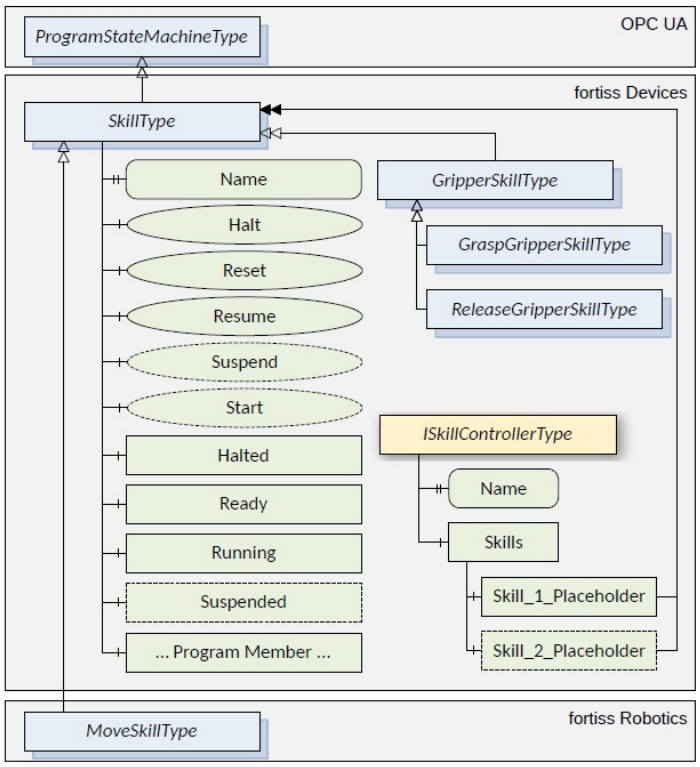


Figure 2.7: OPC UA model of *SkillType* (extracted from [5])

The proposed hardware-agnostic skill model was implemented in C++. OPC UA servers implement the skills to be tested, and a generic client controls the robots through the movement skills. A single generic client is enough to control the robots because every OPC UA server offers the same interface and movement skills. Three different robots were tested to evaluate the architecture: KUKA iiwa, Universal Robots UR5, and Comau e.DO. Each robot has a specific hardware setup and joint lengths. After the successful tests of the movement and pick-and-place skills, it is possible to state that OPC UA allows to semantically model information whilst the skill model has proven to be robust and suitable for robot control.

The usage of a single OPC UA client also proves the hardware-agnostic robot control and the easy exchange of components. However, the authors also state that the lack of support of specific OPC UA companion specifications is a significant issue and expect manufacturers to provide OPC UA servers compatible with the skill model. The driver interface implementation was not fully explored in this paper, but due to the differences in each robot driver, a generalised implementation of the model was not ideally achieved.

Arnarson *et al.* [6] also examined the use of the OPC UA standard for a hardware-agnostic middleware. The Industry 4.0 concept expects a consistent and secure connection between software and hardware components to shape the so-called smart factories. Thus, the communication challenge between heterogeneous components can be solved by introducing a communication middleware. The middleware must handle the specific communication protocols from different robot manufacturers.

Given that the OPC UA protocol is becoming a de facto standard in manufacturing systems, they decided to rely on OCP UA servers to communicate with the robots [5]. Robots' exchange is possible and nearly trivial using a single OCP UA server, which means that different robots can be controlled using the same interface. Moreover, it allows remote robot control since an OPC UA client can connect to the server and obtain robot data, as well as call functions to execute specific movements.

However, establishing a connection between industrial robots and an OPC UA server is arduous because most of them are not sufficiently prepared to communicate with other equipment. Thus, the authors explored how to control multiple robots from an external device. Fig.2.8 illustrates the proposed architecture.

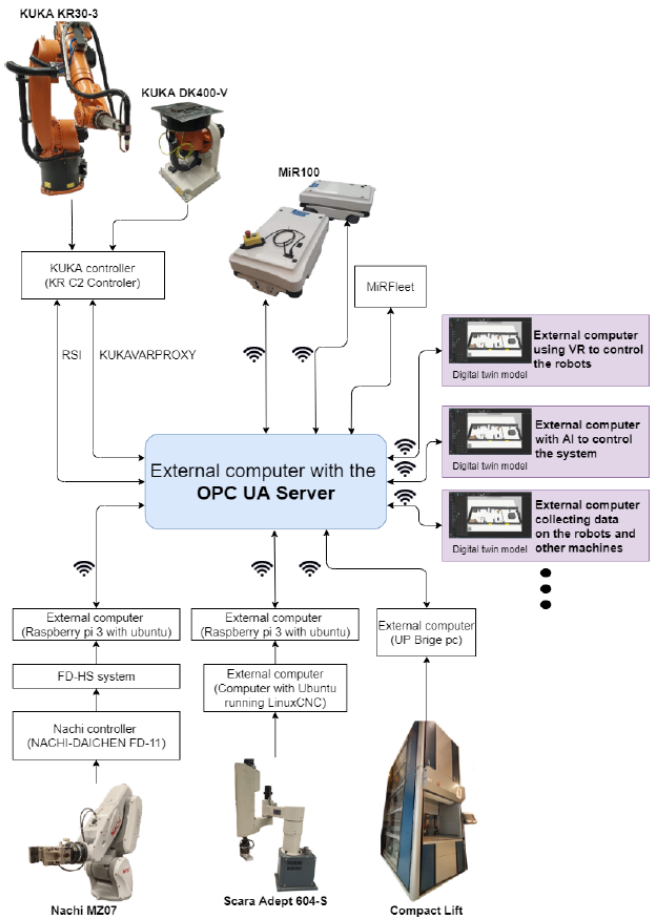


Figure 2.8: OPC UA server connections (extracted from [6])

The key component is the OCP UA server which gathers and shares data (formatted according to the OPC UA standard) from every connected machine. An external computer hosts the server and connects it to a Wi-Fi router. In this manner, machines can connect to the server via ethernet cable or establish a wireless connection.

The complexity of the architecture comes from the interface between robots and the OPC UA server. The KUKA robot and rotary table use the same controller and link to the server with an ethernet cable. Two python applications act as clients to provide real-time control and gather additional data. The two automated intelligent vehicles (MiR100) connect to the server via Wi-Fi. MiRFleet is also connected via an ethernet cable and can control the AIV's. Like the KUKA robot, two python programs are running as clients to control the fleet and acquire their position. The compact lift requires an external UP bridge PC connected to the server to be integrated into the system. The Nachi robot does not support servo control with the OPC UA server, so another external computer is used, running ROS to control the robot and calculate the inverse kinematics. Lastly, the Scara robot was controlled with an external PC and a raspberry pi used to connect to the server. The original control board of the robot was obsolete, so the software LinuxCNC replaced it.

Having all the machines connected to the single OPC UA server, a client can run a python script to control the machines' movement. The performed evaluation tests have shown that KUKA and Nachi robots have faster response time when compared to the other machines. Thus, it is possible to conclude that the additional devices used for the OPC UA server connection affected the system performance.

Additionally, although the OPC UA middleware is considered a helpful approach model for integrating heterogeneous robots, its implementation is still complex and not as generic as desirable. Nevertheless, in the Industry 4.0 context, the OPC UA standard provides a flexible, reliable, and robust middleware solution.

Programming and Management Software System (PMSS)

Cao *et al.* [1] introduced the Programming and Management Software System (PMSS) in order to combine the versatility of middlewares such as ROS and OROCOS with the performance of the special development environments provided by the robots' manufacturers. Although the quality and performance of the specialised software platforms are unquestionable, they are too expensive and do not contribute to a flexible and straightforward integration of heterogeneous robots. The architecture of the proposed solution is divided into five parts: Robot configuration and management, Software system management, off-line programming, PLC module, and Communication interface. Fig.2.9 illustrates the PMSS architecture.

The "Robot configuration and management" models describes the robot's hardware into a standard data structure. Since ROS uses the Unified Robotic Description Format (URDF) for representing the robotic system components, PMSS adopted the same concept. URDF is an XML file format that ROS uses to model the systems and then test them in a simulation environment. In addition, in PMSS, the users can either write this file from scratch or select a provided default model.

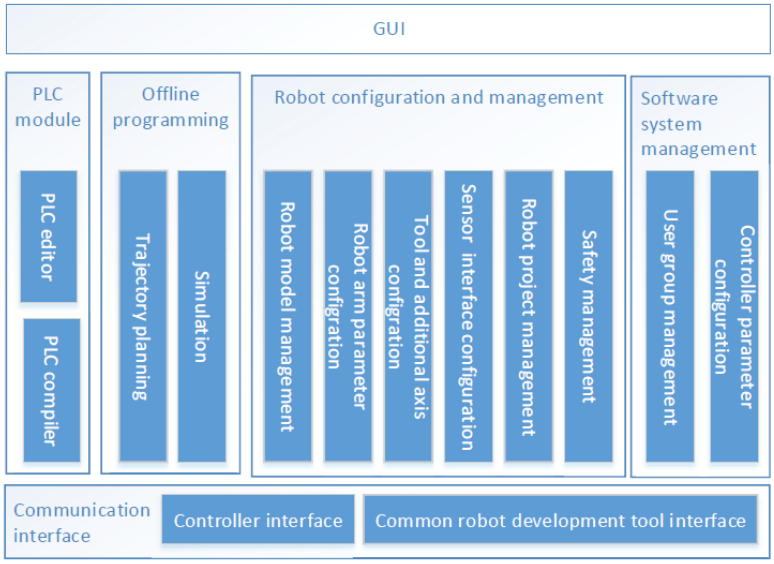


Figure 2.9: PMSS architecture (extracted from [1])

This contributes to the user-friendliness of the framework because, once selected a default model, the URDF file is generated automatically, according to the links and joint parameters defined.

The “Off-line programming” module resembles the special development environments. It is considered a key module of PMSS and provides off-line programming, motion planning, and simulation environments. Considering that this module is relatively complex, it is developed separately so that the system’s modularity and flexibility are not compromised. Moreover, to keep the low coupling degree between modules, “Off-line programming” module data is reachable via shared memory. The user can choose between programming in PMSS or importing external files through the PMSS API.

The PLC module generates the input files for the soft PLC module of a robot’s controller. This PMSS module allows editing and compiling such files based on Beremiz code. Lastly, the Communication interface module guarantees the connection between the PMSS framework and the robot controller. Additionally, it acts as an interface for external tools like simulation environments. Two protocols are available to interact with the robot controller – TCP and FTP.

The “Software system” module manages the PMSS functioning parameters.

In the final phase, a six-axis industrial robot arm was used to validate the implementation of the PMSS framework. Its movement was first tested in a virtual environment, and then the actual experiment was carried out. Despite the successful results in the experimental phase, the tests were not enough to consider the middleware as hardware-agnostic. Nevertheless, the proper operation of PMSS architecture was validated.

2.2.4 Summary

As a consequence of the fourth industrial revolution, enterprises are adapting to flexible, modular, and personalized manufacturing processes by integrating industrial robots into their assembly lines. However, due to the increasing complexity and flexibility of these processes, a wide range of industrial robots are being used, which also introduces new challenges regarding communication and programming. According to its purpose, each robot has diverse technologies, either in software or hardware, and even the communication protocols may differ.

The introduction of middleware will address this problem by developing an abstraction layer that handles hardware diversity and simplifies robotic applications development. It is considered an alternative to the specialised software platforms that allow higher robot integration and great system scalability.

Throughout the years, various contributions have emerged in the robotic middleware area. A complete survey of all existing middleware is obviously impossible due to the numerous existing solutions and constant release of new ones. With that being said, among the most relevant ones, middleware such as ROS, OROCOS, Miro, Orca, and MARIE were described and compared. ROS is the most suitable robotic middleware based on the comparison carried out in Section 2.2.2. Apart from the huge community that develops several software packages with drives and robotics algorithms, the ROS communication mechanism is advantageous for many applications. It supports synchronous communication with the service model and a publish-subscribe communication model through topics and messages. It also stands out for its ability of hardware abstraction. Nevertheless, not being real-time and lacking some security protections are the biggest drawbacks of this middleware.

Additionally, Profanter *et al.* [5] and Arnarson *et al.* [6] proposed a hardware-agnostic middleware based on the OPC UA standard. The OPC UA protocol is a de facto standard in manufacturing systems. The OPC UA middleware relies on OPC UA servers to communicate with the robots and OPC UA clients to provide robot control. It is possible to state that OPC UA allows semantically modeling information, and it was demonstrated that the skill module is suitable for robot control. However, the skill-based model introduces communication and synchronization overhead [24]. Moreover, when dealing with heterogeneous robots, the proposed OPC UA implementation is still complex and not as generic as desirable.

Lastly, Cao *et al.* [1] proposed a framework to combine the versatility of generic middleware with the performance of special development environments provided by the robot's manufacturers.

Chapter 3

Flexible and Modular Middleware for Robotics

This chapter represents the first approach to the problem based on what was studied in the previous one. It describes the progress from the solution design to the case study. Section 3.1 starts with an initial assessment where two potential architectures were dropped. Afterwards, an overview of the proposed solution is presented. Section 3.2 describes the two robots chosen as a case study to test the middleware layer. Lastly, Section 3.3 summarizes the content of this Chapter.

3.1 Proposed Solution

3.1.1 Initial Assessment

Given the analysis carried out in Chapter 2 about the existing middleware and also the recent OPC UA middleware approaches, two distinct frameworks were initially evaluated. It was intended to examine and conclude whether the considered frameworks had the potential to overcome the problem presented in Section 1.4. However, several aspects were addressed in this preliminary analysis, besides simply being able to solve the problem. The simplicity and modularity of the solution were highly relevant parameters in deciding its framework. As such, a solid foundation for future upgrades is more likely to be achieved.

The first framework reviewed for developing the middleware layer, capable of integrating different robotic manipulators, implied using ROS. ROS has several advantages over similar middlewares and is one of the most popular frameworks for robot application development. Therefore, some manufacturers such as ABB, KUKA, Universal Robots, and Techman already have ROS packages that support robot communication and control. As these packages are manufacturer-dependent, they are not compatible with each other, following different data configurations and implementing specific functionalities. Thus, developing a platform that allows the integration of heterogeneous robots alongside their ROS packages was required. Although it was a viable concept, the disparities of the ROS packages analyzed would hinder the modularity and flexibility of the solution. The first approach was dropped following a discussion with the company that

brought these problems to light. Furthermore, ROS currently only runs on Unix-based platforms, which was considered a significant drawback according to the industrial environment in which the company operates.

A second middleware architecture proposal emerged after reviewing the recent middleware approaches in Section 2.2.3. Based on such analysis, it became clear that the use of OPC UA as middleware delivered relatively optimal results. Since the OPC UA protocol is a standard in modern factories, using it as a middleware without ROS interference is a better industrial-based solution. However, the OPC UA architecture and robotic manipulators are still somewhat new to each other. Thus, the communication with the controllers is rather complex. Some robot manufacturers, such as Universal Robots, are beginning to offer licenses that allow using the OPC UA protocol for diverse applications. The OPC UA URCap license aims to intensify the development of Industry 4.0 with the help of UR robots, enabling them to connect to other OPC UA compatible devices [1]. Even though they considerably reduce the communication complexity and are completely customizable, these licenses were unsuitable for solving the problem. Apart from the fact that those licenses are relatively expensive, not all manufacturers offer them yet. For this reason, they do not contribute to developing a hardware-agnostic solution.

A different strategy to use the OPC UA protocol was to adapt and develop a hardware-agnostic and modular solution based on what has already been demonstrated by Profanter *et al.* [5]. This alternative was also excluded for legal reasons since it was covered by a copyright license, with all information contained in any files being property of fortiss GmbH. Therefore, the complex communication problem between robot controllers and the OPC UA protocol would have to be solved from scratch. For this, we would have to develop a similar approach as to [5] which, would be too time-consuming and therefore hinder the possibility of achieving all the objectives of this thesis (that go beyond communication), so the OPC UA approach was also dropped.

3.1.2 Solution Overview

Given the initial assessment mentioned above, a new approach was considered, which proved to be the one that best met the requirements, enabling us to achieve the objectives previously defined. Besides these objectives, there was an ongoing concern about the simplicity and modularity of the solution. Therefore, the middleware API is based on TCP/IP connections between the PC and the robot controller, used to retrieve the current robot's status and send commands with the user's intended action. In addition, with both experienced and inexperienced users in mind, an intuitive and user-friendly Graphical User Interface (GUI) has been developed. Essentially, any operator with reduced programming skills can still take advantage of the middleware layer.

Nonetheless, while experienced users may also benefit from using the GUI, the API content is what matters the most. It provides a set of functions that allow the user to monitor and control the robot remotely. In addition, its structure makes it easy to understand, customise and even modify

[1] https://www.universal-robots.com/plus/products/rocketfarm/opc-ua-clientserver/

the functionality of each module. In Fig.3.1, the overall architecture of the developed system is represented.

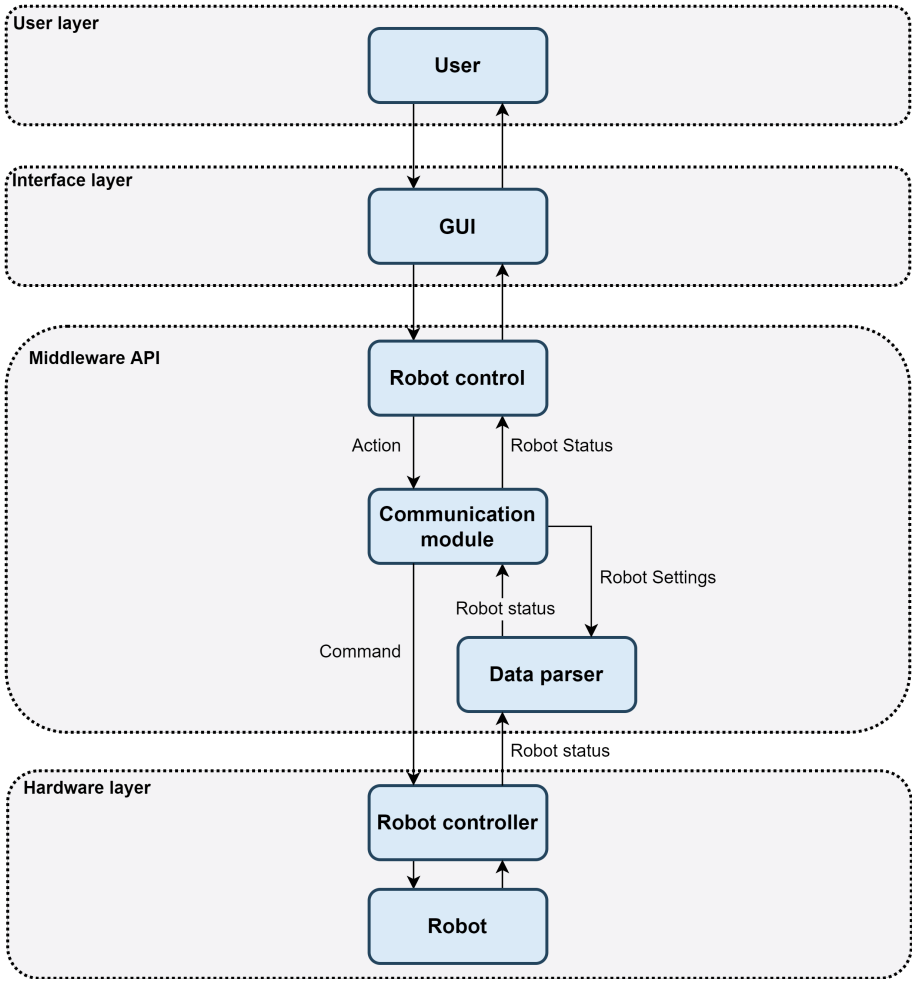


Figure 3.1: Overall architecture of the system

The user layer is on top of the stack and linked to the interface layer. It encapsulates the users' decisions regarding robot control and receives the relevant robot status information.

The interface layer contains the GUI, but as mentioned above, it is not the only way the user can interact with the API. Nevertheless, the GUI is still a valuable tool for communicating with the robot since it uses the API functions while avoiding the need to fill in the necessary parameters. The user-friendly interface further facilitates the robot's integration, but on the other hand, it does not fully support the functionality customisation. Although, if the functions are changed in the API source code, the GUI functionality will also follow those changes.

Regarding the middleware layer itself, it is comprised of a hierarchy of three main modules: Robot Control, Communication module and Data parser. The Robot Control module is on top of the hierarchy and is in charge of high-level control, and even receives the information regarding the robot's status from the modules bellow. It allows to adjust the robot's settings, such as speed,

acceleration, tool center point, as well as defining the trajectory of its movements. The Communication module contains everything needed to establish the connection with the robot being used. As such, it is in the middle of the hierarchy, between the Robot Control and Data parser modules. For high-level control actions sent from higher level modules, as can be seen in Fig.3.1, it communicates directly with the robot controller. This communication is specifically formatted according to the selected robot. Thus, the module is working as a translator that receives the user's action and sends the appropriate scripted command. As it is in between two modules, it operates as an intermediary module for the information sent by the robot controller. Lastly, the Data parser module is responsible for reading and correctly mapping the data received via TCP/IP connection. The information is received and converted into a standard layout used by the entire API.

The hardware layer contains the robot controller and the robot itself. It is at the bottom of the stack and essentially receives the commands sent by the API through a TCP/IP connection. In addition, as previously stated, it periodically sends the current state of the robot, which the API will then process.

3.2 Case Study

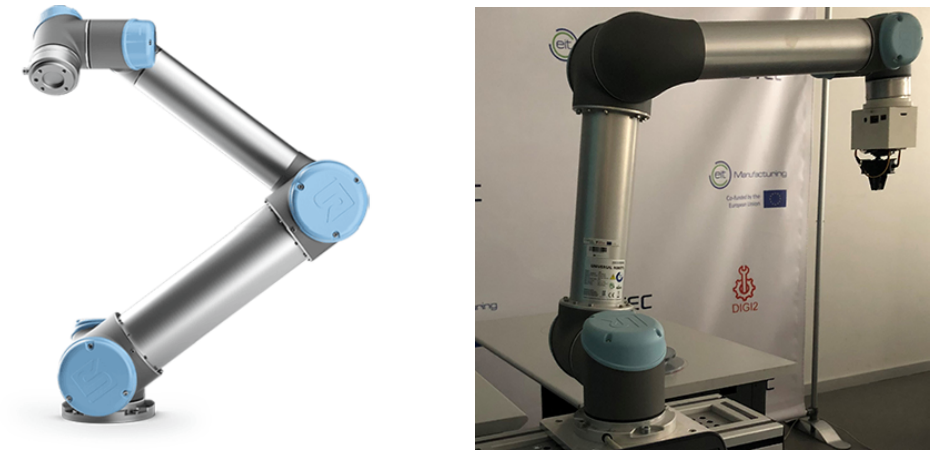
For the purpose of this dissertation, two distinct robots were chosen as a case study to test the developed middleware layer. The Universal Robots UR5 and the Techman TM12 are both collaborative robots but differ in several aspects. Universal Robots and Techman are the collaborative robot brands currently used by Controlar and are thus the most suitable. Firstly, the UR5 is an entry-level robot from Universal robots that provides an ideal balance between size and power. On the other hand, the TM12 is a high-level robot from Techman with superior accuracy and payload. Furthermore, it is also important to mention that slight differences in the intended purpose of each robot were noticed throughout the development. Universal Robots is a very open brand with a large community that, through forums and easily accessible documentation, develops applications independent of the default programming methods using proprietary software. In comparison, Techman focuses on industrial applications mainly based on its own software and has less information available compared with UR. This section will highlight more differences, proving the relevance of implementing the middleware layer mentioned above.

3.2.1 Universal Robots UR5

The UR5[2] is a flexible, lightweight entry-level robotic manipulator from Universal Robots. It is meant to operate in low-weight processing tasks and is designed for seamless integration in several applications. Given its simple setup, it is reasonably intuitive to program and offers a high level of flexibility. The robot was available for testing in FEUP's laboratory, which can be seen in Fig.3.2.

UR5 has 6 rotating joints, 850mm of working radius and a payload of 5 kg. It weighs 18.4 kg and every joint has a motion range of ±360º and a maximum speed of ±180 m/s.

[2] https://www.universal-robots.com/products/ur5-robot/



(a) Universal Robots UR5 (b) Universal Robots UR5 at FEUP

Figure 3.2: Universal Robots UR5

The Universal Robots software for robot programming is called PolyScope and runs on the teach pendant. The teach pendant has an intuitive graphical user interface that allows the user to control and program the connected robot, offering an intuitive programming method named node programming. As such, the user does not need to code and can control the robot by selecting nodes from a menu, corresponding to an instruction such as moving in space or setting a digital output. The programs created using the robot teach pendant's touch screen are called URP programs.

Although most tasks can be completed using the teach pendant and node programming, Universal Robots provides another alternative to program the robot. At the script level, the URScript programming language, like any other programming language, features variables, functions, loops, as well as several built-in functions used to control the robot. More experienced users can thus develop code and explore advanced functionalities. Fig.3.3 shows the PolyScope interface in version 3.4. It is not the latest version of the software, although it was the version currently installed on the robot at FEUP.

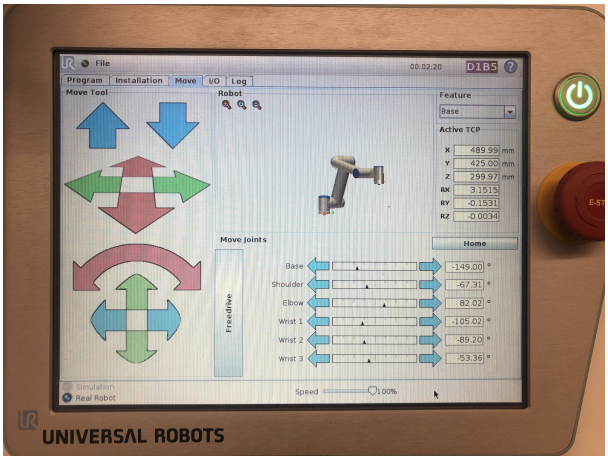


Figure 3.3: PolyScope interface, version 3.4

Despite not being directly supported by UR, remote control via TCP/IP is available using a primary, secondary, or even real-time client interface (RTDE). It is a less intuitive control option and intended for advanced users. Through a couple of sockets, the robot controller frequently supplies the robot's current state, such as the cartesian coordinates, joint angles, manipulator voltage, and so on. Therefore, with a custom external program reading the socket data, it is possible to follow this information. Nonetheless, the supplied data's shape depends on the software version and the selected client interface. The same sockets are available to receive URScript commands, completing the cycle of reading robot data and sending commands to control the robot.

Each interface has different port numbers and, consequently, different purposes. The primary client interface provides robot status information and communicates safety and error control messages. Moreover, the secondary client interface only provides the current robot state. Finally, the real-time client interface aims to synchronize external applications with the robot controller, complying with the real-time constraints. Both primary and secondary client interfaces have an update frequency of 10 Hz while the real-time has 125 Hz, but Universal Robots do not guarantee these values. Table 3.1 summarizes the remote-control specification.

Table 3.1: Overview of remote-control specifications

UR 5		
	Port number	Frequency (Hz)
Primary Client	30001	10
Secondary Client	30002	10
Real-time (RTDE)	30004	125

All data sent by the robot controller complies with the default system units. Distance variables are sent in meters, and angles are expressed in radians. Furthermore, it is also important to highlight that any pose is identified by [x, y, z, rx, ry, rz] where x, y and z represent the position in the coordinate system and rx, ry and rz the orientation of the tool head. In Universal Robots, the axis-angle with magnitude (Fig.3.4) representation is used for robot orientation. In short, the axis-angle representation expresses rotation through a unit vector and an angle theta. While the vector determines the direction of rotation, theta describes the magnitude of the rotation around it. When multiplying theta by the unit vector, we obtain the values of rx, ry and rz.

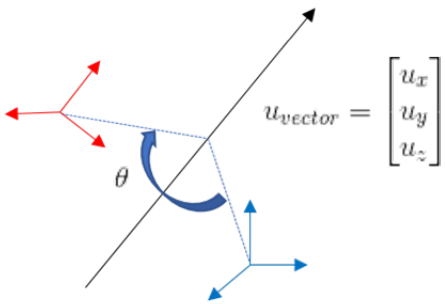


Figure 3.4: axis-angle with magnitude representation

3.2.2 Techman TM12

The TM12[3] is designed to increase productivity and quality of manufacturing processes, taking advantage of its great precision and flexibility. With mainly industrial applications in mind, Techman robots offer safe and efficient operations, thus being suitable for a wide range of industries. It belongs to the medium-heavy payload series of Techman and is also considered one of the most powerful collaborative robots of the moment. The robot was available for testing in Controlar (Fig.3.5)

(a) Techman TM12 (b) Techman TM12 at Controlar

Figure 3.5: Techman TM12

TM12 also has six rotation joints but can carry more weight than the UR5. It has a maximum payload of 12kg, a working radius of 1300 mm and weighs 33.3 kg. Although, unlike the UR5, the joints do not have a motion range of ±360º. Joints 1 and 6 are limited to ±270º, joints 2, 4 and 5 to ±180º, and joint 3 to only +- 166º. The joint's speed also varies for each one.

The Techman software for robot programming is named TMFlow and runs on a Windows PC. TMflow human-machine interface (HMI) is a user-friendly, fully graphical robot programming software based on flowcharts (also called TMflow projects). It is intuitive to program the robot via computer, and even those with no programming experience should be able to accomplish it. Each function is illustrated as an icon, ensuring that robot control is as simple as a drag-and-drop to generate new code. Therefore, the user sets the robot's parameters and uses the flow diagram to plan the robot's movement. Fig.3.6 and 3.7 show the TMflow interface in version 1.84 and the robot stick, respectively. The robot stick allows performing basic operations such as stopping or starting the project, changing the robot's speed and also includes the emergency button.

Like the UR5, the TM12 also provides remote control via TCP/IP. According to TM Ethernet Slave, when the robot is turned on, it establishes a TCP socket in which the controller will send the

[3] https://www.tm-robot.com/en/heavy-payload/

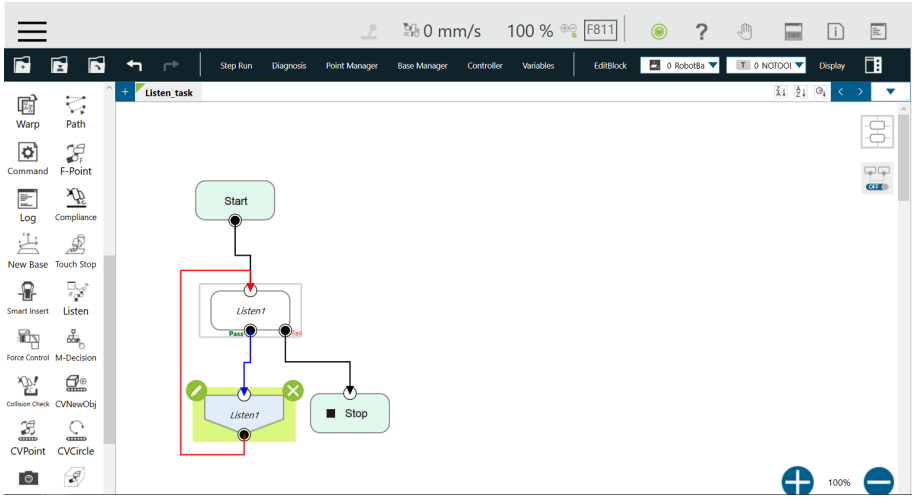


Figure 3.6: TMflow interface, version 1.84

Figure 3.7: Techman's Robot stick

robot's current status and other data. Therefore, it is possible to query and set several parameters of the robot from an external device, running a custom program. On the software manual of the robot, there is an "Ethernet Slave Data Table", which contains a set of variables sent by the controller to the TCP socket. The socket is opened in port 5891 and has an announced update frequency of 100 Hz with no guarantees. In addition to the TM Ethernet Slave, the External Script interface makes it possible to send motion-related commands to the robot. If the robot is on listen mode, remote computers can communicate with it and send properly framed packets according to the defined protocol. This communication is also established via TCP/IP on port 5890.

As expected, the data sent by the robot controller also complies with the default system units. Distance variables are sent in millimetres, and angles are expressed in degrees. Additionally, just like in the UR robot, the pose is identified by [x, y, z, rx, ry, rz], where x, y and z represent the position in the coordinate system and rx, ry and rz the orientation of the tool head. However, Techman robots express orientation with Euler angles or, more precisely, Tait-Bryan angles. Euler angles are one of the oldest rotation representations frequently used in robotics. Thus, rx, ry and rz indicate the value of the axis-aligned rotation, while the final orientation is a composite of them. The order of the composite is called convention. There are twelve possible combinations of conventions, but in this case XYZ is applied.

3.3 Summary

Throughout the early stages of the thesis development, several potential solutions were considered to address the problem outlined in Section 1.4. Each used different technologies, relied on specific requirements, and provided distinguished approaches to solve the same problem.

The first framework considered implied using ROS. The goal was to develop a framework that could easily integrate several ROS packages from each robot's brand. Nonetheless, ROS currently runs on Unix-based platforms and considering the industrial environment in which the company operates, it becomes rather inconvenient. Furthermore, after studying the packages' disparities and the solution's modularity, it was discussed with the company that this approach should be dropped.

The second middleware framework proposed was based on OPC UA and the work presented by Profanter *et al.* [5]. The overall objective was to use OPC UA, a de facto standard protocol in modern factories, to act as a middleware between the user and the robot controller. However, since OPC UA and the robotics field are still recently connected, there is a high degree of communication complexity. The OPC UA licenses provided by some robot manufacturers help simplify communication. However, since not every manufacturer owns them, it is unlikely to be suitable for a hardware-agnostic solution. Thus, the OPC UA approach was also dropped.

The third approach considered fulfilled the objectives, and throughout its development, there was a particular emphasis on the modularity and flexibility of the solution. It is based on TCP/IP connections between the PC and the robot controller and also has a GUI to help inexperienced users in the robotic field. The middleware layer itself has three main modules (Robot Control, Communication module, and Data parser) in charge of querying and updating the robot's internal parameters and sending motion-related commands.

Two robots were selected as use cases to test the developed middleware layer. The Universal Robots UR5 and the Techman TM12 are robots from different brands and ranges. Both are collaborative robots, but the UR5 is an entry-level robot with an ideal ratio between size and power, whilst the TM12 is a high-level robot with a higher payload and precision. Each one has its own software and various ways of being programmed, which overwhelms the user with several alternatives to operate the robot. Furthermore, specific aspects of each robot are different from brand to brand, such as the system units, the representation of the robot's orientation, and other details that will be mentioned in Section 4. All these variations make the user experience of programming and controlling the robot more complex, thus proving the relevance and utility of developing the middleware layer.

Chapter 4

Middleware Solution Development

This Chapter describes the development of the middleware layer based on the architecture illustrated in Fig. 3.1 of Section 3.1. Firstly, the specifications and requirements of the middleware layer are defined in Section 4.1. Afterwards, in Sections 4.2 to 4.4 the implementation of each module that composes the middleware is described in detail from the bottom to the top of the hierarchy. Finally, the structure of the GUI is presented in Section 4.5 together with the details of its implementation.

4.1 Specifications and Requirements

Given the requirements for simplicity and flexibility, *Python* was the programming language of choice for developing both the middleware layer and the GUI. It is highly flexible and versatile, meaning that it can be easily applied in multiple industries or fields. The Python community is continuously working to leverage the language even further with lots of libraries and frameworks that help the development cycle. The 3.9 Python version was used since it was the most recent version fully supported by all libraries.

Regarding the robot's software, the UR5 was running the PolyScope in version 3.4, the version currently installed on the robot at FEUP's laboratory. As previously mentioned, it is not the latest software version, but this does not affect or change the middleware's behaviour. PolyScope 3.15 is the latest version, and between each version Universal Robots fixes bugs and modifies some GUI functionalities on the teach pendant. Furthermore, the data stream broadcasted by the robot controller through the TCP/IP socket can also be slightly modified. Thus, between version 3.4 and version 3.15, the data structure of the streamed data suffered some changes. In the developed middleware layer, versions from 3.2 to 3.5 and 3.12 to 3.15 are directly supported. Versions 3.2 and 3.12 represent major software releases in which the data structure has suffered significant modifications. The TM12 was running the TMFlow interface in version 1.84, the most recent version to date. To communicate with the TM12 robot, a Python communication driver named

techmanpy[1] was used. This driver had two main requirements regarding the python and TMFlow versions. The TMFlow must be above 1.80, and the python version must be above 3.8.

The *techmanpy* driver has functions to query and update the robot's internal parameters and to send motion-related commands. In addition, it uses python's *Asyncio*[2] library for all socket communications, which leads to an efficient concurrent programming API design. It also ensures that all connections are successfully shut down and not kept open if unnecessary.

Regarding the UR5 setup, to make the robot ready to receive external commands in the TCP/IP socket it is enough to switch on the controller. Even if the robot is in idle mode, it still periodically broadcasts the robot's status to the middleware layer. Once the robot is running, if the controller receives an adequately formatted command, it executes immediately. However, the TM12 needs an initial setup before receiving commands from external devices. A project with the Listen node must be running for the TM12 to receive motion-related commands from external devices. In other words, it opens a socket on port 5980 and allows external script functions to be received. This project is easy to build on the TMFlow software by dragging and dropping the Listen node icon from the nodes menu. Once the project is ready, and the Ethernet slave is enabled, the user can run the project and make the robot ready to receive external commands. Fig. 4.1 shows an example of a project that makes the listen node to run. Furthermore, just like the UR5, even if the TM12 is idling and out of the listen node, it still broadcasts the robot's status.

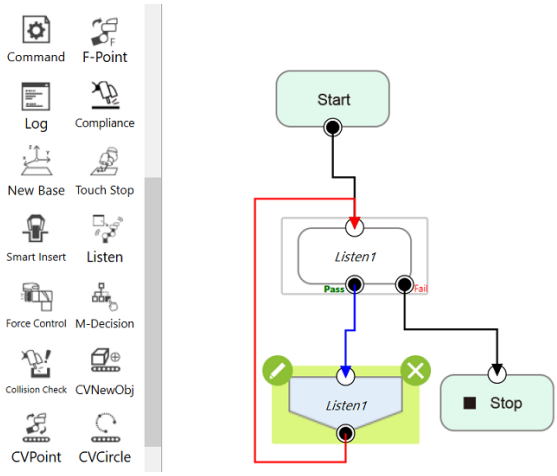


Figure 4.1: Listen node project example

4.2 Data Parser

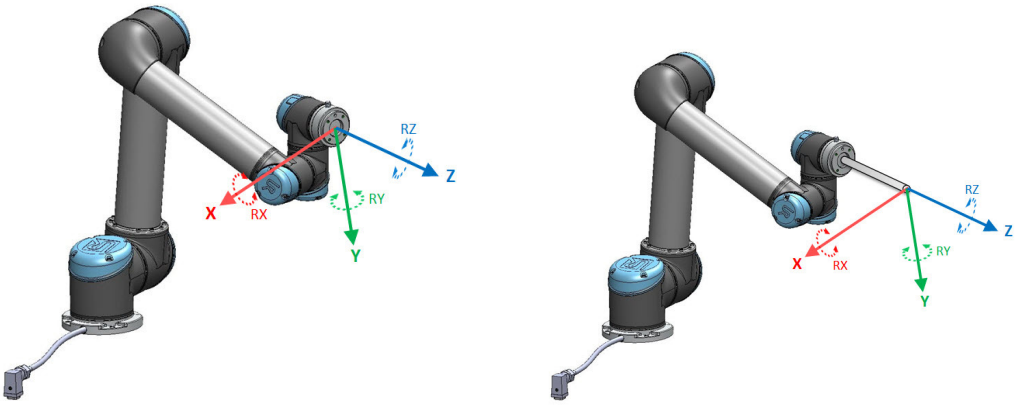
The data parser module is the middleware layer's low-level module, which continuously communicates with the robot controller. Its goal is to receive the streamed data broadcasted by the manipulator's controller and map it appropriately. As expected, each robot manufacturer is free to

[1] https://github.com/jvdtoorn/techmanpy
[2] https://docs.python.org/3/library/asyncio.html

define how the data is sent as well as its format. Therefore, there are several ways to receive and interpret the current state of the robot. So, the purpose of this module is to wrap up the complexity and standardize the data. In this case, and based on the two robots used to test the middleware layer, both robots broadcast the robot status through a TCP/IP socket. The socket contains binary information with UTF8 encoding. Then, information is accessed and interpreted through a decoding process.

The manufacturer defines the order in which the information message is sent and also which information is sent. In this manner, it is the data parser module's duty to interpret and distinguish the variety of incoming messages to suit the robot under operation. Even though the content of the messages is distinct from robot to robot, there is a set of common and relevant information across all of them. Therefore, the most interesting aspects of the handler's state were selected to create a standard layout. The whole API will use the standard layout, and regardless of the robot being operated, it makes the robot data structure in the middleware layer uniform.

The previously mentioned generic robot data structure is divided into four main parts. It has aspects regarding the robot status, joint-related data, cartesian coordinates value and tool center point configuration. The robot status part contains information about the project and robot state. As such, knowing if the project is running, if the emergency button is pressed, what is the current value of voltage and current consumed by the robot, among others, are aspects that belong to the robot status section. The joint-related data contains information about the joint angle and speed of each of the six joints of the manipulator. The cartesian coordinate value in this type of robot is represented by an array of 6 values [x ,y, z, rx, ry, rz], where x, y and z represent the position in the coordinate system and rx, ry and rz represent the orientation of the tool head. Lastly, the tool center point (*TCP*) configuration defines an offset between the robot flange and the actual robot's endpoint. When nothing is attached to the robot's flange, the end of the arm is used as a reference point for navigation. However, when a tool or gripper is installed, the endpoint is no longer the robot's flange but the tool's tip. Therefore, an offset should be added to set the necessary adjustments. Fig. 4.2 illustrates the *TCP* offset value scenario.



(a) No *TCP* offset

(b) Positive *TCP* offset in z axis

Figure 4.2: Tool Center Point offset value

Table 4.1 demonstrates and compares a couple of variable names used in the API with the variable name provided by each manufacturer. There are more variables besides the ones mentioned in the table regarding more specific and not as relevant aspects. Besides, there is also the possibility of adding more variables, even if they are specific to one brand. Given the arrangement and structure of the module, the middleware layer offers a great capacity for easy customisation.

Table 4.1: Standard Layout and robot brand variable comparison

API Standard Layout Variables		
API	**UR**	**Techman**
Project_running	Robot mode data: isProgramRunning	Project_Run
Project_paused	Robot mode data: isProgramPaused	Project_Pause
ESTOP	Robot mode data: isEmergencyStopped	ESTOP
Robot_voltage	Masterboard data: robotVoltage48V	Manipulator_Voltage
Robot_current	Masterboard data: robotCurrent	Manipulator_Current
Joint_angle	Joint data: q_actual	Joint_Angle
Joint_speed	Joint data: qd_actual	Joint_Speed
Cartesian_data	Cartesian info: x,y,z,rx,ry,rz	Coord_Robot_Tool
Tcp_info	Cartesian info: TCPOffset	TCP_Value

4.2.1 Universal Robots

Whilst not directly supported by the brand, UR provides remote control capability via TCP/IP. Through a primary, secondary or Realtime client, it is possible to monitor the robot's status remotely. Therefore, through the secondary client, a connection was established between the PC and the robot's controller via a TCP/IP socket. This client periodically broadcasts messages containing the robot's status information on port 30002.

To access and subsequently interpret the information received, it is necessary to go through a decoding process. The structure of the message sent by the robot controller is detailed in the remote control documentation provided by the brand. Fig. 4.3 summarizes the structure of the byte stream.

Figure 4.3: Universal Robots Secondary Client package structure

The package starts with a parcel specifying the total size of the package sent, and it is especially useful for detecting data losses. It is followed by a parcel containing the message type of the whole package. There are several message types, but the most common and relevant are the 16, 20, and 25. Message type 16 denotes the package containing data about the robot's status, whereas 20 and 25 are reserved for additional security or project status messages. However, because the socket is connected to port 30002 (secondary client), only messages regarding the robot state are

periodically broadcasted. Subsequent to these two parcels that contain general message information are the sets of Sub-Packages with their respective payload. Each set of Sub-Packages contains three parcels concerning length, type and content. Once again, the length of the sub-package is useful for error checking, while the package type acts as an identifier for the content being sent. In other words, it associates an integer number according to the different batches of information sent, such as Robot mode data, Joint data, Cartesian info, Masterboard data, Tool data, and so on. Lastly, the data parcel holds the robot's state content with variables of several types (int, char, float, etc.). Table 4.2 summarizes the message and package types of Universal robots.

Table 4.2: Universal Robots Message and Package Types

Message Types	
Robot State	16
Robot Message	20
Program State	25
Package Types	
Robot mode data	0
Joint data	1
Cartesian info	2
Masterboard data	3
Tool data	4
Kinematics info	5
Configuration data	6
Force mode data	7
Additional info	8

Each data parcel is associated with one sub-package and contains multiple variables holding information about the robot. Table 4.3 shows the content of the Robot mode data package as an example.

Table 4.3: Robot mode data Sub-Package

Robot mode data	
uint64_t	timestamp
bool	isRobotConnected
bool	isRealRobotEnabled
bool	isPowerOnRobot
bool	isEmergencyStopped
bool	isProtectiveStopped
bool	isProgramRunning
bool	isProgramPaused
unsigned char	robotMode
unsigned char	controlMode
double	targetSpeedFraction
double	speedScaling
double	targetSpeedFractionLimit

A python function is then used to handle the binary data from the network connection. However, to correctly identify which type of variable is being received, a string must be taken as a parameter of the function. The string starts with a "!" to denote the network byte order (big-endian), followed by one character for each variable type. Table 4.4 shows some characters associated with a variable type. For instance, the Robot mode data sub-package had the following string: "!Q???????BBddd".

Table 4.4: Variable type and string character association

String characters	
char	c
unsigned_char	B
bool	?
int	i
long	l
float	f
double	d

In short, processing the message sent by the controller starts by identifying the type of message and then parsing and correctly mapping the data of each sub-package. This process is repeated whenever a new message is received by the middleware layer.

4.2.2 Techman

Similar to Universal Robots, Techman also provides remote control capability through the TM Ethernet Slave. Based on client/server connections, when the robot is turned on, it establishes a TCP/IP socket to send the current robot's state or to receive and update internal parameters. In port 5891, the robot periodically broadcasts messages containing such information. In the TMFlow software, it is possible to configure the communication protocol and see all available internal parameters. Also, in the software manual of TMFlow, there is a complete Ethernet Slave Data Table which contains every parameter's value that is possible to obtain from the robot controller. Fig. 4.4 shows a portion of this table.

The structure of the message sent by the robot controller is specified in the "Expression Editor and Listen Node" manual of Techman. Nevertheless, Fig. 4.5 summarizes the structure of the byte stream.

Following the start byte lies the header, which defines the purpose of the communication packages. The header can either be "TMSVR" if the message contains data regarding the robot's state or "CPERR" if the message contains error information such as packet error, checksum error, and so on. Next, separated by commas, comes the parcel indicating the size of the data field. The data parcel is the content of the communication package and uses UTF8 encoding. If the header is "TMSVR", the data parcel is further divided into three segments separated by commas. Otherwise, it only contains the error code of the Communication Protocol Error. The Transaction ID is just a numerical identifier of the message. In addition, the mode defines the format of the data

Item Name (ID)	Description	Type	Size	*A	**W	Note
Robot_Error	Error or Not	bool	1	R		Yes:1 No: 0
Project_Run	Project Running or Not	bool	1	R		Yes:1 No: 0
Project_Edit	Project Editing or Not	bool	1	R		Yes:1 No: 0
Project_Pause	Project Pause or Not	bool	1	R		Yes:1 No: 0
Get_Control	Get Control or Not	bool	1	R		Yes:1 No: 0
Safeguard_A	Safety IO (Safeguard Port A trigger)	bool	1	R		Triggered: 1 Restored: 0
ESTOP	Emergency Stop	bool	1	R		Triggered: 1 Restored: 0
Camera_Light	Light	byte	1	R/ W		Enable: 1 Disable: 0
Robot_Model	Robot Model	string	1	R		
Error_Code	Last Error Code	int	1	R		Format: Hexadecimal
Error_Time	Last Error Time	string	1	R		Format: [YYYY]-[MM]-[DD]T[hh]:[mm]:[ss.sss]
Coord_Base_ Flange	Cartesian coordinate w.r.t. current Base without tool	float	6	R		Unit: mm
Joint_Angle	Joint 1 angle - Joint 6 angle	float	6	R		Unit: degree
Coord_Base_ Tool	Cartesian coordinate w.r.t. current Base with tool	float	6	R		Unit: mm
Coord_Robot_ Flange	Cartesian coordinate w.r.t. Robot Base without tool	float	6	R		Unit: mm
Coord_Robot_ Tool	Cartesian coordinate w.r.t. Robot Base with tool	float	6	R		Unit: mm
	Cartesian coordinate					

Figure 4.4: Portion of the Ethernet Slave Data Table

content. Modes 1/2/3 are for reading the broadcasted data periodically, whereas modes 11/12/13 are for asynchronous reading requests. Furthermore, modes 1 and 11 indicate that the content data type is in binary format, modes 2 and 12 in string format and modes 3 and 13 in JSON format. Lastly, the content parcel contains the data formatted by the mode definition.

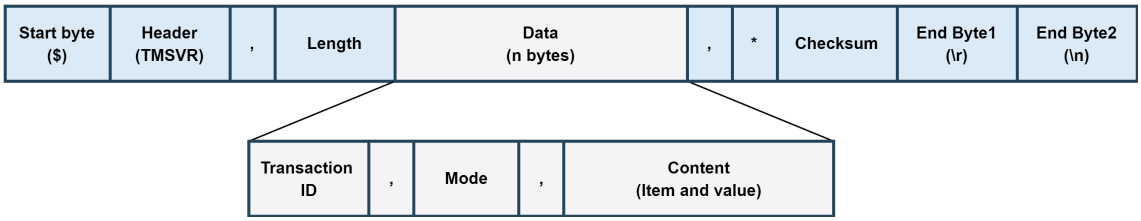


Figure 4.5: Techman's TMSVR package structure

However, querying internal parameters with the *techmanpy* communication driver for Techman robots is as simple as setting up the connection with the TMFLow server through a socket TCP/IP and then executing a function called "get_values". It takes a mandatory argument containing the variable names from the Ethernet Slave Data Table and returns the requested internal parameters along with their values. Essentially, the driver sends a message with the "TMSVR" header and

mode 12, but it handles the framing and interpretation of the message. The received data is then correctly mapped into the API standard layout.

4.3 Communication Module

The communication module is in charge of the functions that establish the connection with the robot being operated. It also holds aspects related to TCP/IP socket configuration, such as host and port, timeout values, and so on. Since it is in the middle of the middleware's layer framework, it plays the role of an intermediary for both the Data Parser module, which transmits the robot's state information, and the Robot Control module. Furthermore, it analyses the robot's current state through the feedback of the data parser module and informs the Robot control module if the robot is ready to receive motion-related commands or not. This is simply achievable by checking the value of the "Project_running" variable in the standard API layout.

It is also in this module that the Universal Robot's commands are shaped and sent. Depending on the user action processed in the Robot Control module, the instruction to send to the robot must be properly formatted according to the documentation. Therefore, before sending the command in the TCP/IP socket, the action goes through some preparatory steps. The following section will describe the proper format for each message in detail. Lastly, to send the command to the TCP/IP socket, it is required to convert the string to a byte stream using UTF8 encoding.

The techmanpy driver mainly handles the communications with the Techman robot. Even so, the communication protocol of the Techman robot can be divided into four parts according to its application. TMSVR, CPERR, TMSTA and TMSCT are the communication protocol headers that define the communication package's purpose. The content of the message is different according to each header.

The TMSVR header is defined for the TM Ethernet Slave protocol, meaning it is used for communications regarding the robot's current state or other internal parameters, as it was already described in Section 4.2. The CPERR header defines the communication packages containing protocol errors, in which the message data section carries the error code. The TMSTA header was not used throughout the development of the middleware layer. Nevertheless, it is employed in querying project-related internal parameters, that is, whether the robot is running any program and its state. Lastly, alongside TMSVR, the TMSCT is the most commonly used header, making it possible to send motion-related commands to the robot. This header defines the communication package as External Script Language.

If the robot is operating in listen mode, with the External Script feature, it is possible to remotely send motion-related commands through a TCP/IP socket in port 5890. When the robot enters the listen node, external devices can connect to it and communicate based on the package format. The structure of the byte stream is similar to the TMSVR detailed in Section 4.2, with minor changes on the data parcel. Fig. 4.6 summarizes the structure of the TMSCT byte stream.

On the communication protocol with the TMSCT header, the data parcel is only divided into two segments, separated by commas. The Script ID is an arbitrary number used as an identifier

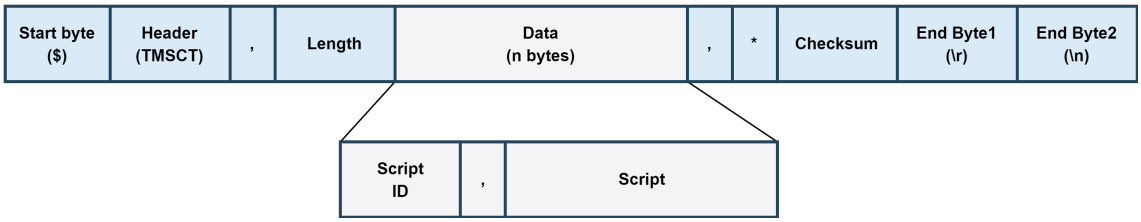


Figure 4.6: Techman's TMSCT package structure

of the message. The script parcel contains commands in Script language, such as robot motion functions. The "Expression Editor and Listen Node" manual has built-in functions that can be sent remotely but only if the robot is in the listen node and the TMSCT header is used.

In short, in order to connect and move the Techman robot, the listen mode must be active. Then, with the external script feature, a range of functions provided by the manufacturer can be used to control the robot remotely. The packages sent with this purpose must have the TMSCT header.

4.4 Robot Control

The robot control module is the core of the middleware layer and lies on top of the other two modules that compose the API. It is in charge of high-level control, interpreting the user's action and subsequently appropriately communicating with the robot. The robot's status information parsed by the Data Parser module is also relevant since some functionalities rely on the actual position of the tool's center point. Therefore, it is required an ongoing communication between these two modules.

Within this module, the user can specify the robot he wants to control. The user will be required to identify the robot manufacturer, the robot model and the software version, as well as the right host and port to establish communication. These aspects verify if the middleware layer directly supports the robot in operation or if any additional changes need to be made. Robots from Universal Robots and Techman were used during the development of the API. Thus, these two manufacturers are fully supported. In addition to the parameters entered by the user, the middleware layer defines a group of additional parameters. Among these parameters are the home position, the tool center point value, system units, joint velocity and acceleration. The joint speed and acceleration are set to safe values, the system units are set according to the manufacturer and default values are assigned to other parameters like the home position and tool center point value.

However, although the middleware layer was only tested with two robots, it was enough to identify their disparities. Different configurations are adopted depending on the manufacturer, even in common aspects between the two, such as distance and angular units or even the speed and acceleration of the robot. These aspects should be standardized throughout the middleware layer, ensuring the flexibility and user-friendliness of the API.

For the Universal Robots, all distance measurements are expressed in meters and angle measurements in radians. That is, the robot's controller broadcasts the robot's state following that configuration. For example, the cartesian coordinates are sent with x, y and z in meters and rx, ry and rz in radians. Furthermore, the controller also expects to receive motion-related commands with those appropriate system units. On the other hand, Techman robots do not follow this configuration and express distance measurements in millimeters and angle measurements in degrees. Consequently, the system units are one of the aspects to be standardized.

When the user selects a robot to operate, the middleware layer establishes its units according to the manufacturer's default units. For instance, if the robot is from Universal Robots, the middleware layer will assume that all distance measurements will be presented in meters and angle units in radians. Nonetheless, the user is free to choose which unit's configuration should be used throughout the robot's operation. It is possible to choose which unit configuration should the middleware API consider, and then, if necessary, the values are converted.

Once the user defines the unit configuration of the API, those values are compared with the manufacturer's default values, and if they differ, the middleware layer applies the appropriate conversion. The conversions are quite simple since 1 metre is 1000 millimetres and π radians are 180 degrees. Fig. 4.7 illustrates an example of operating with the UR5 but with the middleware layer expressing distance and angles in millimetres and degrees.

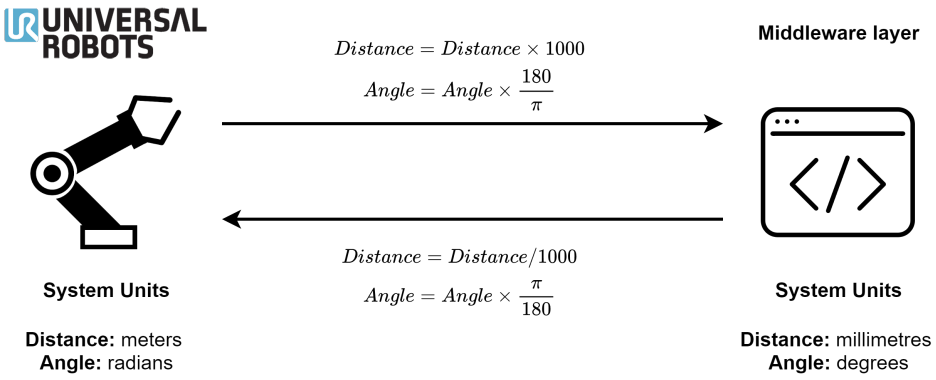


Figure 4.7: Unit conversion example with UR5

According to the source of data, there are two types of conversion to do. If the robot sends the data, it is expressed in meters or radians and must be converted to millimeters or degrees. If, however, the information is from the user input, it must be converted to meters or radians before being sent to the robot.

The joint speed and acceleration definitions are also different between the two robots studied. Universal Robots uses the standard definition of velocity and acceleration expressed in m/s and m/s^2, respectively. On the other hand, for the functions supported by the Techman robots' external script, speed is expressed as a percentage of maximum robot speed and acceleration as the maximum time to reach the desired speed. Thus, for Techman robots, the speed is expressed

in percentage and acceleration in milliseconds. As such, the robot's speed and acceleration are additional factors that need to be homogenised in the middleware layer.

The robot's speed and acceleration in the middleware layer are then given as a percentage of the maximum robot speed and the rate of change of the velocity with respect to time, respectively. All manufacturers specify the maximum speed of the robot, so representing the speed as a percentage of that value is an intuitive and easily convertible option. Acceleration is expressed in m/s^2, so, for Techman robots, it is required to perform the conversion before sending any command. Since Techman robots expect to receive the maximum time to reach the desired speed as the acceleration value, the conversion is possible using the acceleration formula. Fig. 4.8 summarises the conversion between the middleware and the Techman robot acceleration value.

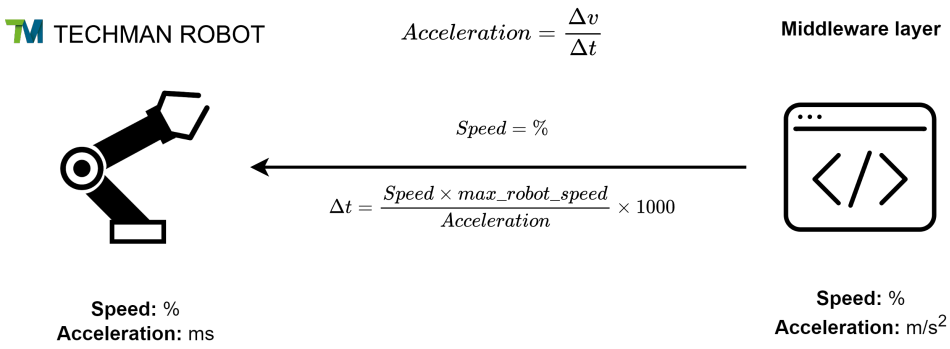


Figure 4.8: Speed and acceleration conversion example with TM12

Considering the acceleration formula, if the value of the velocity change is the value of the desired speed of motion, for a given acceleration value, the time change is the maximum duration of the acceleration. Thus, it is the time change (in milliseconds) that Techman robots expect to receive. Furthermore, as mentioned above, the desired speed of motion is obtained by multiplying the current speed percentage by the maximum robot's speed.

For each API functionality, there is a generic Python function. Once the user defines the robot to operate, the middleware layer decides how to format the message and communicate with the robot. Although they may have different purposes, most functions are designed with the same structure and have the same set of parameters. This structure achieves a high degree of flexibility and modularity while keeping the middleware usability straightforward. Table 4.5 briefly summarises the supported functionalities of the middleware.

Settings

Besides what would be possible to configure in the proprietary software of each robot, in the middleware layer, it is also possible to configure aspects such as the system units and robot speed to the user's preference. As previously mentioned, each robot manufacturer determines how to express the system units and the robot's movement speed. Although having the middleware layer, the user is able to choose how to express the distance and angle units. The middleware layer also

Table 4.5: Middleware functionalities

Settings
System units configuration
Robot speed and acceleration
Tool Center Point configuration
Configure home position
Set robot internal parameters
Motion-related commands
Move joints to target
Move linear to target
Move relative to the tcp
Precise movement
Freedrive

standardizes the speed and acceleration values of the robot. This way, the user sets those values to personal preference, and the middleware takes care of the required conversions.

The middleware layer offers a function that takes as a parameter the offset with respect to the robot flange [x, y, z, rx, ry, rz] and sends the shaped command to the robot in order to configure the tool center point value. For the UR5, the string must be formatted as described in the scripting language manual. It is required to convert the values [x, y, z, rx, ry, rz] into a string with the prefix "p" with each of them between square brackets. As shown below, this string must now be placed within a larger string with the "set_tcp" script command.

```
set_tcp(p[x, y, z, rx, ry, rz])
```

Then the string command must be encoded and sent through the TCP/IP socket. For the TM12, the techmanpy driver has a function that sets the Tool Center Point to a custom configuration. Firstly, the connection is established with the external script interface (TMSCT header) and then a function also called "set_tcp" is executed with the offset [x, y, z, rx, ry, rz] as an argument.

The robot's home position can also be configured. Even though the home position is assigned a default value when the user specifies which robot is being programmed, this value can be changed to a more convenient position. The home cartesian coordinates are stored in the API and can later be used as a motion target.

Motion-related commands

The middleware layer has several functions that allow operating the robot according to its purpose. There are different types of motions, frame changes and some other versatile functionalities for programming the robot.

The first one to mention is a core feature of robot programming that allows moving the end-effector from point to point. It is called "move_joints" and is especially suited when the motion does not need to be linear. That is, the controller estimates the joint angles that correspond to the target point and then executes the motion, resulting in a curved path for the tool. The API function

receives as parameters the target's cartesian coordinates and the speed and acceleration values that the robot must follow. These parameters are common to most existing functionalities, contributing to a modular and flexible framework.

Depending on the robot being programmed, the middleware layer decides the message structure and how to communicate with it. For Universal Robots, the message must be formatted according to the UR scripting language. Therefore, the cartesian coordinates of the target point [x, y, z, rx, ry, rz] must be transformed into a string with a p prefix, with each value between square brackets. The prefix p specifies to the controller that the array represents a cartesian target point. Lastly, the point string is a sub-string of the whole command that is framed as shown below:

```
movej(p[x, y, z, rx, ry, rz], a = value, v = value, r = 0)
```

In this type of motion, `a` expresses the joint acceleration in rad/s^2 and `v` the joint speed, given in rad/s. The `r` parameter is the blend radius in meters, and since it is most useful when a continuous motion between points is needed, it was not used for this function. After being correctly formatted, the string can be encoded and sent through the TCP/IP socket.

The techmanpy driver provides a function to execute the point-to-point motion for Techman robots. It is called "move_to_point_ptp" and takes the target value, robot speed and acceleration as arguments. Therefore, the middleware layer uses Techman's external script interface by establishing a TMSCT connection. Thus, the driver formats the message according to the software manual and sends it through the TCP/IP socket on port 5980.

Moreover, in the point-to-point motion, the target does not always have to be described by the cartesian coordinates and tool head orientation. The middleware layer accepts the end-effector's target as the angles of each handler's joints. For that, the function interprets the target as joint angles and proceeds to edit the commands slightly. For Universal Robots, removing the prefix "p" from the target's string is the only change required. Without this prefix, the controller interprets the array as the joint values to be achieved. For Techman, the tweak is as simple as calling the function named "move_to_joint_angles_ptp" instead of "move_to_point_ptp". This change indicates the external script function that the motion target format will be expressed in joint angles rather than cartesian coordinates.

There is one more relevant detail regarding motion commands, which happens if two commands are sent in a row. If two commands are sent consecutively, the robot does not wait for the first one to finish before executing the next one. Hence, it is necessary to introduce a holding timer so that the robot only executes the second move after reaching the first action's destination. The waiting time cannot be arbitrary because the movements have no fixed duration. For this reason, the middleware layer has a function that signals if the robot has already reached the target. The current position of the robot end-effector is frequently sent by the data parser module and subsequently compared with the target value. Once the Euclidean distance between the current position and the target point is calculated, it is compared with a given threshold. Distances smaller than the threshold indicate that the robot has finished the movement. If the target is expressed in joint angles, the distance can also be calculated through the difference between the current and

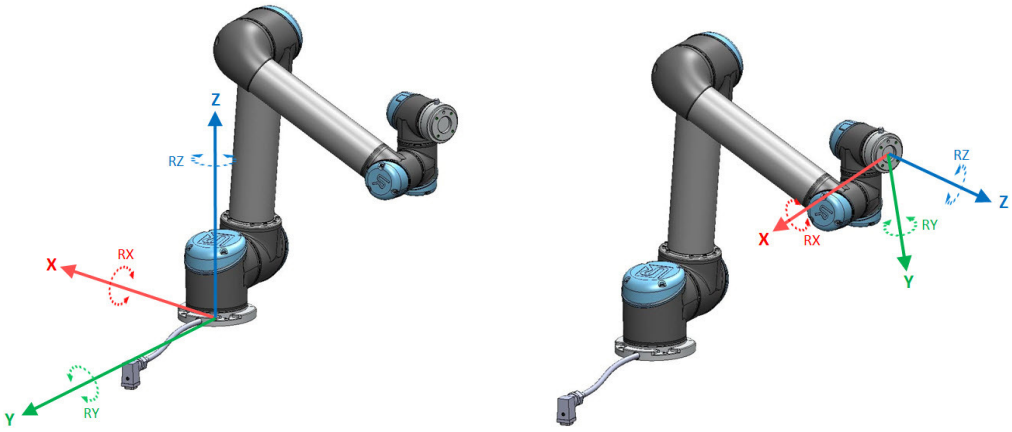
desired angles. Even though the threshold has an assigned default value, it can also be configured by the user. In addition, the techmanpy driver for Techman robots provides another alternative using queue tags. It is possible to set a queue tag for the last-sent command, and then with the function "wait_for_queue_tag", the middleware layer waits for the robot to finish executing the request before sending the following command.

Another similar type of motion also supported by the API is the "move_linear". This function is used when the end-effector's motion needs to be linear and precise. The structure of the message and communication methods are almost the same as the "move_joints" described above. Therefore, as shown below for Universal Robots, it is only needed to replace the keyword "movej" with "movel".

```
movel(p[x, y, z, rx, ry, rz], a = value, v = value, r = 0)
```

In this type of motion, `a` expresses the end-effector acceleration in m/s^2 and `v` the end-effector speed, given in m/s. For Techman robots, it is as simple as changing the function to call from the tehcmanpy driver, which also changes the function on the external script interface. Since it is a linear motion, the driver function to call is called "move_to_point_line". Apart from the changes mentioned, this linear movement is similar to the joint movement and shares the same properties.

Both joint and linear motion types receive as parameters the target with respect to the base coordinate system. That is, the cartesian coordinates of the point are defined with respect to a frame whose origin is located at the center of the Base joint. However, there are more referential frames besides the base one, like the tool coordinate system. In this coordinate system, the y-axis is away from the tool connector, and the z-axis is normal to the tool flange. Fig. 4.9 illustrates the differences between both coordinate systems.



(a) Base coordinate system (b) Tool coordinate system

Figure 4.9: Manipulator coordinate systems

Thus, it is also convenient to allow the robot to move relative to this reference, that is, relative to the tool center point. The middleware layer provides a function that moves the robot relative to the tool center point and also takes the target value [x, y, z, rx, ry, rz], robot speed and acceleration

as arguments. Moreover, it is crucial to notice that this target is distinct from the previous ones since it is a relative and not a global position. The relative position is described with respect to the tool frame, whereas the global position is with respect to the base coordinate systems. For Universal Robots, the command must be formatted as follows:

```
movej(pose_trans(get_forward_kin(), p[x, y, z, rx, ry, rz]),
a = value, v = value, r = 0)
```

The first argument of the "movej" command is the target pose. By default, it is always with respect to the base frame. So, suppose the current position of the end-effector and the relative target are known. In that case, it is possible to represent the relative target with respect to the base frame and use the result as the first parameter of the "movej" command, which is precisely what the pose_trans function is responsible for. It returns the relative target as a pose with respect to the base frame. The first argument of the pose_trans function is the starting pose, and the second is the pose of the relative target. Therefore, considering that get_forward_kin() returns the current tool center point pose, as formatted above, the pose_trans command will return the target point in the base coordinate system. With the target value calculated, the motion relative to the tool center point is similar to "move_joints" and follows the same procedures. For Techman robots, the external script interface already provides a function which allows movements relative to the tool. The techmanpy driver properly formats the message and uses the "Move_PTP" function from Techman's external script list. Consequently, after establishing the TMSCT connection with the robot, the techmanpy move_to_relative_point_ptp function can be used.

Although the target in this type of movement is relative to the tool centre point, the middleware layer still needs to calculate the cartesian coordinates of the endpoint with respect to the base frame. This result is required to judge when the motion is complete. If the poses [x, y, z, rx, ry, rz] were regarded as transformation matrices, the global target would be calculated as follows:

$$T_{\textit{target w.r.t base frame}} = T_{\textit{current pose w.r.t base frame}} \times T_{\textit{target w.r.t current pose}}$$

The transformation matrix T can be defined as a 4-by-4 matrix, with R being the rotation matrix and p the position vector. In other words, the matrix R represents the rotation by combining rotations in the x, y and z direction, and p is the vector containing the [x, y, z] values.

$$T = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix} \quad R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \quad p = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

The cartesian coordinates of the endpoint with respect to the base frame are the `p` sub-matrix of $T_{\textit{target w.r.t base frame}}$. It is also important to mention that the method of obtaining the rotation matrix `R` is not the same between UR and Techman, as they use different representations of rotation.

For UR, the rotation matrix should be obtained from the rotation vector [rx, ry, rz], whereas for Techman should be obtained from the Euler angles [rx, ry, rz].

Alongside the movement options mentioned so far, it is also beneficial to operate the robot with precise and small movements for delicate tasks or even fine adjustments of the robot's position. That is the purpose of the API's precise_move function. The user can move precisely in either cartesian space or gradually increment each joint angle. By default, the increment is set to 1 mm or 1 degree if the movement is in cartesian space or with joint angles, respectively. Either way, the user can also set which increments to use each time the function is called as a function parameter. For Universal Robots and Techman, the data parser module sends the current cartesian coordinates each time the function is called. Then the increment in the selected direction is added to the current position, and the API formulates a simple "move_joints" command with the calculated target. With this method, it is possible to obtain precise control of the robot's position, which is helpful in several scenarios.

Since the tested robots are collaborative, both have a free drive mode. The free drive mode allows the operator to move the robot joints to the desired position by hand. For UR robots, there is a button on the teach pendant that, whenever it is pressed, the robot's joints are unlocked which allows them to be moved by hand. There is the same button for Techman robots, but it is situated on the robot's end-effector. Howeve, only Universal Robots can activate the free drive mode remotely. Therefore, the middleware layer provides a function that enables the UR free drive mode by sending a UR Script command `freedrive_mode()` through the TCP/IP socket.

4.5 Graphical User Interface (GUI)

The primary target group of the Graphical User Interface (GUI) are the less experienced users in the robotics field or even those with inferior programming skills. However, even the most experienced users can take advantage of the GUI since it is a useful tool for programming robots based on the middleware layer developed. Given that it uses the API functionalities, the user-friendly interface simplifies the integration of different robots by avoiding the need for coding and dealing with multiple functions.

Because the middleware layer was developed using Python, the goal was to build the GUI using a python framework to guarantee high flexibility and compatibility. Moreover, throughout the GUI development, there was also an ongoing concern about the modularity and scalability of the solution. Thus, since there are numerous alternatives to GUI python frameworks, it was required to examine their advantages and disadvantages according to the purpose of the GUI. As such, some of the most popular frameworks such as Tkinter[3], Kivy[4] and PyQt5[5] were considered. Tkinter is a lightweight and straightforward framework built in the Python standard library. Due to its simplicity, it is often used to develop an interface quickly. Unfortunately, Tkinter interfaces

[3] https://docs.python.org/3/library/tkinter.html
[4] https://kivy.org/#home
[5] https://pypi.org/project/PyQt5/

have an old-fashioned design and do not offer a range of elements as wide as other frameworks. Since the objective was to develop a modern interface, the Kivy framework, a cross-platform open-source Python library, was also a suitable alternative. With Kivy, the same interface can be deployed to desktop, web and even mobile applications. Although it is a viable option, only a desktop interface is needed, and the development complexity is much higher when compared to the PyQt5 framework. PyQt5 is based on the Qt framework, which is also cross-platform. To support the design and development of the GUI, QtDesigner has several elements that the developer can drag and drop to build the interface. Thanks to its simplicity and scalability, PyQt5 is the chosen framework to develop the GUI and the one that best suits this application.

Using QtDesigner or writing plain python code are the two possible ways to build the GUI using PyQt5. QtDesigner is a tool from the Qt framework that speeds up the GUI development process due to its simple drag-and-drop interface. It provides a list with several elements like buttons, labels, input widgets, and so on that can be styled and placed as desired. Fig. 4.10 shows the QtDesigner window used to design the GUI.

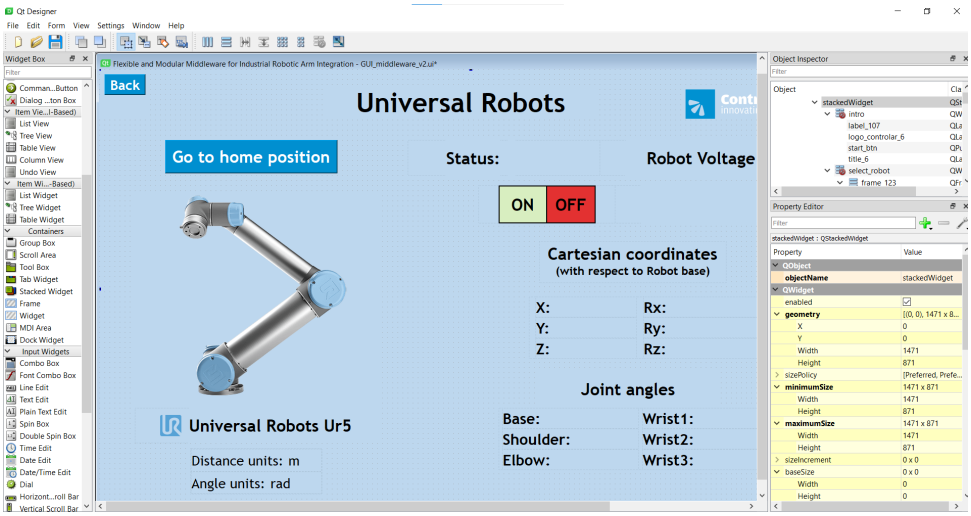


Figure 4.10: QtDesigner window example

Therefore, the front-end component of the GUI was developed in the QtDesigner interface using the available widgets. QtDesigner is platform independent as the design results in a *ui* file. The *ui* file contains the GUI design and style properties in an XML representation specifying which widgets were used. Afterwards, using the *pyuic5* command line tool from PyQt, it is possible to translate the *ui* file into an equivalent python file. This python file is similar to the code that would be obtained from hand programming the GUI from scratch. The *pyuic5* command should be executed as follows:

```
pyuic5 -x gui.ui -o gui.py
```

The python file is now a self-executable standalone application with the design created on QtDesigner. However, the GUI in that condition does not have any functionality because it only

features the front-end component. The GUI's back-end component was then developed from hand-coding using the front-end elements. The back-end component covered aspects like button actions, page navigation and connection with the middleware layer. It consists of three areas divided between threads. The main thread is in charge of GUI responsiveness, that is, navigation across pages, button clicks, etc. A second thread constantly receives the robot's state sent by the data parser module and updates the values in the interface. Lastly, there is a third thread dedicated to executing the functions of the Robot Control module.

Fig. 4.11 shows the first page to pop up when the GUI is launched. Since the interface allows to program several robots, the first action is to choose the desired robot. In this specific application, Universal Robots and Techman are directly supported brands. Besides the brand, the user also specifies the robot model, the software version and communications details.

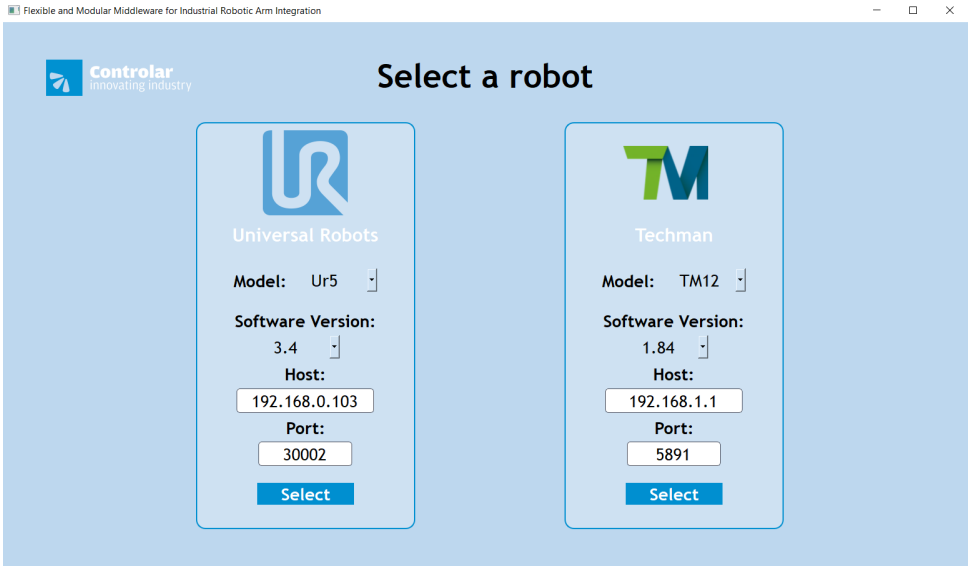
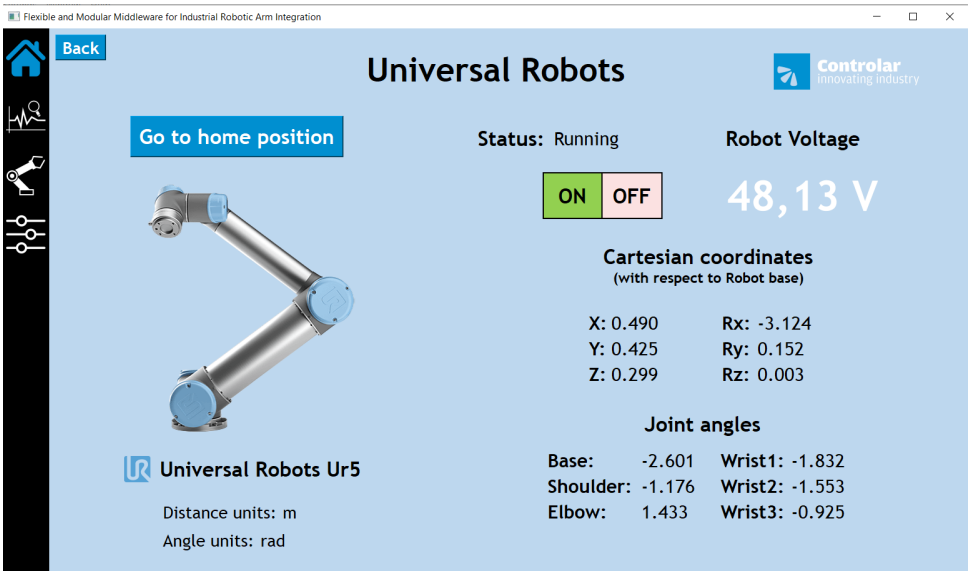
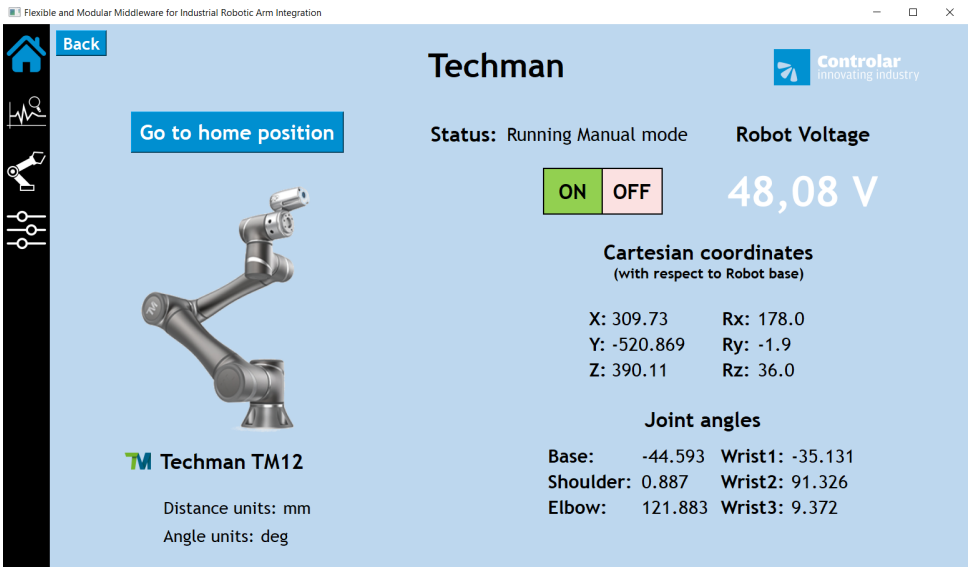


Figure 4.11: GUI landing page

Once the robot is selected, the middleware layer interprets the parameters introduced and establishes the connection with it. Afterwards, as shown in Fig. 4.12, the page displayed in the interface is the main page, also called the home page. This page displays the most relevant data about the robot's status. On the left side of the page is an image of the robot along with its brand and model. Underneath, since it is essential to notice, the default system units are displayed. On the right side, the current status and position of the robot can be seen. The status is described in detail by words but also with the help of an ON-OFF signal. Even if the robot is switched on, for example, in idle mode, the signal only shows ON if the robot is ready to receive external commands. The Cartesian coordinates of the robot's current position and the angle of its six joints are displayed beneath the robot voltage value. These values are expressed according to the default unit configuration of the robot. In addition, on the left-hand side of each page is the vertical menu which allows the user to navigate between pages. The icons represent the home page, the robot's status, motion control and settings, respectively.



(a) Universal Robots Ur5 home page



(b) Techman TM12 home page

Figure 4.12: GUI home page

If the user presses the icon after the home page on the menu, the GUI will redirect to the robot status page, as shown in Fig. 4.13. It complements the home page as it also presents aspects related to the robot's status but with greater detail. Besides the cartesian coordinates and angles of the robot's joints already shown on the previous page, it presents data about the tool center point offset and the project status. On the right side of the page, it is possible to see if the emergency button is pressed, if the project is running, the robot's voltage and current and the motion speed parameters. Moreover, the workbase values set to zero demonstrate that the current frame is the base coordinate system.

If the user presses the icon of the motion control page, the page shown in Fig. 4.14 is displayed.

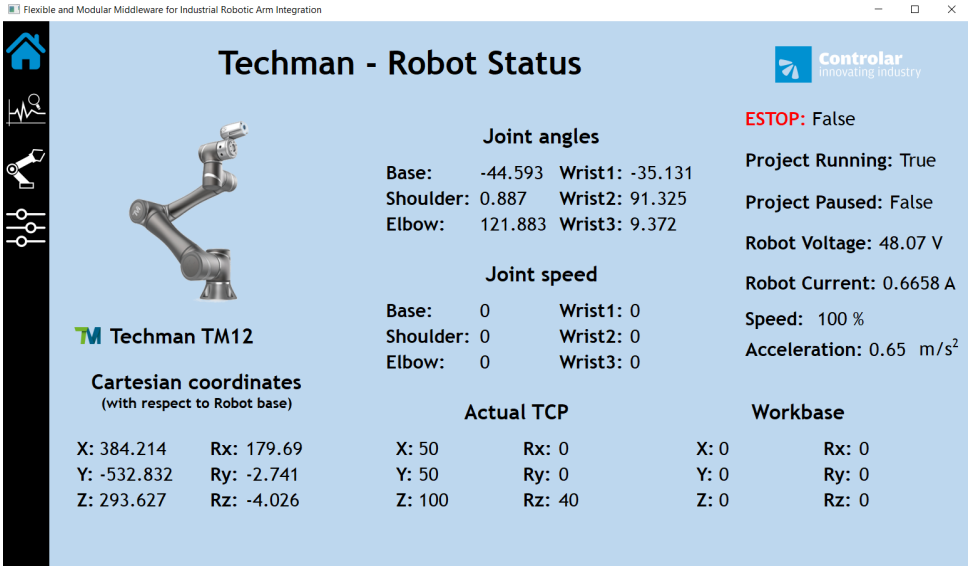


Figure 4.13: GUI Robot status page

The motion control page is the GUI's core because it is where the user can move the robot using the middleware layer functions.

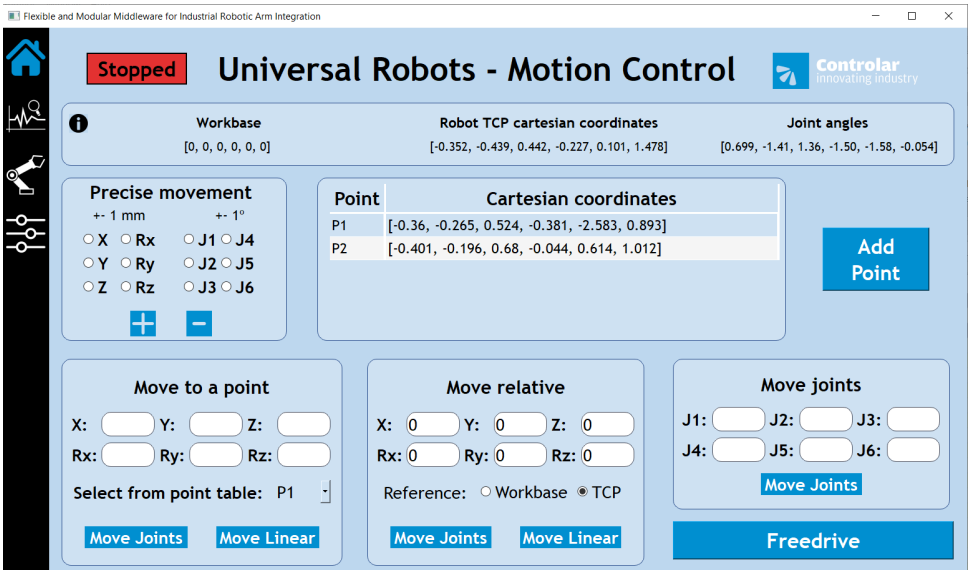


Figure 4.14: GUI Motion control page

At the top of the page is a sign that indicates whether the robot is moving or standing still. It is only allowed to send a motion command if the signal indicates that the robot is stopped, given that it must reach its target before starting the next movement. Below the signal, the robot's current position is displayed in cartesian coordinates and joint angles. In addition, in the middle of the page, the user can find a section for performing precise movements and also a point table.

To perform precise movements, it is only necessary to select the direction or joint and press the positive or negative increment button. As subtitled in the section, the increment is 1 millimetre or 1 degree. The point table is a unique feature and allows the user to store the robot's current position using the "add_point" button. The saved points can then be used as motion targets. At the bottom of the page are the available options for moving the robot. At the left section, the robot moves to a point inserted by the user. The user can fill in each cartesian coordinate or else pick a target from the point list. After picking the target, there is still the option to execute joint or linear movements. The middle section is devoted to relative movements, in this case, relative to the tool center point. The user can insert the relative target into the blank spaces and then confirm the motion. The right-side section is similar to "Move to a point", but the target is expressed as the angle of six joints instead of a Cartesian coordinate. Finally, this page also has a button to activate the freedrive mode remotely.

The last page is dedicated to the robot settings. Moreover, as shown in Fig. 4.15, it also has a section with informative details about the robot on the left side of the page. Concerning the settings, it is possible to configure the offset of the tool center point, redefine the home position, vary the motion speed and acceleration and, lastly, select which systems units the GUI should express the values in. There is also an additional freedrive button on this page.

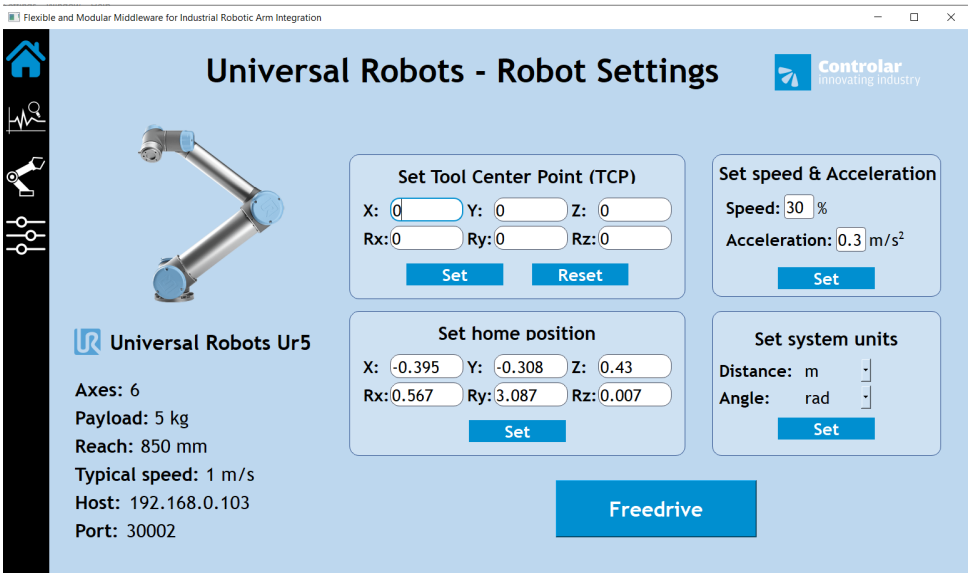


Figure 4.15: GUI Robot Settings page

4.6 Summary

Python was the programming language of choice to develop both the GUI and the middleware layer based on the architecture summarised in Fig. 3.1 of Section 3.1 according to the flexibility and modularity requirements. Moreover, the necessary setup procedure for the robots chosen for case study was described. Whilst the UR5 merely requires the controller to be switched on to be programmed, the TM12 needs some additional configurations. Therefore, the TM12 must run a project on the TMFLow software, which enables the Listen node and only then will it be ready to receive commands from external devices.

The middleware layer development can be divided into three main modules: Data Parser, Communication module and Robot Control. The whole API was designed to emphasise the solution's modularity and to build a solid foundation that allows easy integration of new robots. Furthermore, to communicate with the TM12 robot, a python communication driver named *techmanpy* was used.

The Data Parser module, the middleware layer's lowest-level module, aims to receive the streamed data broadcasted by the manipulator's controller and map it appropriately. Since each manufacturer is free to define how the data is sent, the packages have to be parsed according to the robot connected. The whole API uses a standard layout containing the most interesting aspects of the handler's state to map the received data. Although the packages sent by the two robots are somewhat different, this module handles them, and all information is summarised in the generic layout.

The Communication Module is in charge of establishing the connection with the robot and holds aspects related to TCP/IP socket configuration. It also correctly shapes and sends Universal Robot's commands, whereas the *techmanpy* driver mainly handles the communication with the Techman robot.

Lastly, the Robot Control module is the core of the middleware and is responsible for high-level control according to the user's actions. Given the disparities between the two robots, aspects such as distance/angular units or even the speed/acceleration of the robot are standardised in this module, which ensures the flexibility of the API. Moreover, after identifying the robot to be programmed, the user may adjust the API settings to personal preference and use the several movement commands available. For each type of movement, the API correctly formats the command to send to the robot controller.

The Graphical User interface was built using the PyQt5 python framework to guarantee high flexibility and compatibility. Firstly, the front-end component of the GUI was developed in the QtDesigner interface using the available widgets. The back-end component was developed from hand-coding using the front-end elements.

Three threads comprise the GUI operation. The main thread is in charge of GUI responsiveness, a second thread constantly receives and updates the robot's state, and the third thread is dedicated to executing the Robot Control module functions.

Chapter 5

Experiments and Results

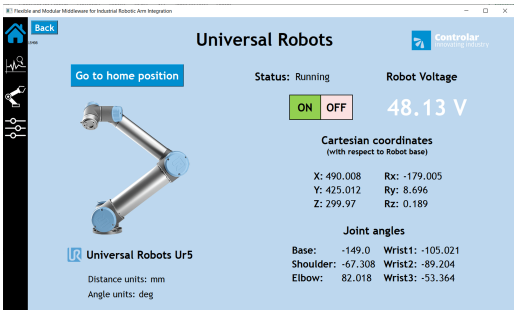
The main objective of this Chapter is to delineate the experiments performed to test the developed middleware layer and validate it. Section 5.1 presents the experiments performed regarding the Data Parser module, that is, the experiments intend to ensure that the system receives and interprets the information sent by the robot controller. Then, Section 5.2 discusses the experiments performed to evaluate the robot's trajectory for each type of motion. Finally, Section 5.3 describes the usability testing method used to evaluate the GUI design and functionality.

5.1 Data Parsing and Communication

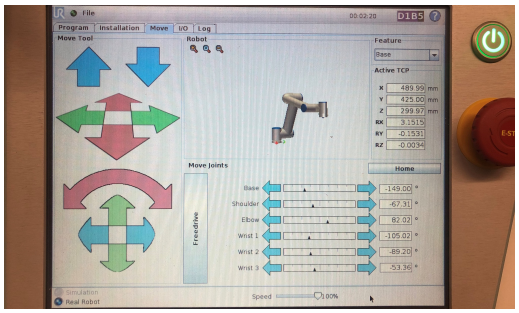
The middleware layer, together with the GUI, has two main objectives. Firstly, it must be apt to receive and parse the robot's state through the broadcasted byte stream into the TCP/IP socket. On the other hand, it must be able to send correctly formatted commands to move the robot. Since several control functions depend directly on the robot's current state and position, the data must be correctly mapped.

Hence, the first experiment aimed to confirm the reliability of the data received. To do so, the data displayed in the GUI was compared with what was displayed in the proprietary software of each robot. In Fig. 5.1, is it possible to compare the data displayed in the GUI with what is sent by the PolyScope software of Universal Robots UR5. Although the default units of the Universal robots system are meters for distance measurements and radians for angular measurements, the teach pendant interface does not follow this format. As can be observed from Fig. 5.1b, distance units are expressed in millimeters and the orientation of the tool head [rx, ry, rz] in radians. The joint angles, on the other hand, are displayed in degrees. For this reason, millimeters and degrees were the selected representation in GUI. Thus, Fig. 5.1 proves that the data presented in the GUI is consistent with what the robot software displays.

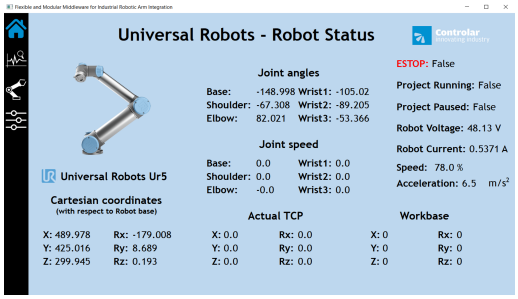
Furthermore, Fig. 5.2 also proves the compatibility of the data presented in the GUI when compared with the TMFlow software from Techman. Besides the electrical controller details, the cartesian coordinates and the joint angles also match the real value.



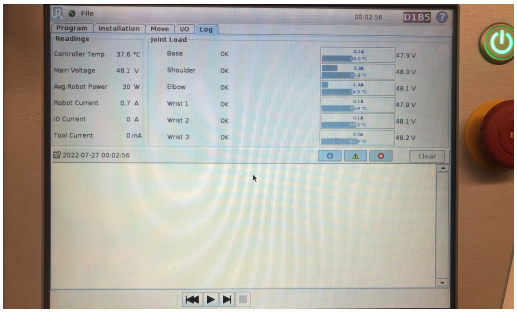
(a) UR5 GUI home page



(b) PolyScope home page

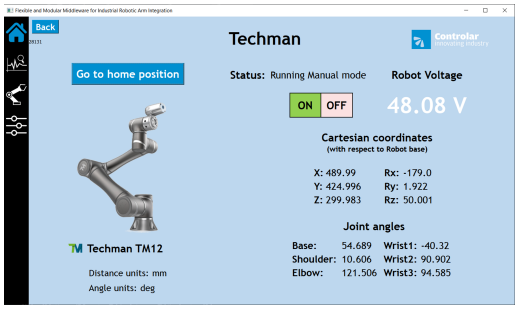


(c) UR5 GUI Robot Status page

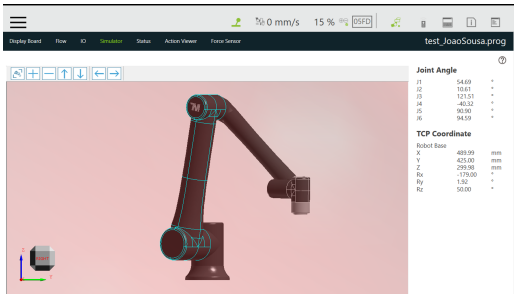


(d) PolyScope log page

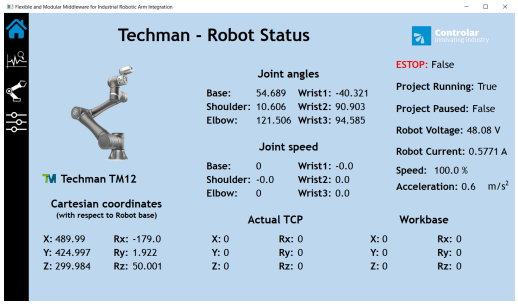
Figure 5.1: UR5 status comparison between the GUI and PolyScope



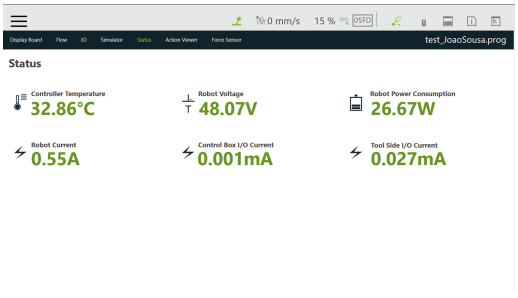
(a) TM12 GUI home page



(b) TMFlow Simulator page



(c) TM12 GUI Robot Status page



(d) TMFlow status page

Figure 5.2: TM12 status comparison between the GUI and TMFlow

Another essential aspect to examine is the rate at which information is updated on the GUI. For this purpose, measuring the time between two consecutive updates is required. Likewise, for a

sample of 150 messages received on the TCP/IP socket, the time between two consecutive packets was measured, and then the average period was calculated. This process was repeated ten times for each robot and obtained the results presented in Table 5.1. It was performed on a laptop with an Intel(R) Core(TM) i7-7500U CPU @ 2.70 GHz and 8 GB of RAM connected to the robot through an Ethernet cable. The results are expressed as frequency values, computed by the inverse of the average time. They may however vary according to the hardware used.

Table 5.1: GUI data update frequency

Status update frequency (Hz)		
Trial number	**Universal Robots**	**Techman**
1	3.319 Hz	9.361 Hz
2	3.319 Hz	9.198 Hz
3	3.320 Hz	9.145 Hz
4	3.312 Hz	9.145 Hz
5	3.313 Hz	9.252 Hz
6	3.319 Hz	9.198 Hz
7	3.312 Hz	9.146 Hz
8	3.313 Hz	9.197 Hz
9	3.320 Hz	9.145 Hz
10	3.319 Hz	9.146 Hz
Average	3.316 Hz	9.193 Hz

It can be concluded that the GUI update frequency varies depending on the robot that has been selected. The main reason behind this is the difference in the frequency with which the robot controller broadcasts the byte stream. For Universal Robots, the announced frequency of the secondary client interface is 10 Hz, whereas for Techman, the TM Ethernet Slave announces a frequency of 100 Hz. However, none of these values is guaranteed by the manufacturers. In any case, there is a drop in the robot's status update frequency, but it is impossible to fully assess its cause since the frequency with which the robot sends data is unknown. Nevertheless, the middleware layer is expected to impact the frequency values because the packages must be received, parsed, mapped and finally updated in the GUI.

Finally, there was no sign of packet loss throughout the experiments, and all the received messages contained the entire data as described in the documentation.

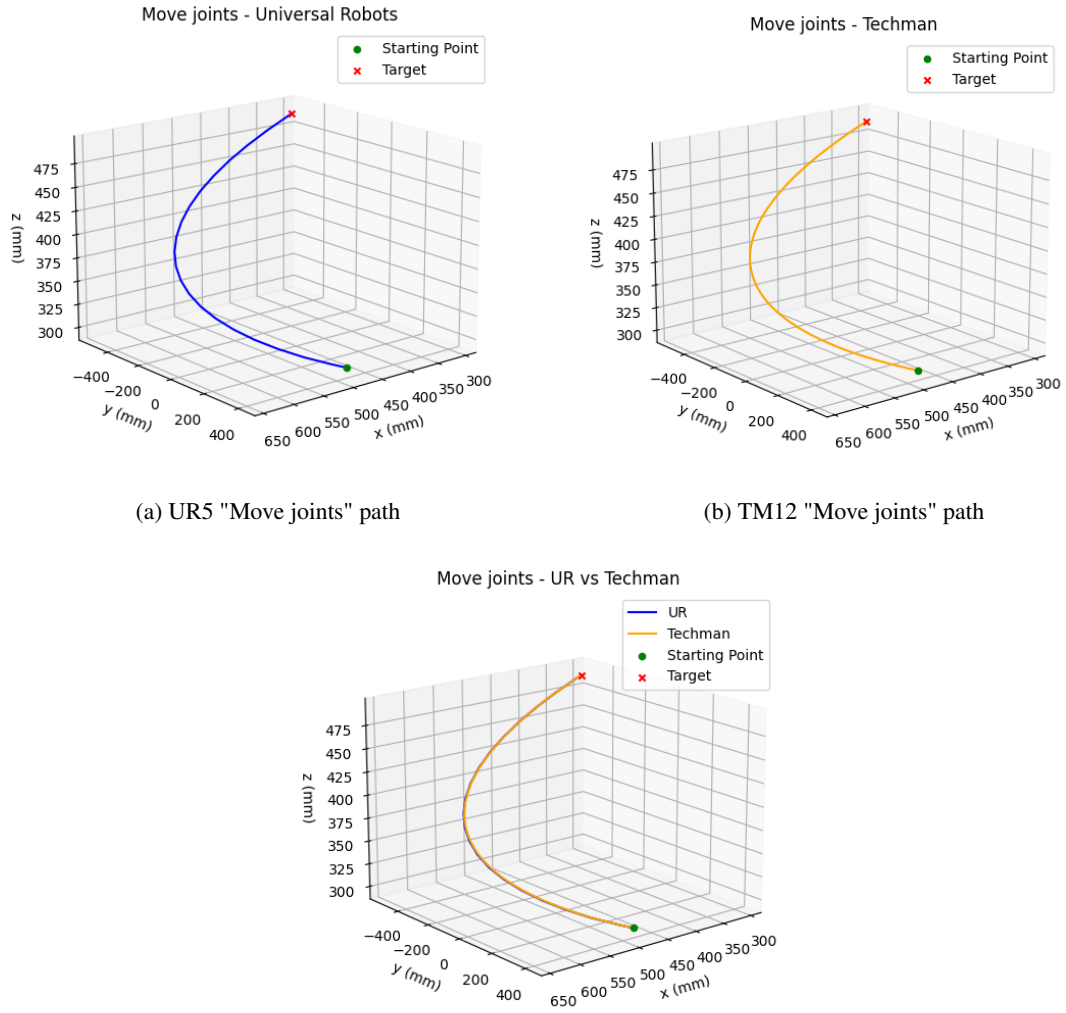
5.2 Robot Motion

The middleware layer has several functionalities that allow programming and moving the robot. Moreover, supported by the GUI, the user can define the target point and observe the robot's motion. For this purpose, several experiments were performed to analyse the robot's behaviour after receiving motion-related commands.

To study the motion path and evaluate if the action sent by the user was correctly executed, the cartesian coordinates [x, y, z] of the robot's end-effector were registered and later used to depict

the movement graphically. Once the command is sent to the robot, the data parser module is asked to periodically store the robot's current position. An arbitrary speed and acceleration were chosen to perform the experiments since it does not affect the robot's path.

The first task involved moving the robot's joints to a target, that is, performing a point-to-point motion. This function is one of the key components of the robot's movement. Therefore, for the first experiment, the robot starts at the point [490, 425, 300] and moves to the target point [305, -520, 490]. Fig. 5.3 illustrates the path the UR5 (Fig. 5.3a) and TM12 (Fig. 5.3b) take from the starting point to the target point. Moreover, the blue line expresses the UR5 path, whereas the orange line represents the TM12 path.



(a) UR5 "Move joints" path

(b) TM12 "Move joints" path

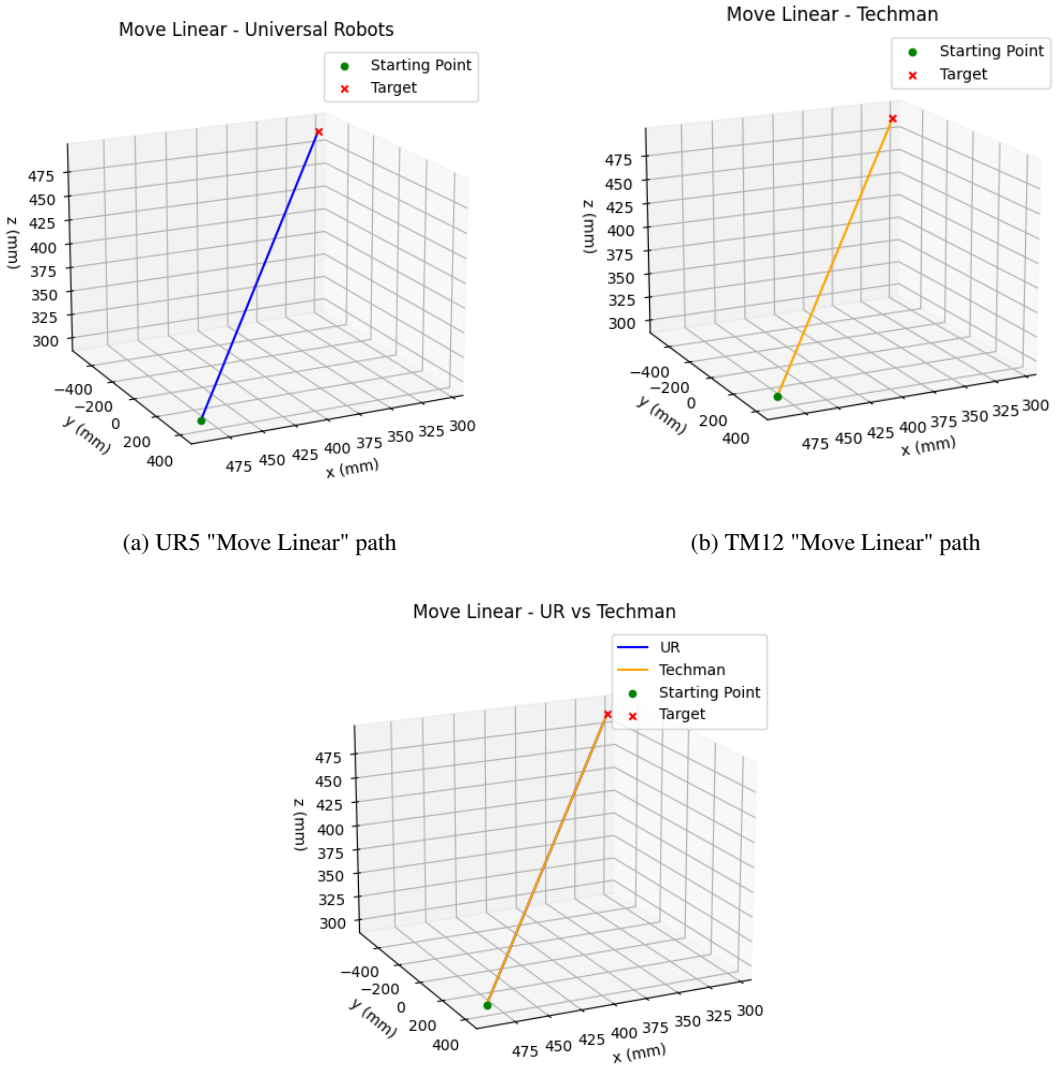
(c) UR5 vs TM12 paths

Figure 5.3: Robot's path executing a "Move joints" function

As mentioned before, in this type of movement, the controller estimates the joint angles corresponding to the target point and then executes the motion, resulting in a curved path for the tool. Furthermore, Fig. 5.3c demonstrates the overlapping paths of both robots, showing that, although

they are distinct robots with unequal ranges, they follow the same path from the origin to the end.

The following experiment performed was similar to the previous "move joints" one, with the same start and destination point but a distinct motion type. This time, as can be seen in Fig. 5.4, the function executed was "move linear", causing the robot to travel along a straight path. Once again, Figures 5.4a and 5.4b show the UR5 and TM12 paths, respectively.



(a) UR5 "Move Linear" path

(b) TM12 "Move Linear" path

(c) UR5 vs TM12 paths

Figure 5.4: Robot's path executing a "Move Linear" function

Fig. 5.4c shows the two overlapping paths, concluding again that the robots follow the same path. However, it was already expected in this case since the rectilinear path between two points is unique.

Furthermore, in order to better understand the differences between these two types of movement, Fig. 5.5 shows on the same graph the two trajectories travelled by the TM12 robot between

the initial and final points previously defined.

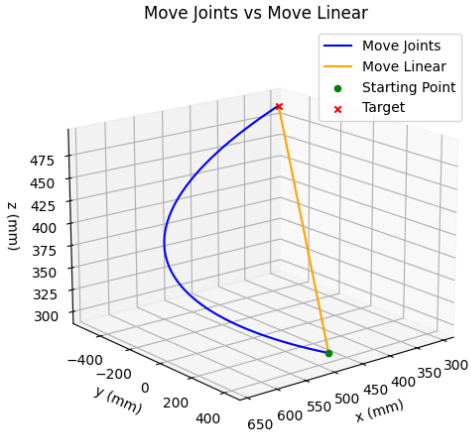
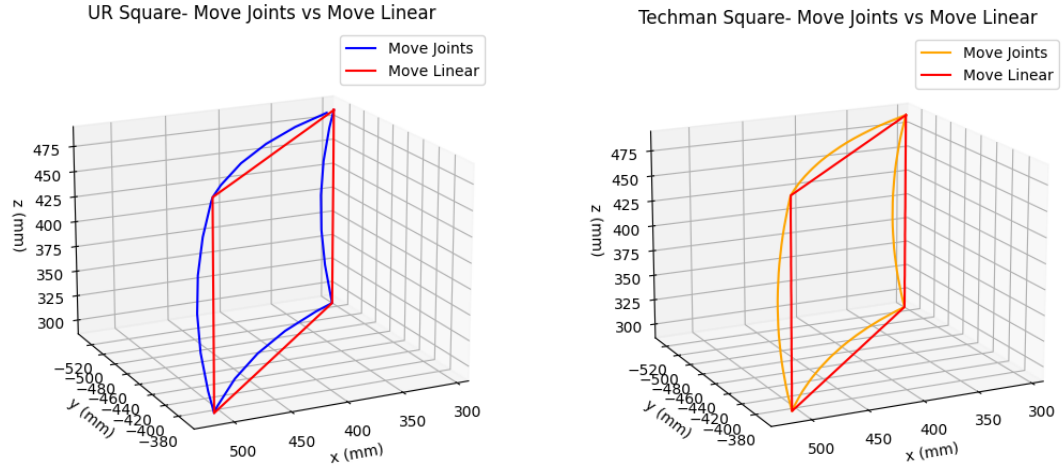


Figure 5.5: Move Joints vs Move linear for TM12 robot

It is also shown in Figures 5.6a and 5.6b the trajectories that the UR5 and TM12 robots travelled when the introduced points form a square. Through those two figures, the difference between the two motions is perfectly perceptible. While the function "Move linear" forms a 3D plane, the function "Move joints" makes the robot's path between points always curvy.



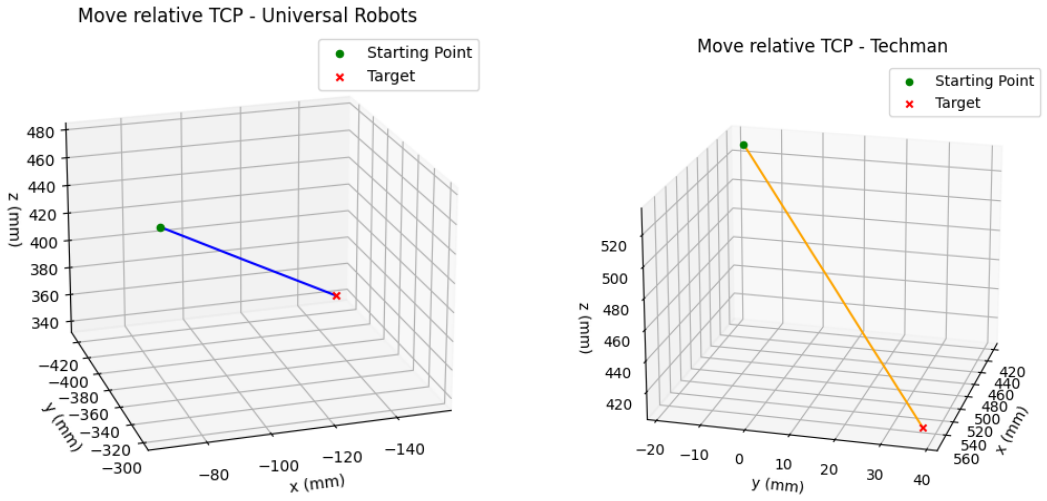
(a) UR5 square paths

(b) TM12 square paths

Figure 5.6: Difference between Move Joints and Move linear when performing a square path

After completing the point-to-point movement studies, the next step was to conduct experiments on a different type of movement. Thus, it was then evaluated the function that allows moving the robot relative to the tool center point frame. In this case, in addition to the cartesian coordinates [x, y, z] of the initial point, the values [rx, ry, rz] are also needed since they indicate the robot's tool head orientation. Furthermore, as described in Section 4.4, the input target value

is a relative measure, unlike the global target used in the motion types mentioned above. For both robots in this experiment, the goal was to move the robot 200 mm along the z-axis, which means the inserted target value was [0, 0, 0, 200, 0, 0, 0]. Opposite to what happened in the previous experiments, the starting position of the two robots is not the same. For UR5 the starting point is [-68, -301, 475, 24, -138, 45] and for TM12 is [420, -18, 528, 126, -1,8, 109], both expressed in millimeters and degrees. Fig. 5.7 shows the path of UR5 (Fig. 5.7a) and TM12 (Fig. 5.7b) when performing the motion previously detailed.



(a) UR5 motion relative to the tool center point

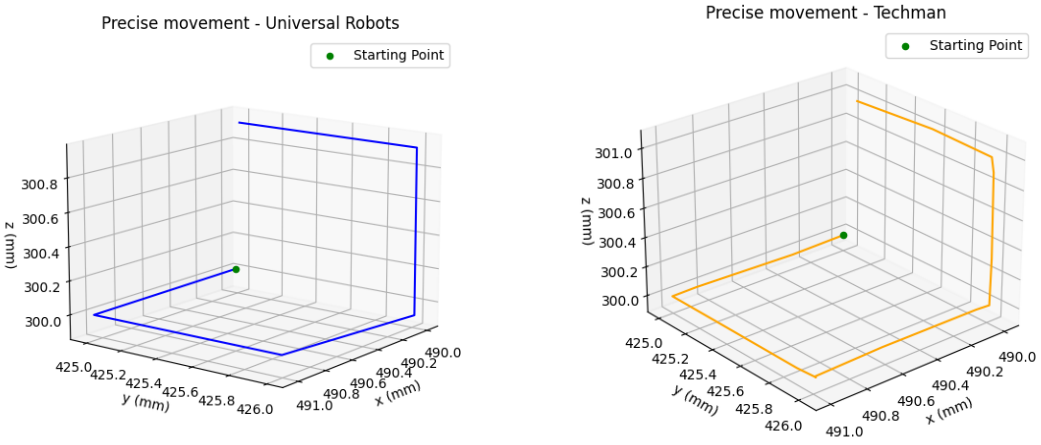
(b) TM12 motion relative to the tool center point

Figure 5.7: Tool Center Point relative motion example

The robot path was a rectilinear motion, decreasing the z-axis value as expected since the tool frame has the z-axis in the opposite direction of the base coordinate system. The goal points expressed in the global frame were calculated for both robots and matched with the real value. Due to the discrepancy between the two robots' representations used for tool orientation, the endpoint computation is different for each. The endpoint of the UR with axis-angle representation was [-151.4, -424.2, 341.4, 24, -138, 45]. On the other hand, the endpoint was [571.8, 38.2, 410.5, 126, -1.8, 109] for the TM with Euler angles representation. It is worth highlighting that the parameters of the robot's orientation [rx, ry, rz] were not changed since the motion was only performed on the z-axis.

The precise movement feature was also one of the experiments performed. This feature allows the user to perform fine movements to adjust the robot's position as accurately as possible. Each time the function of the precise movement is called, the robot moves 1 millimetre or 1 degree in the selected direction. Therefore, the following sequence was created to test the robot's accuracy: x+, y+, x-, z+, y-. Fig. 5.8 shows the path taken by the UR5 (Fig. 5.8a) and TM12 (Fig. 5.8b) robots. Both robots executed the sequence correctly with high precision, following the sequence and moving millimetre by millimetre, as intended.

With all motion types studied, the next step was to perform experiments evaluating the robot's

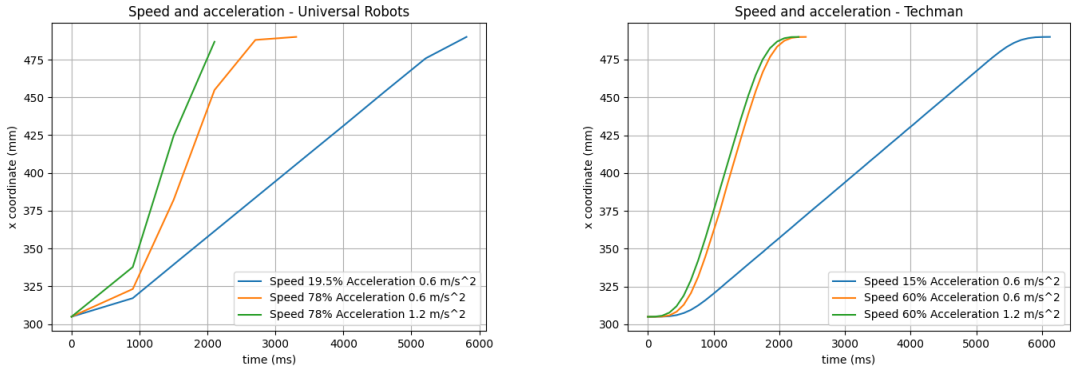


(a) UR5 Precise movement sequence path

(b) TM12 Precise movement sequence path

Figure 5.8: Path for the precise movement sequence

behaviour in different scenarios defined in the settings. As such, the robot's movement was first examined under different values of speed and acceleration. To do so, the variation of the cartesian coordinates was measured over time while the robot performed a point-to-point motion. For both robots, the starting point was [305, -420, 490], and the target was [490, 425, 300]. Fig. 5.9 shows the UR5 (Fig. 5.9a) and TM12 (Fig. 5.9b) x-coordinate variation over time for three different combinations of speed and acceleration.



(a) UR5 x-coordinate variation over time

(b) TM12 x-coordinate variation over time

Figure 5.9: Speed and acceleration influence in robot's motion

It is enough to examine one cartesian coordinate to study the effect of velocity and acceleration on the robot's motion. Even though the percentage value of speed differs from the UR to TM, the three combinations of speed and acceleration are common to both robots. Since the speed is a percentage of the robot's max speed, 15% on TM12 is the same speed value as 19.5% on UR because the TM has a max speed of 1.3 m/s while the UR has 1 m/s. The same happens between the 60% for TM and 78% for UR.

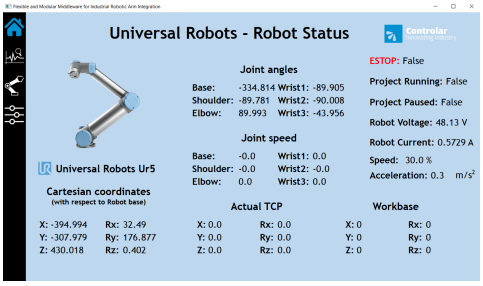
Therefore, the first configuration (blue line) has low and arbitrary speed and acceleration values. The second configuration (orange line) increases the speed value, maintaining the acceleration. Lastly, the third configuration (green line) increases the acceleration values, maintaining the previous speed.

The first aspect to observe from the graphics is that the speed and acceleration settings are really the same between the robots because the timestamp at which each scenario ends is similar. In other words, for each speed/acceleration configuration, both robots reach the target simultaneously.

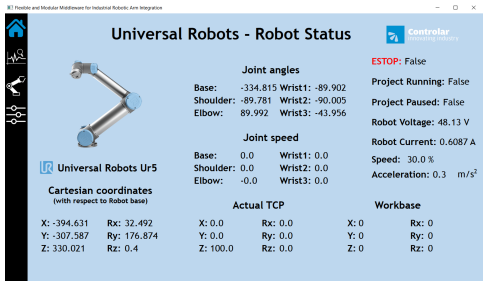
Moreover, another aspect worth mentioning is the effect of a slower data update rate from the UR robot. For lower speeds, the update rate difference is insignificant because the GUI has time to store several positions throughout the movement. However, for higher speeds, the motion graph displays a sharper curve indicating the lack of data to form a smooth curve.

Finally, the effect of the settings on the robot's motion is quite clear. From the first to the second scenario, the robot travels the distance to the target much faster since the robot's speed has been changed. From the second to the third scenario, it is possible to study the minor influence of the acceleration. The acceleration will only change the initial and final moment of the motion, which is when the robot accelerates to the established speed and brakes when it gets close to the target. Between those two moments, the robot moves with uniform velocity, resulting in null acceleration. Either way, the small influence is enough to reduce the total travel time.

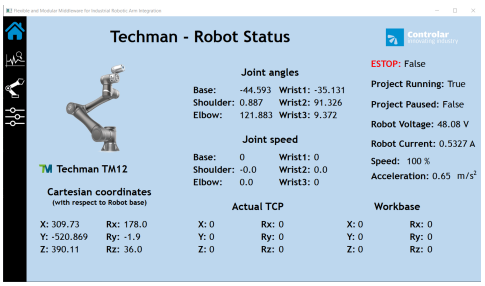
Another aspect that both the middleware layer and the GUI allow to be configured is the tool center point offset. Fig. 5.10 shows the changes in the cartesian coordinates of the robot's end-effector according to the tool center point configuration used.



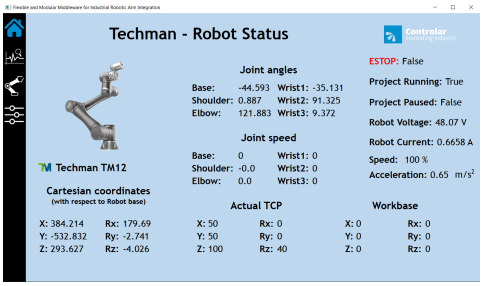
(a) UR5 with no Tool Center Point offset



(b) UR5 with Tool Center Point offset configured



(c) TM12 with no Tool Center Point offset



(d) TM12 with Tool Center Point offset configured

Figure 5.10: Tool Center Point configuration

From Fig. 5.10a to 5.10b, only the value of the tool centre point was changed to 100 millimeters along the z-axis, meaning that the offset value was then [0, 0, 100, 0, 0, 0]. It also proves that the robot's current position did not change because the joint angles remain precisely the same. On the other hand, the cartesian coordinates value shifted according to the offset entered. Figures 5.10c and 5.10d illustrate the same behaviour for the TM12 robot.

5.3 GUI Usability Testing

Finally, in order to verify that all our functions work properly and that the key features are easily accessible, and ensure an easy utilization for all users, usability testing was carried out using the developed Graphical User Interface. Ultimately, these tests allowed to identify weaknesses, observe the participants' interaction with the interface, and later obtain feedback from the users for improvements.

In short, it is an evaluation approach that involves measuring user performance on typical tasks. The facilitator is in charge of guiding the participants during the task performance, providing the same test conditions for each one of them. It is also the facilitator's responsibility to define the set of tasks based on frequent and critical actions. An action is considered critical if, in some way, it affects other tasks. Furthermore, user satisfaction questionnaires are used to evaluate the interface. The questionnaires focus on how well users perform tasks with the GUI. It helps capture the participants' overall perception of the system's usability and specific perception regarding usability concerns. Finally, analysing the results through charts and graphs helps identify the GUI issues and highlight the design's positive aspects.

Therefore, usability testing was carried out with thirteen participants with different backgrounds. Fig. 5.11 summarizes some of their relevant characteristics.

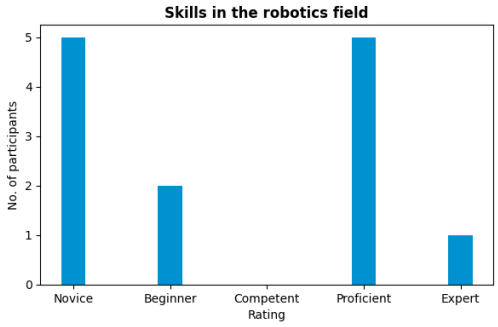
According to Fig. 5.11a, we conclude that the participants have different levels of expertise in the robotics field. It is advantageous to have such distribution as it ensures diversified GUI feedback, given our objective of providing a simple interface for non-expert and expert users. It also suggests that half of the participants are not familiar with robotics' theoretical aspects. Furthermore, Fig. 5.11b shows that most participants do not program robotic manipulators. This is particularly relevant since the GUI is intended to simplify robot programming for inexperienced users. Lastly, the participant breakdown between FEUP and Controlar is presented in Fig. 5.11c.

Initially, the participants had their first experience with the GUI, navigating between pages to understand their content and learn about the available functionalities. Afterwards, every participant answered two questions addressed to each of the interface's pages. The questions were as follows:

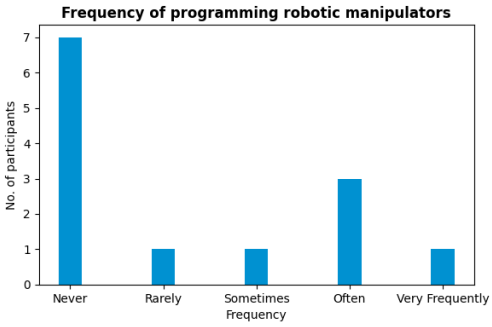
1. Did you find the layout easy to use?

2. Did you understand every functionality available on this page?

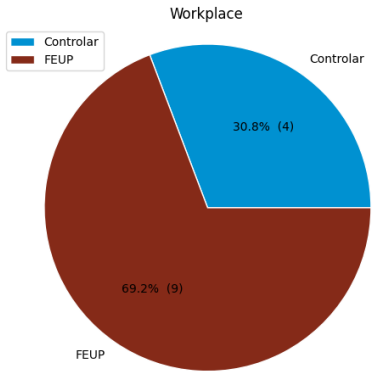
The answer to both questions was yes for all pages, except the Robot Motion page. This result was expected since it is the most complex page of the GUI and contains several motion types



(a) Participants' skills in the robotic field



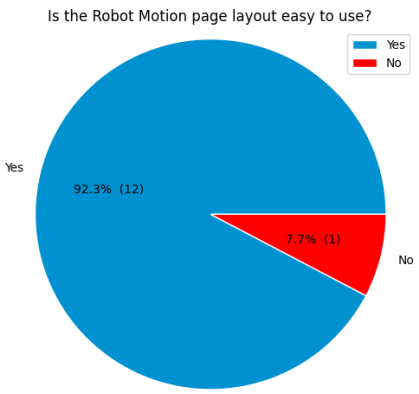
(b) Frequency with which the participants usually program robots



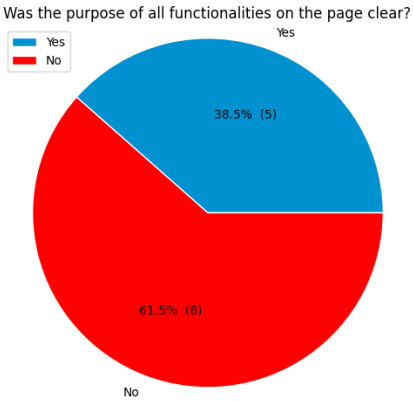
(c) Breakdown of participants by workplace

Figure 5.11: Participants background

available to execute. Participants with no theoretical background in robotics also face additional difficulty in understanding the whole page content at first glance. The statistical details regarding the answers to the two questions for this page are presented in Fig. 5.12.



(a) Results of the first question



(b) Results of the second question

Figure 5.12: Results of the answers for the Motion Control Page

Participants who answered no to the second question stated that the section of the precise movement was not clear enough. For this reason, they did not understand its purpose and suggested minor changes. At the time that the usability testing was performed, the label identifying the increment ($\pm 1mm$ or $\pm 1°$) did not exist and, consequently, adding it was one of the suggestions for improving the GUI. Moreover, it was also mentioned that it was not very intuitive to choose the target point from the table.

After the initial questions, participants performed a set of tasks presented in Table 5.2. These tasks comprised aspects of the whole GUI functioning, such as interpreting the robot's state and sending motion commands. The time it took participants to complete the tasks was also documented.

Table 5.2: GUI Usability Testing tasks performed by the participants

GUI Usability Testing Tasks	
1	Select the UR5 robot and access it through host 192.168.0.103 and port 30002
2	Understand the current robot's state
3	Indicate the current cartesian coordinates and default system units
4	Read the Tool Center Point offset
5	Set the Tool Center Point offset to 100 mm away from the robot on the z-axis
6	Set the speed and acceleration to safe values (15% and 0.15 m/s^2)
7	Store the actual robot's position on the point table
8	Move the robot to the cartesian coordinates [-0.4, -0.3, 0.3, 0.5, 3.1, 0]
9	Move to the previous position with a linear motion
10	Move 100 mm along the z-axis relative to the Tool Center Point
11	Move the robot up by 2 mm

Once the tasks were completed, the participants rated each task on its difficulty. Fig. 5.13 summarises the results in a graph.

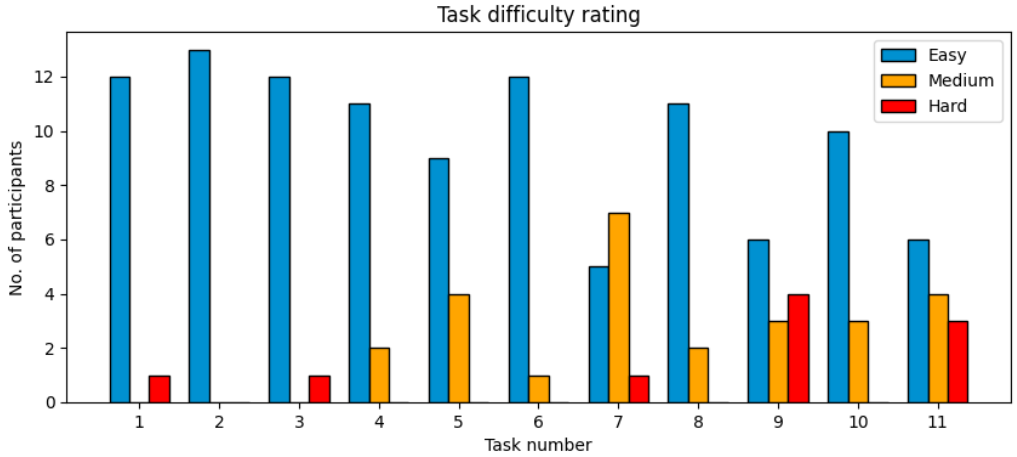


Figure 5.13: Difficulty rating for each task

From the graph analysis, we can conclude that the vast majority of users could overcome the proposed tasks with great ease. On the other hand, tasks 7, 9 and 11, related to the point table and precise movement, are more often classified with greater difficulty. This agrees with what was previously mentioned about this page, leading to the conclusion that these are the two less intuitive functionalities. Regarding the point table and the selection of its points as a target, the results suggest that some changes would contribute to a more user-friendly interface. An alternative would be to move to the target point by double-clicking it.

As for the time taken to complete the tasks, only two participants exceeded the expected value (Fig. 5.14). Therefore, it is another index that proves that, in general, the GUI is intuitive.

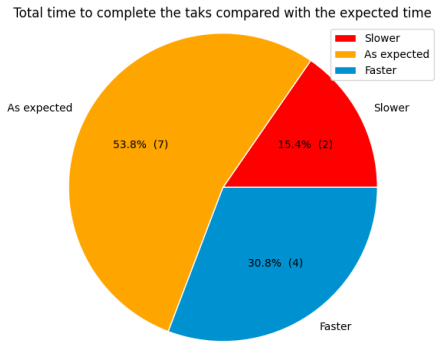
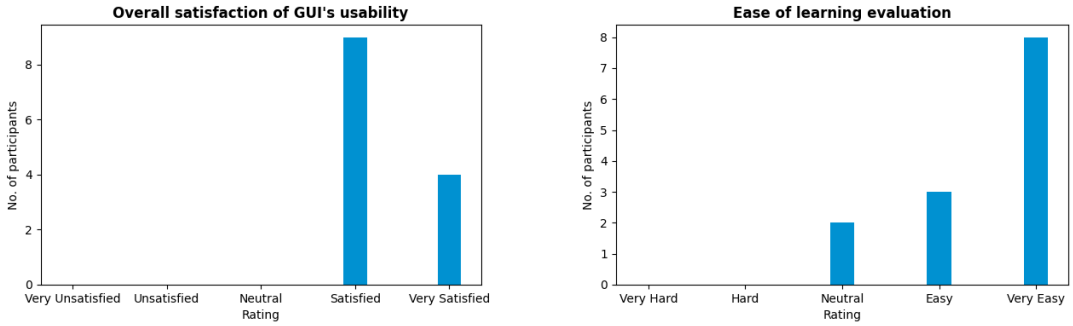


Figure 5.14: Comparison of total and expected time for completing the tasks

Lastly, the participants were asked two more questions about overall aspects. Thus, they were asked to rate the overall satisfaction of the GUI usability (Fig. 5.15a) and the ease of adaptation to the interface (Fig. 5.15b). As it is possible to deduce from Fig. 5.15, the results are very positive and suggest that the interface's purpose was successfully accomplished.



(a) Overall satisfaction rating of the GUI's usability

(b) GUI learning curve rating

Figure 5.15: Overall GUI evaluation

Since the participants found the GUI learning curve relatively smooth, it is predictable that if they repeated the tasks, they would not encounter any difficulty and would finish them faster. All participants answered positively when asked if they would use the GUI to program the robot.

5.4 Summary

Throughout this chapter, the performance of the middleware layer and the GUI was evaluated and validated. As such, each experiment's outcome is described in detail, which allowed us to prove the performance of the work developed.

First of all, the reliability of the data received was assessed by matching what was displayed in the GUI with what was displayed in the proprietary software of each robot. Furthermore, the update rate on the GUI was also measured. The results expressed as frequency values prove that the update frequency varies depending on the selected robot, as each robot controller has a different package broadcast frequency.

Afterwards, the experiments performed to evaluate the robot's behaviour after receiving motion-related commands were described. The cartesian coordinates [x, y, z] of the robot's end-effector were stored and plotted on a graph to study the motion path of each robot. This set of tests allowed us to compare the different types of motion, such as point-to-point, linear and relative to the tool center point frame, as well as other features implemented. Moreover, it was concluded that although both robots have unequal ranges, they follow the same path if the starting and target points are the same. To study the effect of the speed and acceleration values, only the x coordinate was examined over time.

Lastly, a usability testing method was carried out with 13 different users with distinct robotics backgrounds to validate the GUI's simplicity. Participants began by exploring and rating each page individually. Afterwards, they performed a set of tasks that briefly addressed all functionalities and evaluated the GUI utility. Its development was successful since all participants gave positive feedback.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In conformity with the new industrial revolution, factories integrate advanced and heterogeneous industrial robots in their assembly lines. However, that is one of the biggest challenges in the robotics field. Given the wide variety of robotic handlers, there is an ongoing concern about developing methods that provide seamless integration in the automation processes.

To address this challenge, the main objective of the thesis is to study and develop a middleware layer capable of integrating several robotic manipulators. It is meant to be flexible and modular, allowing the user to cope with various functionalities suitable to a wide variety of robots. Scalability is another fundamental property of the hardware-agnostic middleware, essential to answer the ever-changing manufacturing requirements. Additionally, with both inexperienced and experienced users in mind, we developed an intuitive and user-friendly GUI integrated with the middleware content.

As an introductory stage to this thesis, a study on robotics middleware solutions was carried out. Due to the wide range of available solutions and the constant release of new ones, only the most relevant in the scope of the thesis were studied in detail. Therefore, ROS and OPC UA were two distinct architectures highlighted as potential foundations for achieving the thesis objectives.

However, the proposed solution is not ROS or OPC UA based. Even though almost every manufacturer has ROS packages that support robot communication and control, they are manufacturer-dependent and therefore incompatible. It would then be required to develop a platform that allowed the integration of the robots alongside their ROS packages. Since ROS currently only runs on Unix-based platforms and the packages' disparities would compromise the system's modularity and flexibility, this approach was dropped following a discussion with the company. The same happened to the OPC UA approach. Since the OPC UA architecture and robotic manipulators are still new to each other, communication with its controllers is rather complex. Some manufacturers are providing OPC UA licenses that simplify communication, although they are costly. As such, the communication problem would have to be solved from scratch. This approach would be too

time-consuming and would hinder the possibility of achieving all the defined objectives that go beyond communication.

The proposed solution was designed based on the middleware solutions studied. Despite not having been developed using the studied frameworks, they were nevertheless helpful in properly structuring the solution. The middleware layer is based on TCP/IP connections between the PC and the robot controller. Given the flexibility and generality requirements of the solution, a modular architecture based on three major modules was defined: Robot Control module, Communication module and Data parser module. The Robot Control module is in charge of high-level actions such as adjusting the robot settings and sending motion commands. Moreover, the Communication module contains everything needed for establishing the connection with the robots. Lastly, the Data Parser reads and maps the information received into a standard layout. With this framework and its implementation, the integration of new robots is straightforward. Using the middleware layer, any robot from the brands used as a case study can be directly programmed. Nevertheless, for other brands, it is intuitive to extend the middleware content to support them. It is as simple as complementing the existing functionalities. Therefore, high levels of flexibility and scalability were achieved.

In addition to the middleware layer, a GUI was developed to provide a better user experience. It is user-friendly and mainly dedicated to less experienced users or operators with reduced programming skills so they can still take advantage of the middleware functionalities. The interface layer contains a valuable tool for communicating with the robot as it uses the API functions and avoids the coding task.

In order to validate the middleware solution, a case study was defined with two different industrial robots: Universal Robots UR5 and the Techman TM12. They are collaborative robots from different brands and differ in several aspects. Firstly, TM12 has a higher range, better accuracy and more power. Moreover, the system units and the representation of the robot's orientation are specific aspects that also differ between the UR5 and the TM12. The middleware layer encapsulates these differences and provides a set of standardized functions. In addition, due to the flexible and modular architecture design, other robots from UR and Techman can be programmed using the middleware layer without requiring modifications. It is also important to highlight the several controller software versions supported by the middleware. These versions change the robot's internal parameters but the differences are handled by the API. For new robot manufacturers, it is necessary to complement the API, but the developed framework is ready to ease the integration. Therefore, it is safe to answer positively to the first research question: Is it possible, using the same middleware layer, to interact and program industrial robots of distinct types or specifications?

The following two research questions ("Does the modular middleware layer allow simple and quick integration of handler integration modules?" and "Is the middleware layer scalable so that new features for the manipulators can be incorporated?") can also be answered positively due to the structure of the middleware layer. The Robot Control module framework allows adding new functions defined by the manufacturer or the user. Each API function acts as a sub-module of Robot Control, so the process of adding new functionalities is simple and intuitive.

Several experiments were run to evaluate the performance of the middleware layer. Also, to address the ability to receive and interpret the robot's state, to validate its capability to formulate and send commands to move the robot. More specifically, the robot status information from the middleware layer was compared with what the controller periodically sends. The rate at which information is updated was also measured. The obtained results provide a positive answer to the following research question (Does the middleware layer fulfil the existing real-time constraints?) since the middleware layer can read the received data at a rate between 3 and 9 Hz. For middleware layer applications, this rate is enough, although there is the possibility of replacing the secondary client interface with the real-time interface of Universal Robots, which would allow higher update rates.

Lastly, a usability testing method was performed to verify that our solution behaves appropriately and that the key features are easily accessible to different types of users. This method was carried out with 13 participants with distinct robotics profiles: experts and non-experts. The difference in the participants' backgrounds is helpful to validate the solution's simplicity. The results were again positive since every participant was either satisfied or very satisfied with the GUI's usability. Moreover, they found the learning curve relatively smooth and would use the GUI to program the robots if needed. This feedback contributed to a positive answer to the last research question: "Considering inexperienced operators in coding, does the Graphical user Interface simplify the integration of different robots?".

In sum, the proposed middleware solution allowed us to achieve a simple, general and flexible approach to the robotic arm integration problem. Transparency and modularity were also key concepts throughout the development. Even though we achieved all the expected objectives, it is important to note some limitations. Besides the update frequency of the robot's information being modified compared to what is announced by the manufacturers, the main limitation of this work is programming cyclical tasks. Despite being possible to use the middleware layer to create routines through code, the GUI does not allow it. Therefore, unlike many manufacturers' software, it is impossible to program the robot through a flow diagram.

6.2 Future Work

Throughout the design and implementation of the solution, simplicity and modularity were highly relevant aspects so that a solid foundation for future upgrades is more likely to be achieved from the beginning. Since the middleware layer is modular and highly scalable, there is plenty of room for further development.

Therefore, according to the user's needs, other motion functions can be developed and easily integrated into the API. An example would be, through 3 non-collinear points, to define a new plane. This plane can then serve as a new working frame, and all movements are relative to it. This is particularly useful because robots often only operate in a limited area. Therefore, it is more intuitive to use the Cartesian coordinates of the working area as a target instead of the global coordinates of the base frame.

On the other hand, to develop the GUI, we were also careful to use user-friendly technologies, allowing for easy improvements. As future work, we intend to allow the user to graphically program the robots using the GUI. With the help of a menu containing the movement functions represented by icons, it would be enough to drag and drop them onto the diagram and draw the robot's path.

References

[1] Rou Cao, Xudong Ma, Chengjing Yu, and Pengcheng Xu. Framework of industrial robot system programming and management software. In *2019 14th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, pages 1256–1261. IEEE, 2019.

[2] Ayssam Elkady and Tarek Sobh. Robotics middleware: A comprehensive literature survey and attribute-based bibliography. *Journal of Robotics*, 2012, 2012.

[3] Alexei Makarenko, Alex Brooks, and Tobias Kaupp. Orca: Components for robotics. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 163–168. Citeseer, 2006.

[4] Carle Côté, Dominic Létourneau, Clément Raïevsky, Yannick Brosseau, and François Michaud. Using marie for mobile robot component development and integration. In *Software Engineering for Experimental Robotics*, pages 211–230. Springer, 2007.

[5] Stefan Profanter, Ari Breitkreuz, Markus Rickert, and Alois Knoll. A hardware-agnostic opc ua skill model for robot manipulators and tools. In *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1061–1068. IEEE, 2019.

[6] Halldor Arnarson, Bjørn Solvang, and Beibei Shu. The application of open access middleware for cooperation among heterogeneous manufacturing systems. In *2020 3rd International Symposium on Small-scale Intelligent Manufacturing Systems (SIMS)*, pages 1–6. IEEE, 2020.

[7] Sergii Iarovyi, José L Martinez Lastra, Rodolfo Haber, and Raúl del Toro. From artificial cognitive systems and open architectures to cognitive manufacturing systems. In *2015 IEEE 13th international conference on industrial informatics (INDIN)*, pages 1225–1232. IEEE, 2015.

[8] Emanuele Gabriel Margherita and Alessio Maria Braccini. Managing the fourth industrial revolution: A competence framework for smart factory. In *The Fourth Industrial Revolution: Implementation of Artificial Intelligence for Growing Business Success*, pages 389–402. Springer, 2021.

[9] Pedro Neto and Nuno Mendes. Direct off-line robot programming via a common cad package. *Robotics and Autonomous Systems*, 61(8):896–910, 2013.

[10] William D Smart. Is a common middleware for robotics possible? In *Proceedings of the IROS 2007 workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware*. Citeseer, 2007.

[11] Raimund Edlinger, Lydia Leimer, Michael Zauner, and Roman Froschauer. Towards a flexible industrial robot system architecture. In *Proceedings of the Joint ARW & OAGM Workshop, Pichler, A.; Roth, PM*, pages 60–64, 2019.

[12] Steve Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications magazine*, 35(2):46–55, 1997.

[13] Sumair Khan, Kalim Qureshi, and Haroon Rashid. Performance comparison of ice, horb, corba and dot net remoting middleware technologies. *International Journal of Computer Applications*, 3(11):15–18, 2010.

[14] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[15] Pablo Iñigo-Blasco, Fernando Diaz-del Rio, Ma Carmen Romero-Ternero, Daniel Cagigas-Muñiz, and Saturnino Vicente-Diaz. Robotics software frameworks for multi-agent robotic systems development. *Robotics and Autonomous Systems*, 60(6):803–821, 2012.

[16] Bernhard Dieber, Severin Kacianka, Stefan Rass, and Peter Schartner. Application-level security for ros-based applications. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4477–4482. IEEE, 2016.

[17] Herman Bruyninckx. Open robot control software: the orocos project. In *Proceedings 2001 ICRA. IEEE international conference on robotics and automation (Cat. No. 01CH37164)*, volume 3, pages 2523–2528. IEEE, 2001.

[18] Harrisson Fischer, Philippe Vulliez, Jean-Pierre Gazeau, and Saïd Zeghloul. An industrial standard based control architecture for multi-robot real time coordination. In *2016 IEEE 14th International Conference on Industrial Informatics (INDIN)*, pages 207–212. IEEE, 2016.

[19] Hans Utz, Stefan Sablatnog, Stefan Enderle, and Gerhard Kraetzschmar. Miro-middleware for mobile robot applications. *IEEE Transactions on robotics and automation*, 18(4):493–497, 2002.

[20] Carle Cote, Yannick Brosseau, Dominic Letourneau, Clément Raïevsky, and Francois Michaud. Robotic software integration using marie. *International Journal of Advanced Robotic Systems*, 3(1):10, 2006.

[21] Gergely Magyar, Peter Sinčák, and Zoltán Krizsán. Comparison study of robotic middleware for robotic applications. In *Emergent Trends in Robotics and Intelligent Systems*, pages 121–128. Springer, 2015.

[22] James Kramer and Matthias Scheutz. Development environments for autonomous mobile robots: A survey. *Autonomous Robots*, 22(2):101–132, 2007.

[23] Nader Mohamed, Jameela Al-Jaroodi, and Imad Jawhar. A review of middleware for networked robots. *International Journal of Computer Science and Network Security*, 9(5):139–148, 2009.

[24] Stefan Profanter, Alexander Perzylo, Markus Rickert, and Alois Knoll. A generic plug & produce system composed of semantic opc ua skills. *IEEE Open Journal of the Industrial Electronics Society*, 2:128–141, 2021.