UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO DI TECNICA E GESTIONE DEI SISTEMI INDUSTRIALI

CORSO DI LAUREA IN INGEGNERIA MECCATRONICA

Human-Guided Autonomous Mobile Manipulator in Dynamic Environments: A Case Study Using LoCoBot

Advisor
Prof.ssa Giulia Michieletto

Co-Advisor
Prof.ssa Monica Reggiani

Candidate
Alessio Lovato

ACADEMIC YEAR 2023-2024

Yesterday it worked

Abstract

In the context of Industry 4.0, human-robot interaction emerges as one of the major topics to relieve human operators from heavy and repetitive tasks. The objective of this research is to develop a real-world human-robot interaction to help humans with tool transportation. In this study case, a mobile robot equipped with a robotic arm is tasked to autonomously follow a human operator through a predefined space while avoiding dynamic obstacles and carrying a payload. Once the human operator has reached the destination, the robot has to transfer the item to him/her safely. The platform is controlled using the ROS 2 middleware; all the subtasks required to perform the entire operation have been implemented through the adaptation of pre-existing packages into a unique framework. A depth camera is incorporated to detect dynamic obstacles in the path of the mobile robot, while external cameras are used to obtain the poses of both the robot and the individual during the entire operation. These are determined using fiducial marker-based computer vision. To control the behavior of the robot, a state machine is implemented and the changes between states are performed using hand gesture recognition to enhance interactivity. The proposed framework is evaluated through simulation tools, followed by implementation in a real-world scenario.

Contents

Abbreviations

Abbreviation	Meaning
AMM	Autonomous Mobile Manipulator
AMR	Autonomous Mobile Robot
API	Application Programming Interface
BT	Behavior Tree
CNN	Convolutional Neural Network
DDS	Data Distribution Service
DoF	Degree of Freedom
GPS	Global Positioning System
HRC	Human Robot Collaboration
HRI	Human Robot Interaction
IDL	Interface Description Language
IMU	Inertial Measurement Unit
KPI	Key Performance Indicators
LIDAR	Laser Imaging Detection and Ranging
LLM	Large Language Model
MPPI	Model Predictive Path Integral
OMG	Object Management Group
QoS	Quality of Service
RMW	ROS MiddleWare
ROS	Robot Operating System
RPP	Regulated Pure Pursuit
SRDF	Semantic Robot Description Format
STVL	Spatio-Temporal Voxel Layer
TEM	Trajectory Execution Manager
TF	Transform
TTL	Transistor-Transistor Logic
UGV	Uncrewed Ground Vehicle
URDF	Unified Robot Description Format

List of Figures

Chapter 1

Introduction

Contemporary robotics permeate daily life, assisting humanity with tasks ranging from the simplest to fully automated factory operations. The term *robot* was originally introduced by Karel Čapek in his play *R.U.R. (Rossumovi univerzàlnì roboti)* in the 1920s [1], symbolizing humanoid entities capable of performing human tasks.
The current interpretation of the term *robot* refers to a "*programmed actuated mechanism with a degree of autonomy to perform locomotion, manipulation or positioning*" — as defined by ISO 18646 [2] — encompassing a wide variety of machines with diverse morphology. These range from bio-inspired robots, like those with humanoid or serpentine shapes, to stationary robotic arms designed for lifting and transport, and extend to mobile robotics, which is a major focus of modern research.

As the term suggests, a mobile robot is a "*robot able to travel under its own control*" [2]. Mobile robotics encompasses a broad category of systems, including subsets like Uncrewed Ground Vehicles (UGVs). While some UGVs rely on human teleoperation — for example, robots used for inspecting hazardous environments — others, such as Autonomous Mobile Robots (AMRs), are designed to navigate their surroundings independently, without human intervention.
Navigation can be achieved offline, where the robot follows a pre-planned path, or online, where the robot computes its path in real time, based on environmental cues like RF tags or obstacles detected by onboard sensors and behavior algorithms developed through software.

Movement mechanisms typically include tracks, wheels, or legs. While wheels and tracks are primarily used in indoor environments where conditions favor these forms of locomotion, legged robots are often preferred in rougher environments, as their structure offers greater degrees of freedom and agility.

Recent advancements have seen indoor tracked robots being replaced by wheeled robots with holonomic wheels. These holonomic wheels allow rotation on the spot and enable omnidirectional movement, optimizing space usage and reducing maintenance costs — key factors driving industrial research and implementation.

The industrial environment presents significant challenges for mobile robotics due to the continuous presence of humans, static objects such as walls and racks, and dynamic objects that must be avoided or navigated around. Modern sensors such as LIDAR, ultrasound, and depth cameras facilitate navigation by converting the external world into digital 2D/3D maps for robotic use.

A robot's ability to localize itself within its environment is another essential aspect of navigation. Methods like Simultaneous Localization and Mapping (SLAM), where the robot builds a map as it navigates, or Adaptive Monte Carlo Localization (AMCL), which uses sensor data (e.g., IMU, LIDAR) to locate the robot within a pre-existing map, are commonly used.

Additionally, external methods for the estimation of the pose (*combination of position and orientation in space* [2]) can be employed, including GPS, motion capture, RF signal triangulation, and computer vision. As reported in [3], many computer vision techniques, such as visual odometry and fiducial markers, can be used to determine the robot's pose.

In outdoor applications, GPS is often the preferred technology for its reliability and familiarity. However, for more challenging terrains, LIDAR is frequently used to scan the surroundings and create a 3D map of the environment. Although LIDAR is widely used indoors as well, GPS is typically replaced by Visual Odometry or RF signal triangulation when working in enclosed spaces.

A key emerging trend in robotics and mobile robotics is human-robot interaction (HRI) and collaboration (HRC). According to [4], a collaborative operation is defined as *"state in which a purposely designed robot system and an operator work within a collaborative workspace"*. Therefore, conducting human-robot collaboration within an industrial context necessitates strict adherence to this normative and adoption of proper robot to ensure safety. In contrast, the interaction between humans and robots is less restrictive, even within a collaborative workspace, as the two entities undertake distinct tasks, thereby enhancing safety.

State-of-the-art Autonomous Mobile Manipulators (AMMs) [5–10] are equipped with advanced sensors such as LIDAR and cameras, enabling them to detect and navigate around obstacles and humans along their paths with high precision. The robotic arms integrated with these AMMs are designed for collaboration, featuring force-feedback mechanisms to ensure

Figure 1.1: Example of state-of-the-art Autonomous Mobile Manipulator (Robotnik RB−KAIROS+UR5e). Source [11]

safe interactions with the environment. In the event of a collision with an object or person, these arms can safely halt the process, greatly reducing the risk of injury or damage.
High payload requirements often necessitate larger chassis and more powerful drive systems, increasing the overall cost of the robot. Lastly, robust designs are essential for meeting the demands of industrial applications, where reliability and safety are paramount.
Figure 1.1 provides an example of these AMM.

This study takes an industrial-oriented approach by proposing a practical scenario in which an autonomous mobile manipulator assists a human in completing tasks. Specifically, the robot is tasked with transporting an object while following a human through a predefined map, avoiding both static and unknown dynamic obstacles.
To achieve this objective, this thesis focuses on the development of a robust framework to control the LoCoBot WX200, an AMM developed by Carnegie Mellon University and supported by Trossen Robotics.

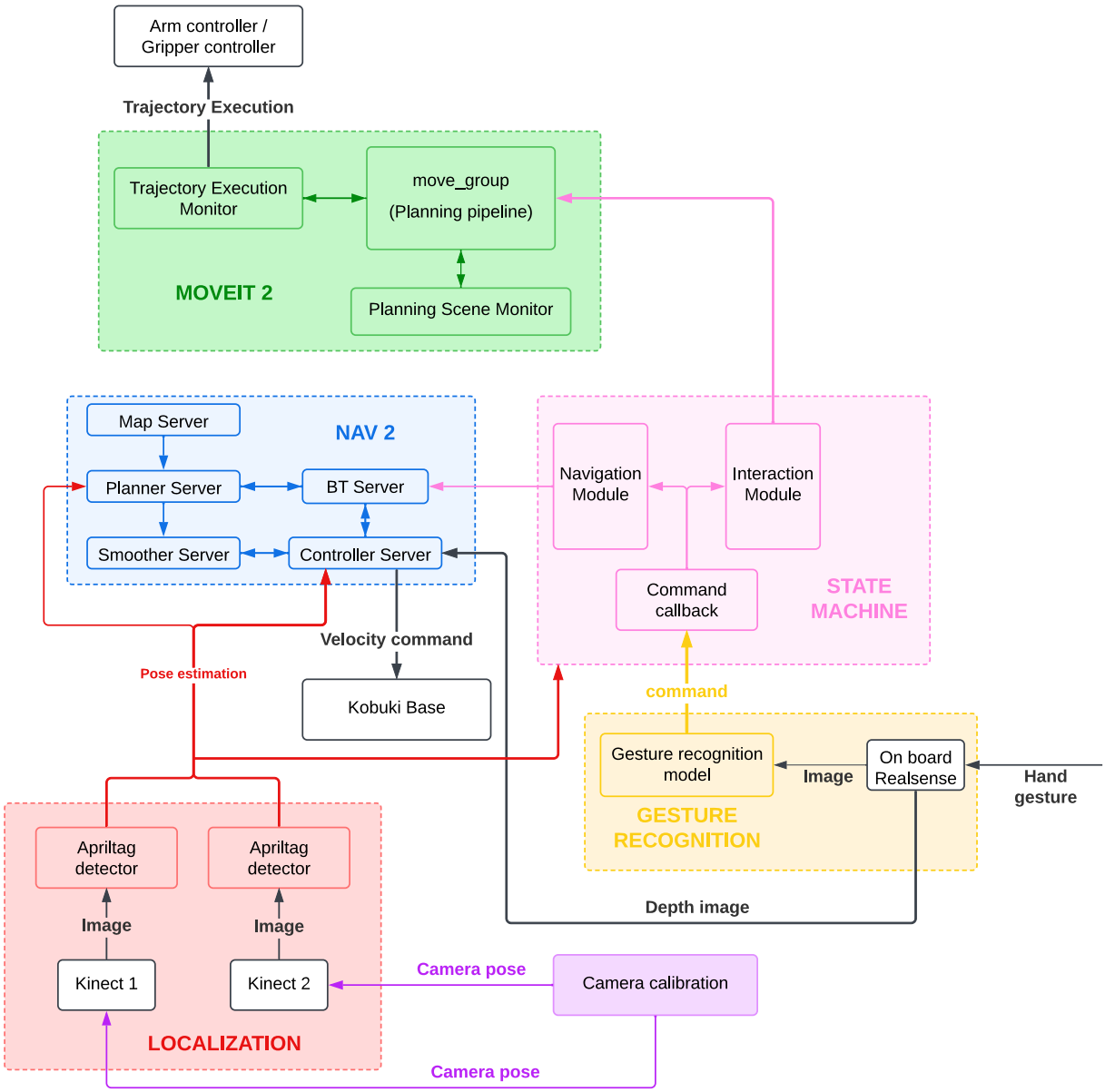


Figure 1.2: Block diagram of the proposed framework

In *Figure 1.2*, the framework proposed in this thesis is illustrated. A key feature of this framework is the high modularity of its components, an attribute inherited from its foundation on the ROS 2 framework, which itself is designed for modularity and scalability.

The primary objective of the proposed framework is to enable human-robot interaction through gesture recognition, leveraging computer vision techniques. Gestures are captured using a pre-trained model, which performs recognition based on the keypoints of the hand. The recognized gesture is then translated into a command for the state machine.

Depending on its current state and the nature of the received command, the state machine interprets the command and triggers either the navigation or interaction module. These

modules, provided by the Nav2 and MoveIt 2 stacks — widely recognized standards in robotic navigation and manipulation, respectively — handle the execution of low-level tasks.

Accurate navigation requires the robot's position within the environment to be known. To achieve this, a computer vision-based localization system utilizing fiducial markers has been employed. This approach not only determines the robot's position but also tracks the human operator, enabling seamless human-following behavior using the same technology.
Since the camera system plays a crucial role in computer vision tasks, its spatial relationship with the environment's origin must be well-defined. To ensure this, a calibration methodology has been implemented to determine the static positions of all cameras within the environment.

To validate the framework, a comprehensive proof-of-concept demonstration is presented in this thesis. This includes a general tests to confirm the framework's functionality and performance.

Thesis Structure

The structure of this thesis is as follows:

- Chapter 2 provides an in-depth overview of the LoCoBot system, detailing the modifications made to Trossen Robotics' ROS 2 packages to operate the LoCoBot.
- Chapter 3 introduces the foundational tools used to develop the framework. It begins with an overview of basic ROS 2 concepts, which are essential for understanding the terminology and architecture employed in this work. Next, it provides a brief analysis of the pipelines utilized by Nav2 and MoveIt 2 for planning and executing navigation and robotic arm motions. Finally, the chapter outlines the tools and techniques used for the localization.
- Chapter 4 focuses on the framework's architecture, including customizations made to MoveIt 2 and Nav2, the implementation of the gesture recognition module, and the design of the state machine.
- Chapter 5 presents the validation process, describing the tests performed for individual modules and for the overall framework.
- Chapter 6 concludes the thesis by summarizing the results and identifying potential future implementations.

Chapter 2

LoCoBot - the case study AMM

The autonomous mobile platform used is a LoCoBot WidowX-200 (Low Cost Robot) developed by Carnegie Mellon University [12] and supported by Trossen Robotics [13].
LoCoBot is built on a third party mobile base, while the robotic arm is designed using DYNAMIXEL motors [14]. Hardware communication is handled via ROS 2 using an onboard computer, so that the whole system can be used as a mobile manipulator, able to perform tasks autonomously. This chapter will cover the hardware of the LoCoBot in detail (*Section 2.1*) and the various ROS 2 packages developed by Trossen Robotics to control the system (*Section 2.2*).

2.1 Hardware

In addition to the base and the robotic arm, other components are integrated into the LoCoBot. The "brain" of the AMM is an Intel® NUC (*Next Unit of Computing*), powered by a MAXOAK K2 50,000mAh powerbank.
Additionally, an aluminum frame tower supports the installation of an Intel® RealSense™ camera, whose pan and tilt are controlled through two DYNAMIXEL motors.

2.1.1 Mobile base

The mobile base used for the LoCoBot is the Kobuki Base, developed by Yujin Robot.
This base uses differential-drive movement and equips caster wheels for balance. Furthermore it integrates with ROS 2 to receive velocity commands through the `ros2_control` package (*Appendix A*).

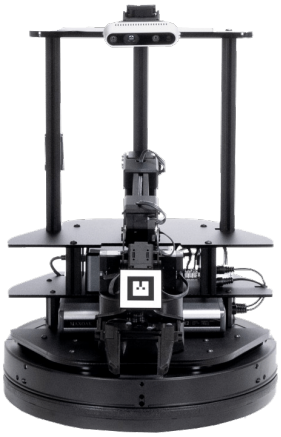


Figure 2.1: LoCoBot WidowX-200. Source [15]

Kinematics of differential driven mobile bases

A differential-driven robot can move forward, backward, and rotate around an instantaneous center of rotation (ICR). The motion is provided by two motorized wheels positioned at opposite ends of the base, which can either be driven independently or coupled with a differential gear. If both wheels move in the same direction and at the same speed, the robot will move linearly. If the speeds differ, angular movement is produced.

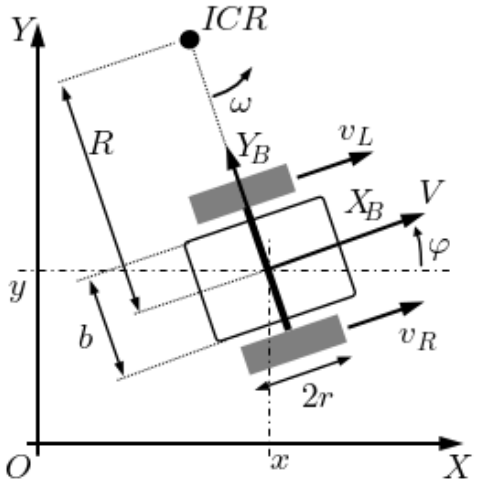


Figure 2.2: Differential drive kinematics. Source [16]

Let $r > 0$ be the radius of the wheels, $R \geq 0$ the distance between the ICR and the mobile base center, $b > 0$ the width of the vehicle and v_L, $v_R \in \mathbb{R}$ the modules of left and right wheel ground contact speed, respectively.

Referring to *Figure* 2.2, considering X and Y as the global coordinate system, $(X_B, Y_B) \in \mathbb{R}^2$ as the position of the base in the global coordinate system, and $\varphi \in [0, 2\pi]$ as the planar rotation of the base respect to the X-axis, it is possible to define the module of angular velocity $\omega \in \mathbb{R}$ of the vehicle as:

$$\begin{cases} \omega(R + \frac{b}{2}) = v_R \\ \omega(R - \frac{b}{2}) = v_L \end{cases}$$

Solving those equations for ω results in:

$$\begin{cases} \omega = \dfrac{v_R - v_L}{b} \\ R = \dfrac{\frac{b}{2}(v_R + v_L)}{v_R - v_L} \end{cases}$$

The instantaneous velocity V of the base can be obtained from the angular velocity as

$$V = \omega R = \frac{v_R + v_L}{2}$$

with $v_R = r \cdot \omega_R$ and $v_L = r \cdot \omega_L$, where $\omega_L, \omega_R \in \mathbb{R}$ are the modules of angular velocities of the wheels. The kinematics expressed in local body coordinates is

$$\begin{bmatrix} \dot{x}_B \\ \dot{y}_B \\ \dot{\varphi} \end{bmatrix} = \begin{bmatrix} \frac{r}{2} & \frac{r}{2} \\ 0 & 0 \\ -\frac{r}{b} & \frac{r}{b} \end{bmatrix} \begin{bmatrix} \omega_L \\ \omega_R \end{bmatrix}$$

and using the planar rotation matrix $\mathbf{R_z}(\varphi) \in \mathbb{R}^{\mathbf{3\times 2}}$ the kinematic model in global coordinates becomes:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\varphi} \end{bmatrix} = \mathbf{R_z}(\varphi) \begin{bmatrix} V \\ \omega \end{bmatrix} = \begin{bmatrix} \cos\varphi & 0 \\ \sin\varphi & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} V \\ \omega \end{bmatrix}$$

2.1.2 Robotic arm

The robotic arm utilized is the WidowX-200 from Trossen Robotics [17]. This arm supports a maximum payload of 200g, with a reach of 550mm, and a total span of 1100mm, althought the recommended workspace has a radius of 770mm . It is driven by 7 servo motors (DYNAMIXEL XM430-W350 and DYNAMIXEL XL430-W250) allowing for 5 degrees of freedom (DoF). The number of motors equipped is higher than the DoF because, to support the arm, the shoulder

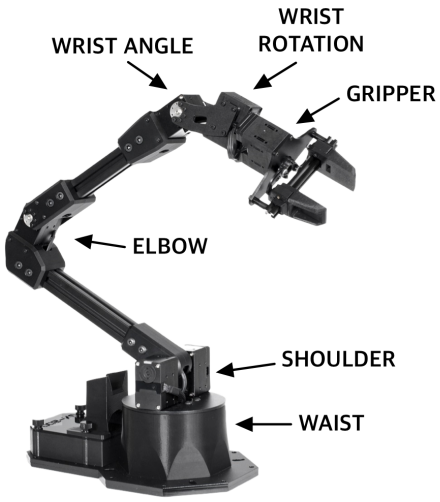


Figure 2.3: WidowX-200 arm's motors

joint (*Figure 2.3*) is equipped with two twin motors and an additional motor is used to control the movement of the gripper.
The motors' communication line is set up in a daisy-chain configuration, with each motor assigned a unique ID. All motors are controlled via the ROBOTIS U2D2 module, which converts USB signals from the NUC into Transistor-Transistor Logic (TTL) serial signals, subsequently sent to the motors [18].
Table 2.1 outlines the joint limits of the WidowX-200 arm. The joint angles are measured with respect to a centered position, and the gripper's limit refers to its translational movement, achieved by a horn mechanism connected to the motor rotor.

Joint	Min	Max	Servo ID
Waist	-180 deg	180 deg	1
Shoulder	-108 deg	113 deg	2+3
Elbow	-108 deg	93 deg	4
Wrist Angle	-100 deg	123 deg	5
Wrist Rotate	-180 deg	180 deg	6
Gripper	30 mm	74 mm	7

Table 2.1: Joint limits and IDs of WidowX-200 arm.

DYNAMIXEL motors

DYNAMIXEL motors, developed by ROBOTIS, are specifically engineered for robotics applications. These compact motors offer high payload capacity due to their elevated gear ratio and they can be controlled in various modes, including position, velocity, current (torque), and

PWM control. A PID control system is incorporated for position and velocity regulation, allowing adjustments to be made to suit specific applications. Real-time feedback on parameters such as position, velocity, current, temperature, and voltage is also provided.
The motors are controlled through the U2D2 module, which supports multiple development environments such as C++, Python and MATLAB® , across different operating systems (Linux, Windows, MacOS and RTOS). The WidowX-200 arm uses two types of motors:

- **XM430-W350**: Mid-level performance motor, powered by a 12V input, with a gear ratio of 350:1. The ”430” refers to the dimensional class.
- **XL430-W250**: Low-cost motor, also powered by a 12V input, with a gear ratio of 250:1. Used for wrist rotation and gripper.

Kinematics in robotic arms

The kinematics of a robotic arm can be approached from two perspectives: forward kinematics and inverse kinematics. Forward kinematics involves determining the end-effector's pose from given joint values, while inverse kinematics addresses the opposite problem.
Forward kinematics can be computed using methods such as the Denavit-Hartenberg convention or the Product of Exponentials (PoE) formula.
LoCoBot documentation provides the Poe method, where $\mathbf{T_0} \in \mathbb{R}^{4\times4}$ represents the home configuration of the end-effector, and $\mathbf{S\_list} \in \mathbb{R}^{6\times5}$ contains the screw axes for each joint.

$$\mathbf{T_0} = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.408575 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.31065 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

$$\mathbf{S_{list}} = \begin{bmatrix} \mathbf{S_1} & \mathbf{S_2} & \mathbf{S_3} & \mathbf{S_4} & \mathbf{S_5} \end{bmatrix} = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 1.0 & 1.0 & 1.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & -0.11065 & -0.31065 & -0.31065 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.31065 \\ 0.0 & 0.0 & 0.05 & 0.25 & 0.0 \end{bmatrix}$$

The transformation matrix, $\mathbf{T}(\boldsymbol{\theta}) \in \mathbb{R}^{4\times4}$, which represents the pose of the end effector relative to the base frame, is calculated by multiplying the exponentials of the screw axes, scaled by their corresponding joint variables:

$$\mathbf{T}(\boldsymbol{\theta}) = e^{\mathbf{S_1}\theta_1} e^{\mathbf{S_2}\theta_2} e^{\mathbf{S_3}\theta_3} e^{\mathbf{S_4}\theta_4} e^{\mathbf{S_5}\theta_5} \mathbf{T_0}$$

Although the Poe represents a more efficient method for computational analysis, Denavit-Hartenberg provides a more intuitive table to understand the relative position of each joint. The transformation matrix, $\mathbf{T_i} \in \mathbb{R}^{4\times4}$, maps the frame i to frame $i-1$. In representing a joint position within three-dimensional space, the vector $\mathbf{j} \in \mathbb{R}^{4\times1}$ is used.

$$\mathbf{j_i} = \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix} = \mathbf{T_i}\,\mathbf{j_{i-1}} = \begin{bmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{i-1} \\ y_{i-1} \\ z_{i-1} \\ 1 \end{bmatrix}$$

Within the Denavit-Hartenberg notation, $a_i \in \mathbb{R}^+$, signifies the length of the common normal, defined as the distance between frames i and $i-1$ along the x-axis of frame $i-1$. The term $\alpha_i \in [0, 2\pi]$ enumerates the angle between the z-axes around the common normal. $d_i \in \mathbb{R}^+$ delineates the link offset, describing the distance from frame $i-1$ to frame i along z-axis of frame $i-1$. The joint angle, identified by $\theta_i \in [0, 2\pi]$, denotes the rotational movement around the z-axis of frame i.

$\mathbf{T_i}$	$\boldsymbol{\alpha_i}$ [rad]	$\mathbf{a_i}$ [mm]	$\boldsymbol{\theta_i}$ [rad]	$\mathbf{d_i}$ [mm]
$\mathbf{T_1}$	0	0	θ_1	113,25
$\mathbf{T_2}$	-$\frac{pi}{2}$	0	θ_2	0
$\mathbf{T_3}$	0	206,16	θ_3	0
$\mathbf{T_4}$	0	200	θ_4	0
$\mathbf{T_5}$	-$\frac{pi}{2}$	0	θ_5	174,15

Table 2.2: Denavit-Hartenberg table for Widow-X 200 arm

Table 2.2 is obtained by applying the Denavit-Hartenberg method to the WidowX-200 arm.
As shown in *Figure* 2.4, a simplification has been introduced between frames 2 and 3. The arm's actual structure between these two joints is formed by an L-shaped connection, allowing to consider the hypotenuse of the triangle created by this junction as the minimal distance between the axes, eliminating the need for an additional dummy frame.

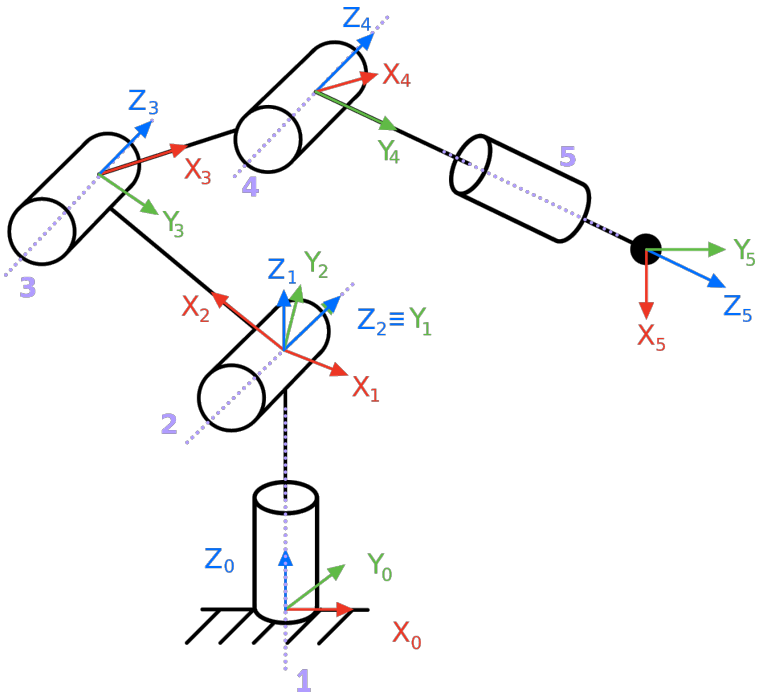


Figure 2.4: Frame position of the joints of the WidowX-200 arm for Denavit-Hartenberg

Inverse kinematics, on the other hand, is more challenging and can be solved either analytically using algebraic methods or numerically using iterative techniques, such as the Jacobian matrix or Newton-Raphson method.

2.1.3 INTEL® NUC

The computing platform used in this project is an Intel® NUC, featuring an Intel® Core™ i5-7260U CPU, 8 GB of DDR4 RAM, and a 240GB SSD. The NUC provides native Bluetooth and Wi-Fi connectivity, along with HDMI and USB ports for peripheral connections. For this project, Ubuntu® 22.04 with ROS 2 Humble distribution has been installed on a dedicated partition.

2.1.4 RealSense™ camera

The Intel® RealSense™ D345 is a depth camera capable of generating 3D perceptions of the environment using infrared light. It has a field of view of 87° horizontally and 58° vertically, with an optimal depth range between 0.3 and 3 meters, and an accuracy of less than 2% at 2 meters. Additionally, the camera captures RGB Full HD images at 30 frames per second, with a field of view of 69° horizontally and 42° vertically.

Figure 2.5: Realsense camera D345. Source [19]

Using the `Realsense-ROS` package [20], the camera 3D depth data are published through a ROS 2 topic using the *PointCloud2* message type. The camera is mounted on two DYNAMIXEL motors that control the pan and tilt. In the updated version of the LoCoBot, a single DYNAMIXEL 2XL motor is equipped, offering the same feature. Each axis of this latter motor is assigned a unique ID to manage independent movement.

2.2 LoCoBot ROS 2 packages

The LoCoBot is part of Trossen Robotics's Interbotix product line, which focuses on open-source solutions tailored for research and development purposes. The Interbotix series implements both shared and specialized ROS 2 packages. The specialized packages are tailored to specific applications (such as LoCoBot or X-Arms), whereas shared packages provide universal functions across all products.
These packages are designed to simplify the integration of hardware and software, enabling users to quickly set up and operate robots. Although well documented and robust, some minor adjustments were necessary during the implementation of the packages.

2.2.1 Interbotix native packages

The main Interbotix ROS 2 packages required to operate the LoCoBot are outlined below.

- **Interbotix_ros_core** [21]: This package provides ROS 2 wrappers around various types of actuators used by Interbotix robots, such as the DYNAMIXEL motors.

- **Interbotix_ros_toolboxes** [22]: This repository contains toolboxes shared among different families of products.

- **Interbotix_ros_rovers** [23]: This package is built on top of the previous packages and includes specific modules designed for the LoCoBot and other rovers from the Interbotix family.

The aforementioned packages include several ROS 2 packages within them. In the following, a detailed overview will be provided specifically for those included in the `Interbotix_ros_rovers` package.

Interbotix_xslocobot_descriptions

This package is responsible for providing the Unified Robot Description Format (URDF) file of the LoCoBot model, which describes the robot's physical configuration. It publishes the state of all non-fixed joints via the `joint_state_publisher` node [24]. Using these data, the `robot_state_publisher` node [25] computes the robot's forward kinematics, publishing the results to the *tf* topic. This allows real-time visualization of the robot state in RViz 2 [26], and can be used for both real robots and simulations.

Interbotix_xslocobot_control

The control package builds upon the robot's description and integrates the `Interbotix_xs_sdk` package [27], which is responsible for controlling the DYNAMIXEL motors. The `dynamixel_workbench_toolbox` [28] is used to manage the low-level operations of the DYNAMIXEL motors. This package also launches key components, including the Kobuki base nodes and the RealSense™ camera node.

Interbotix_xslocobot_sim

This package facilitates the simulation of the LoCoBot in Gazebo Classic [29], a widely used robotic simulator. The package utilizes the `gazebo_ros` package from the `gazebo_ros_pkgs` collection [30] to simulate the robot in a virtual environment. When using this package, the LoCoBot's hardware components (e.g., motors, sensors) are replaced with simulated counterparts in Gazebo, but the same control architecture is maintained.

Interbotix_xslocobot_moveit

This package contains the configuration files used to run MoveIt 2 (*Section 3.3*) with any robot of the LoCoBot family. It implements the *FollowJointTrajectory* interface both in simulation and on real robots. *Figure 2.6* shows the complexity and inclusions of this package.
It is essential to highlight that the launch file associated with this package is dedicated solely to arm control. Consequently, when the launch file of the `Interbotix_xslocobot_ros_control` is included, the `use_base` parameter is purposely not passed to it.

Interbotix_xslocobot_ros_control

This package loads the necessary controllers and controller manager from `ros2_control` package (*Appendix A*), properly tuned for the LoCoBot application.
Furthermore it includes the launch file from `Interbotix_xslocobot_control` package.

Interbotix_xslocobot_nav

Specifically designed for real-world applications, this package provides the necessary configuration and launch files to run SLAM and autonomous navigation using the ROS 2 Nav2 stack (*Section 3.2*). It is optimized for the LoCoBot's sensor suite and facilitates autonomous exploration and navigation in dynamic environments.

2.2.2 Interbotix custom packages

The modifications focused on two packages: `Interbotix_ros_rovers` and `Interbotix_ros_toolbox`. Both packages failed to operate the LoCoBot properly under the ROS2 Humble distro.
The major issues encountered were the arm not responding to commands given via the MoveIt 2 interface (*Section 3.3*), and the depth camera's perception was not aligned with the real world. Additionally, the robot in the simulation was drifting around even without any given velocity commands. These modifications were implemented by creating a fork of the repositories on GitHub, with those used in this thesis available at `https://github.com/Alessio-Lovato-Unipd/interbotix_ros_rovers/tree/humble` and `https://github.com/Alessio-Lovato-Unipd/interbotix_ros_toolboxes/tree/humble`.
It is important to note that all modifications were made specifically for the LoCoBot WX200, as it was not possible to test the changes with other models.

The packages modified from the repository `Interbotix_ros_rovers` are:

- `Interbotix_xslocobot_description`
  The URDF of the Kobuki base was assigning an incorrect link name to the Gazebo reference of the link (e.g., *wheel_right_link* instead of *locobot/wheel_right_link*). This mismatch prevented parameters assigned to the link (such as friction coefficients) from being correctly loaded. Additionally, the friction values were increased to keep the simulated LoCoBot stationary when no external velocity commands were sent to the controller.
  In the URDF of the camera (*pan_and_tilt.urdf.xacro*), a parameter was incorrectly defined

as *frameName* instead of *frame_name*, causing the point cloud generated from the camera to be oriented incorrectly relative to the model.

- `Interbotix_xslocobot_control`
  This issue concerned the configuration file of the RealSense™ camera. Specifically, variable names used to set RGB and depth profiles in the RealSense™ launch file were incorrect, preventing the adjustment of these values during startup.

- `Interbotix_xslocobot_moveit`
  This package encountered several issues due to incorrect link naming. All links were missing the *locobot/* prefix in the *locobot_wx200.srdf.xacro* file. In the same file, it was also necessary to disable a collision between *arm_cradle_link* and *upper_arm_link* when adjacent, as this caused errors during robot startup: the model slightly differed from the actual components, and in the arm folded position, MoveIt 2 detected a false collision between the arm and cradle. Additionally, in the configuration file for the controller of the LoCoBot WX200, the joint for the right finger was added to publish its joint position. The default Rviz 2 configuration file was also updated since the original did not allow Rviz to load correctly. Lastly, the gripper's operating mode was changed from *position* to *linear_position* to enable MoveIt 2 to execute correct movements on the physical robot.

- `Interbotix_xslocobot_ros_control`
  In this package, the operating mode of the gripper was changed from *position* to *linear_position* within the configuration file.

Additionally, the package modified from `Interbotix_ros_toolboxes` is:

- `Interbotix_xs_ros_control`
  The issue encountered was specific to the LoCoBot application, as this repository's code is designed to be compatible with multiple applications. The `xs_hardware_interface`, responsible for receiving and processing joint states, was correctly configured to recognize the number of joints for the robotic arm model. However, when LoCoBot started with the configuration that also contains its mobile base, the joint states of the wheels were published before those of the arm. This caused the hardware interface to expect additional joint states, leading to attempts to access elements beyond the range of the available vector, resulting in out-of-range errors.

2.3 Conclusions

After an overview of the hardware used during the development of this thesis, a comprehensive review of all the necessary packages needed to operate the LoCoBot has been conducted. Despite the open-source nature of this project, it became evident that some packages were not functioning as expected and needed fixing. Most issues were related to incorrect parameter configurations, which, while not critical, highlights the need for more attention during development.
Once these packages were fixed, the LoCoBot system could finally be utilized as intended. This achievement paved the way for the development of the framework later discussed in this thesis.

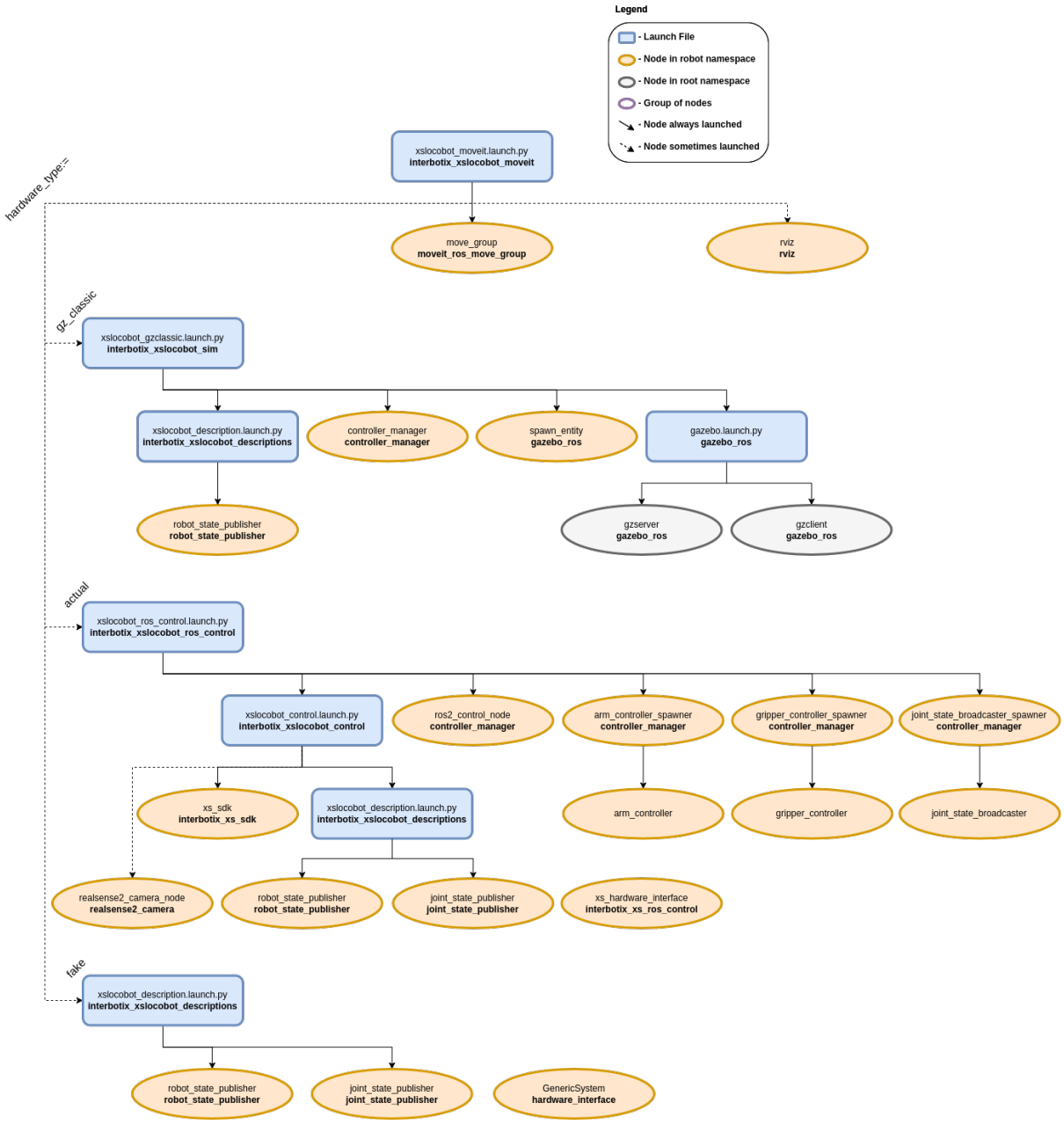


Figure 2.6: `Interbotix_xslocobot_moveit` package overview. Source [31]

Chapter 3

AMM maneuverability tools

Before introducing the framework concept, it is essential to understand its fundamental components. As shown in *Figure 3.1*, this chapter covers the key topics necessary for understanding the tools utilized to maneuver the LoCoBot.

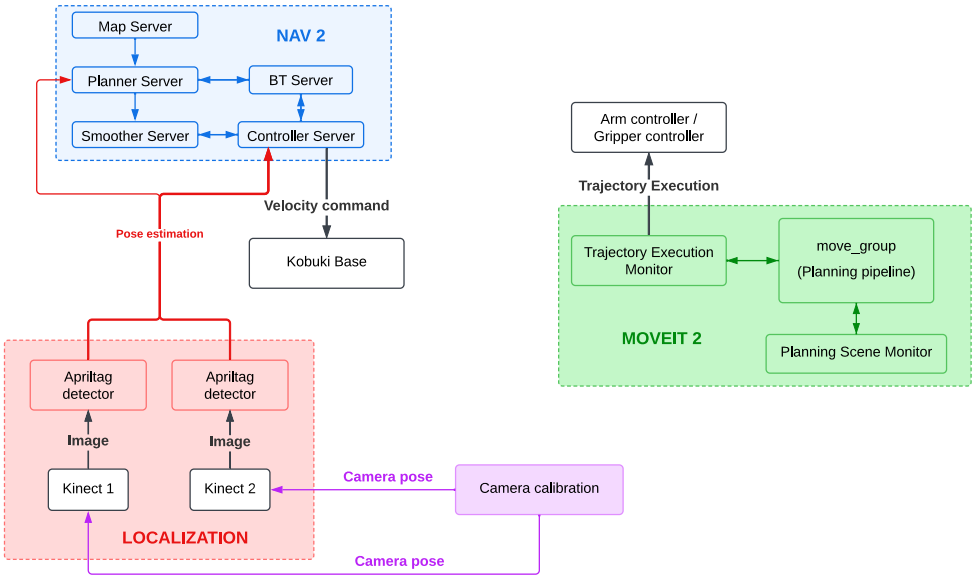


Figure 3.1: Overview of the framework components discussed in Chapter 3

The chapter begins with an introduction to the ROS 2 framework (*Section 3.1*), which forms the basis of this work. A detailed discussion of the Nav2 and MoveIt 2 packages follows in *Sections 3.2* and *3.3*, respectively. The chapter then delves into fiducial marker localization technology, with a particular focus on Apriltags, in *Section 3.4*. Finally, calibration techniques for the camera system are outlined in *Section 3.5*.

3.1 ROS 2

ROS 2 (Robot Operating System 2) is an open-source framework designed for developing robotic applications. ROS 2 has a modular architecture (*Section 3.1.2*) and can effectively manage distributed communications in complex and real-time environments [32].

In this section, the fundamental communication mechanisms in ROS 2 (Humble distribution) are explored. Specifically, the methods of node interaction — Topics, Services, and Actions — are explained, highlighting their roles and typical use cases.
Additionally, the importance of the Data Distribution Service (DDS) and the ROS Middleware Layer (RMW), which form the backbone of ROS 2's communication framework, is examined.

3.1.1 ROS 2 Network

The ROS 2 network is composed of nodes and their connections, forming what is called the *ROS graph*. Nodes integrate various components (such as publishers, subscribers, servers, and clients) and communicate with each another through the DDS.
Node discovery happens automatically: when a node initializes, it announces its presence and, when it shuts down, it signals its disconnection. Nodes also periodically broadcast their presence to allow new connections.
Communication between nodes is limited to those with compatible QoS settings [33].

ROS 2 nodes are compiled as executables. To avoid the need to create a different executable to launch each node, ROS 2 introduced the *Components*. These elements allow nodes to be compiled as a shared library [34], offering the flexibility to be either loaded into a container process or executed independently.
The primary advantage of loading multiple components into a single process is that it enables intra-process communication, which bypasses the need to publish data across the network, thus reducing communication overhead.

Finally, ROS 2 provides a node lifecycle management model, allowing nodes or *Components* to transition through various states such as unconfigured, inactive, active, and finalized. This model enables system integrators to control when specific nodes should be active, especially in distributed asynchronous systems.

Figure 3.2 illustrates examples of connections using node interfaces. In this figure, *Node C* creates a publisher for a topic to which *Node A* and *Node B* subscribe. The *topic* serves as the

channel through which the *Message*, or data packet, is transmitted.
Additionally, *Node C* establishes an *Action client* and a *Service Server*. These interfaces are typically used to perform tasks on demand. Detailed explanations of messages, topics, and more complex interfaces are provided below.

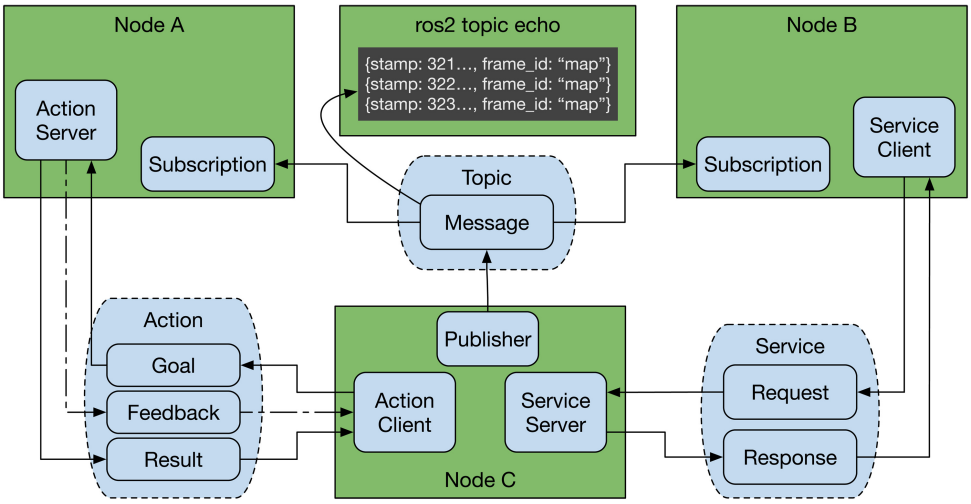


Figure 3.2: Examples of ROS 2 node interfaces: topics, services, and actions. Source [32]

Messages

Messages are data structures defined using an interface description language (IDL). ROS 2 supports two IDL formats: the ROS IDL format (.msg files) and the OMG IDL standard (.idl files). ROS IDL is designed for simplicity and ease of use, making it ideal for rapid message definition within the ROS ecosystem. In contrast, OMG IDL is an international standard that offers more complexity and flexibility, enabling greater interoperability across different systems and programming languages.
User-defined interface definitions are generated at compile time, producing the necessary code for communication in any supported client library language (*Section 3.1.2*).

Topics and Publisher/Subscriber

A *topic* serves as a shared identifier that allows publishers (data producers) and subscribers (data consumers) to find each other and exchange messages. A topic can support multiple publishers and subscribers simultaneously [35]. The topic interface is primarily used for continuous data streams.

Services

A ROS 2 Service is a mechanism for remote procedure calls [36]. It consists of two components: the *Service Server*, which receives a request and performs the required computation, and the *Service Client*, which sends the request and waits for the response.
The service model is ideal for short-duration operations, with one server potentially serving multiple clients. Requests and responses are based on messages defined by the ROS IDL.

Actions

ROS 2 Actions are similar to services but are designed for long-duration operations, offering feedback (via topic interface) during execution and allowing cancelation or preemption of goals [37].
Actions are divided into two components: the *Action Server*, which executes the goal and provides feedback, and the *Action Client*, which sends the goal request, monitors progress via the feedback and await the response upon completion of the goal.
As with services, it is preferable to have a single *Action Server* and multiple *Action Clients*. The Goal, Feedback, and Result constructs are defined using ROS IDL.

3.1.2 ROS 2 Architecture

ROS 2's architecture [32] consists of several key abstraction layers distributed across decoupled packages. These layers allow for flexibility in selecting different solutions for essential functionalities, such as middleware or logging frameworks.
Figure 3.3 shows the ROS 2 Client Library API stack, which consists of various libraries written in different programming languages. For example, *rclcpp* is used for C++ and *rclpy* for Python. These libraries provide streamlined access to core communication APIs for user applications. Moreover, they are built on top of the *rcl interface*, which offers shared functionality across them and acts as a bridge to the ROS Middleware (RMW) - "*rmw*" in *Figure 3.3* - layer.

ROS MiddleWare (RMW)

ROS 2 introduces the ROS Middleware abstraction layer [38] to support interoperability with multiple DDS implementations. Since the RMW connects the *rcl interface* with a specific DDS vendor, each supported DDS requires a corresponding RMW interface.
This layer converts ROS data objects into DDS-compatible data objects and vice versa.
With this abstraction, users can switch between DDS vendors based on project requirements without needing to learn the DDS API or modify their existing code.

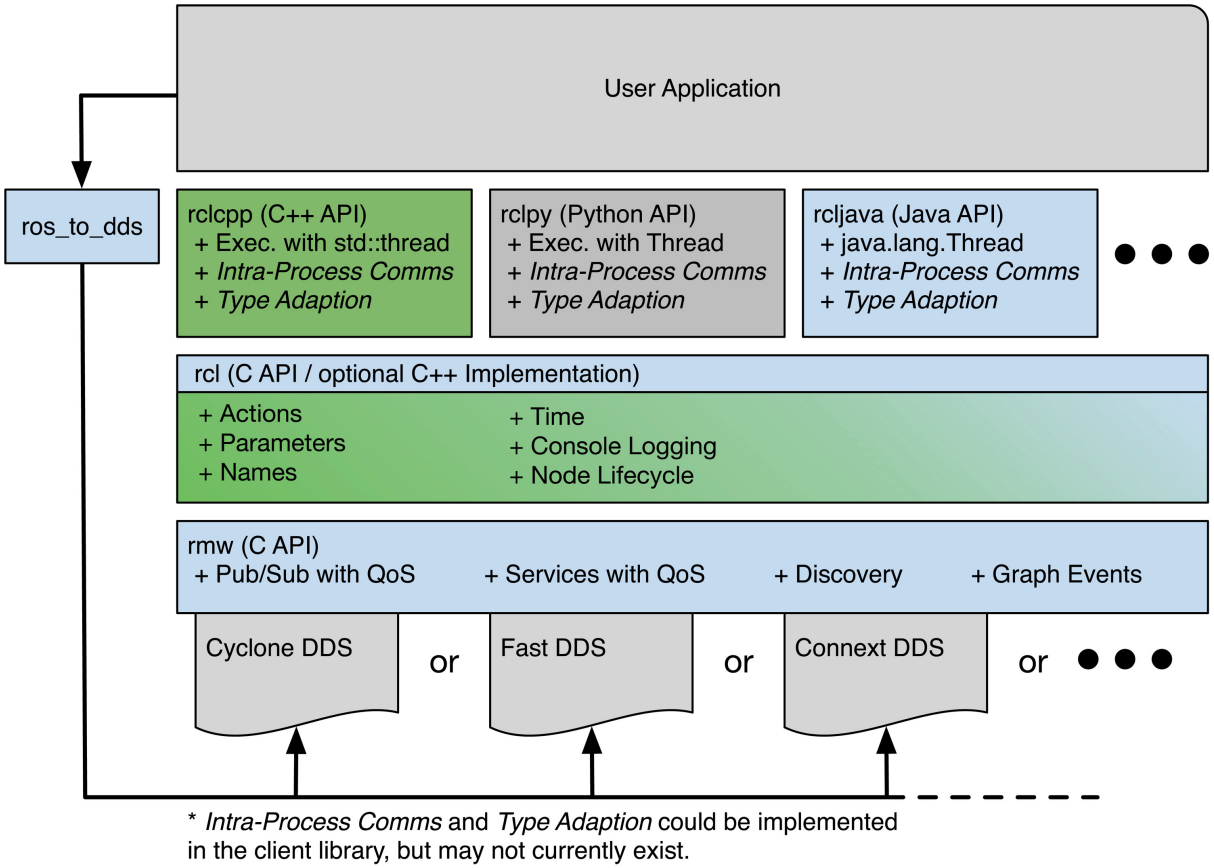


Figure 3.3: ROS 2 Client Library API Stack. Source [32]

Supported RMW implementations in ROS 2 Humble include: *Eclipse Cyclone DDS, eProsima Fast DDS, RTI Connext DDS*, and *GurumNetworks GurumDDS* [39].

Data Distribution Service (DDS)

The Data Distribution Service (DDS) is a middleware protocol and standard developed by the Object Management Group (OMG) for data-centric connectivity, enabling interoperable data exchange between publishers (data producers) and subscribers (data consumers) in a distributed system [40]. The primary features of DDS are:

- **Quality of Service (QoS) Policies**: Controls various aspects of data delivery such as reliability, durability, and latency.
- **Scalability**: Supports large-scale systems with thousands of nodes and millions of data points, ensuring efficient data distribution.
- **Interoperability**: Guarantees interoperability between different DDS implementations, allowing seamless integration of components from various vendors.
- **Real-time performance**: Designed for real-time systems, DDS ensures timely and predictable data delivery, meeting strict timing requirements.

This layer is agnostic to the system's architecture, whether it operates within the same process, across different processes, or on separate machines.

3.2 Nav2

Nav2 [41] is the successor of the ROS Navigation Stack and allows mobile robots to navigate through complex environments by providing functionalities for perception, planning, control, localization, and visualization throughout servers. Nav2 also implements costmaps (*Section 3.2.1*) to represent the environment based on the robot's perception [42]. Navigation pipeline is explained in *Section 3.2.2*.
The input for Nav2 includes the TF tree, which connects the robot to the map's origin, with frames defined as in [43], a Behavior Tree (BT) defined in an XML file, sensor data, and a map if the *Static Layer* of a costmap is implemented.

3.2.1 Costmaps

A costmap is a way of representing the environment based on the robot's perception. The current implementation in Nav2 uses a 2D grid of cells that can be categorized as *unknown*, *free*, *occupied*, or *inflated*.

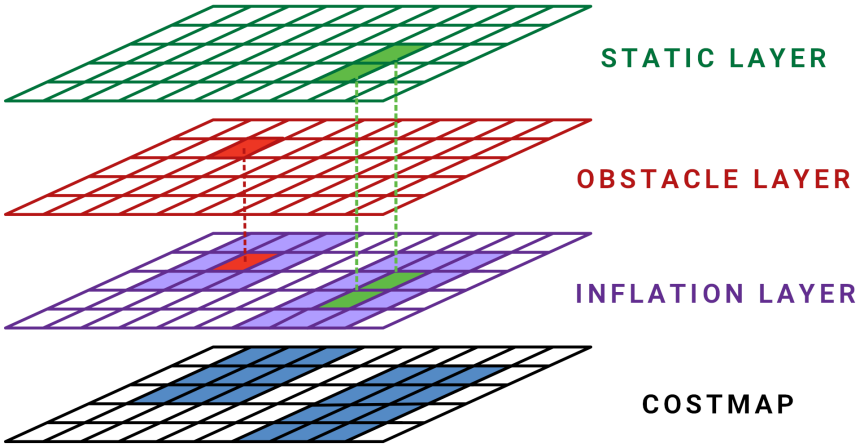


Figure 3.4: Example of costmap and its sublayers

As shown in *Figure 3.4*, a costmap can be composed of multiple layers, each representing different types of costs. The most common layers are:

- **Static Layer:** This layer marks the *occupied* cells in the map provided by the Map Server.
- **Obstacle Layer:** Uses sensor data (such as LIDAR or depth cameras) to detect obstacles in the environment.
- **Inflation Layer:** Inflates the cells next to the those occupied by other layers with exponentially increasing costs.
- **External Plugin Layer:** An additional layer created by external plugins, such as the Spatio-Temporal Voxel Layer.

Spatio-Temporal Voxel Layer (STVL)

The Spatio-Temporal Voxel Layer (STVL) [44] will be briefly explained due to its specific implementation. A voxel represents a value on a regular grid in three-dimensional space. The ”spatial” aspect of the name indicates that the voxel grid represents the environment, while ”temporal” refers to the time decay of voxels, since they disappear after a certain period if no longer detected. This helps generate more realistic costmaps from sensor data by removing dynamic obstacles from the costmap if they are no longer perceived. Furthermore, STVL enables obstacle representation in 3D space, which prevents objects with sufficient clearance from being marked as obstacles, a limitation of the 2D Obstacle Layer.

3.2.2 Nav2 Navigation Pipeline

Nav2's navigation pipeline, as illustrated in *Figure 3.5*, integrates multiple servers that collaboratively ensure reliable navigation for mobile robots in diverse environments. Each server in the pipeline operates as a ROS 2 node, typically managed by a common lifecycle manager to ensure proper initialization, operation, and error handling. The pipeline relies on both static and dynamic environment representations, enabling path planning, execution, and adaptation in real time.

At the heart of the system are two costmaps: the *global costmap* and the *local costmap*. The global costmap provides a comprehensive overview of the environment based on preloaded maps and static obstacles. This map is crucial for the *Planner Server*, which calculates optimal paths using algorithms such as A* or Theta* to avoid known obstacles and navigate efficiently to the target. The local costmap, on the other hand, is updated dynamically with real-time sensor data, allowing the *Controller Server* to react to moving obstacles and compute a valid control effort to follow the global path while ensuring safety in rapidly changing scenarios.
While the local costmap can include information from the preloaded map, this practice is generally discouraged because avoiding known obstacles should be the responsibility of the planner. However, in some cases, obstacles that were not previously known can be added to the global costmap to enable the planner to avoid them during path replanning.

The *BT Navigator Server* acts as the system's decision-making core, utilizing a customizable Behavior Tree (BT) to orchestrate the interactions between the Planner, Controller, and other servers. It determines the next action based on the robot's current state, sensor input, and navigation goals, ensuring adaptability to different tasks.

Additional components enhance the pipeline's capabilities:

- *Behavior Server:* Manages task-specific behaviors, such as obstacle recovery and special maneuvers as docking.
- *Smoother Server:* Refines planned paths to reduce sharp turns, enhancing robot stability and motion fluidity.
- *Map Server:* Handles map storage and retrieval. It also includes a map saver server, which can save the generated map in response to a service request.

These represent some of the available servers in Nav2. Each server offers high customization, enabling users to either implement custom plugins tailored to their specific needs or use existing

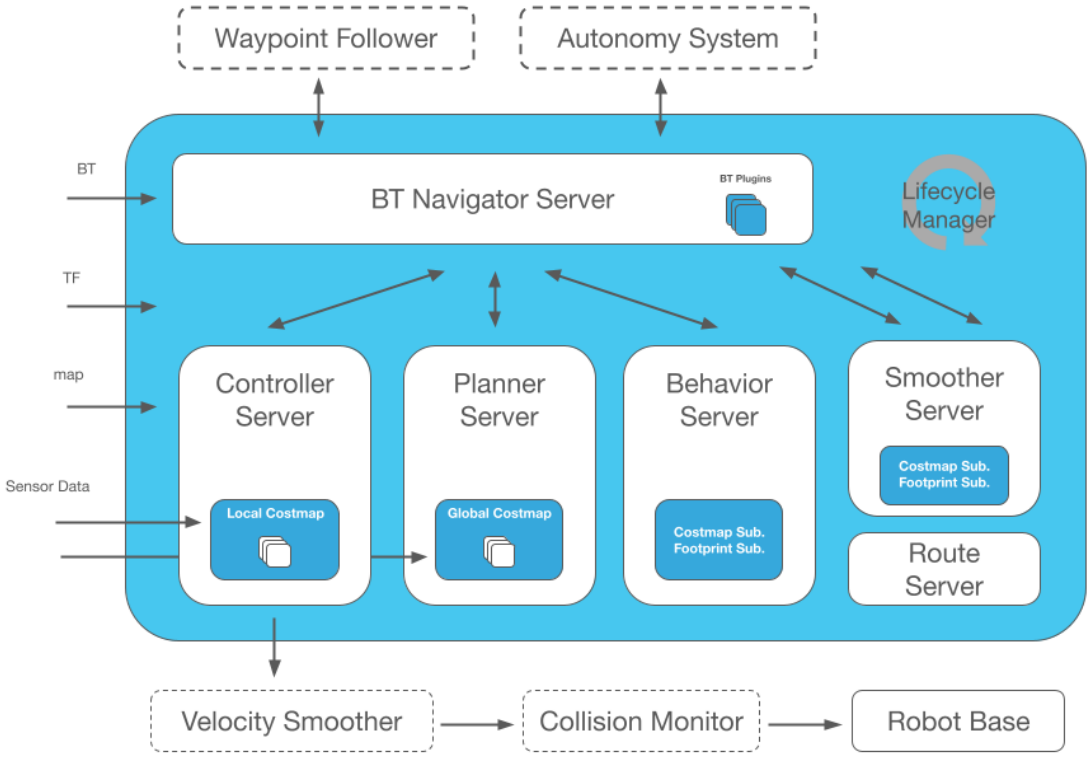


Figure 3.5: Example of Nav 2 Pipeline. Source [42]

plugins from Nav2's extensive library.
Configuration is handled via YAML files, permitting to set parameters for costmap layers, planners, controllers, and additional servers, tailoring Nav2 to specific applications.

3.3 MoveIt 2

MoveIt 2 is the ROS 2 robotic manipulator platform that implements motion planning, manipulation, and kinematics [45]. As visible in *Figure 3.6*, MoveIt 2 is divided into two packages: `moveit_core`, which includes planning scenes, planning interfaces, and collision detection plugins, and `moveit_ros`, which wraps these components within the ROS 2 framework. This second package extends functionality by offering a complete planning pipeline, the Trajectory Execution Manager (TEM), and a variety of user-friendly interfaces, including RViz 2 plugins. In this section, the planning pipeline will be briefly explained (*Section 3.3.1*), followed by the `move_group` node (*Section 3.3.2*), the trajectory execution (*Section 3.3.3*), and finally the Planning Scene Monitor (*Section 3.3.4*).

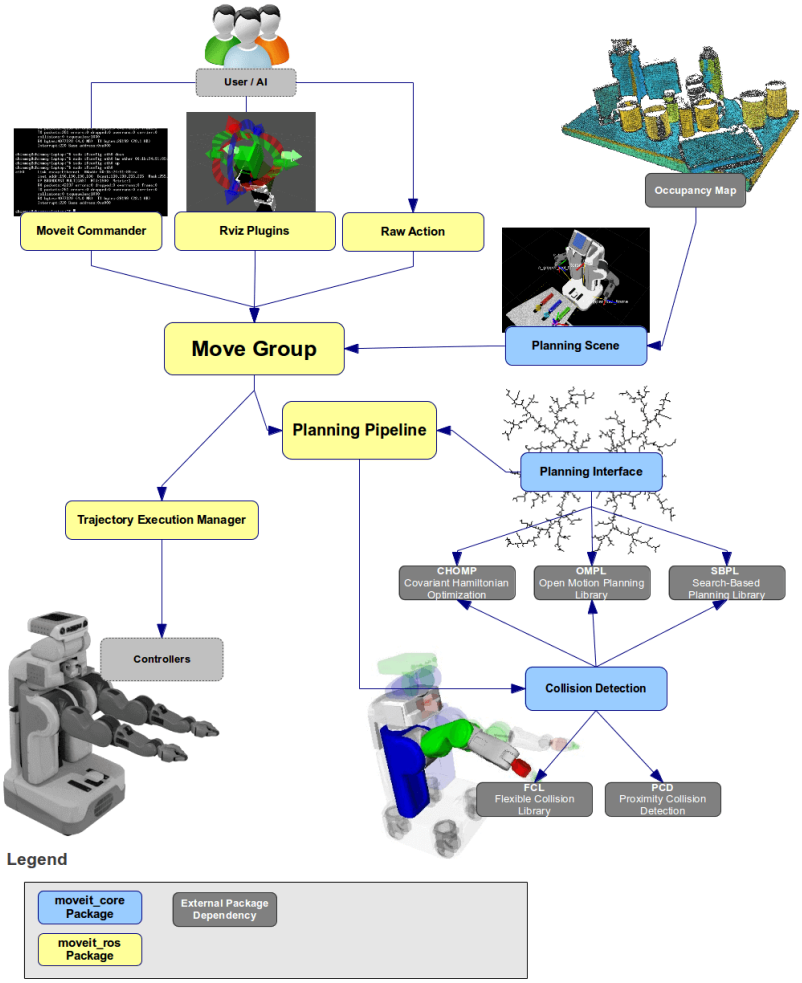


Figure 3.6: MoveIt 2 pipeline. Source [45]

3.3.1 Motion Planning Pipeline

The motion planning pipeline in MoveIt 2 is performed through the `move_group` node (*Section 3.3.2*). This pipeline (*Figure 3.7*) typically consists of five stages:

- **Motion Plan Request:** A request that specifies what the motion planner should calculate, such as moving the arm to a different position or the end-effector to a new pose, is sent to the `move_group` node. Collision checking is performed by default on self-collisions and attached objects. Via the planning pipeline, users can set the planner and add more kinematic constraints such as: Position constraints, orientation constraints, visibility constraints, joint constraints and user-specified constraints.
- **Planning Request Adapters:** These adapters preprocess the request to adjust the start state, workspace bounds, or resolve collisions.
- **Motion Planner:** This stage calls the actual motion planner plugin from `moveit_core`,

such as OMPL [46], Pilz, or CHOMP [47], to generate a feasible trajectory for the robot.

- **Path Post-Processing:** After the planner generates a kinematic path, this stage adds time parameterization and enforces velocity and acceleration limits.
- **Motion Plan Response:** The `move_group` node generates a desired trajectory based on the request and the desired maximum velocities and accelerations. The difference between a trajectory and a path is that the trajectory encapsulates the time information in the planned path.

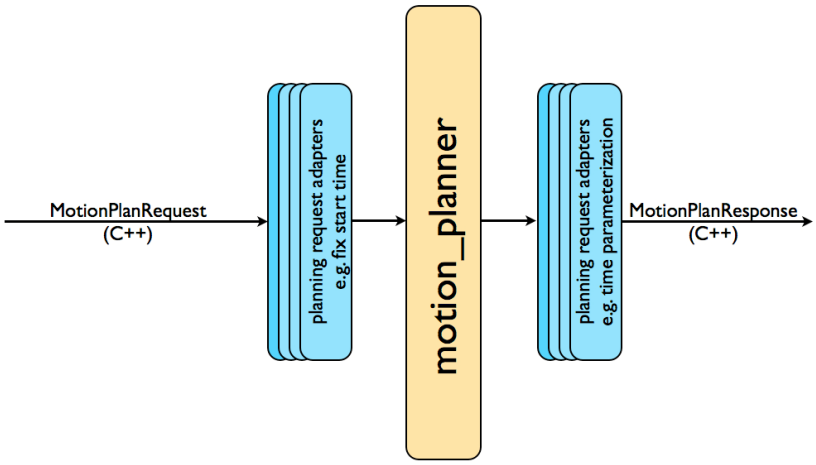


Figure 3.7: Motion planning pipeline. Source [45]

Hybrid Planning

MoveIt 2's default motion planning architecture follows the "Sense-Plan-Act" methodology, which involves a sequence of tasks: perceiving the environment and the robot state, planning the motion, and then executing it. This approach is considered safe only in static environments. In dynamic contexts, however, it is preferable to use "*Hybrid Planning*" provided by MoveIt 2 to enhance safety. This planning strategy is similar to that of Nav 2, as it employs a global planner to compute the trajectory and a local planner to avoid dynamic obstacles and follow the trajectory in real-time.

3.3.2 `move_group` node

The `move_group` node is the ROS 2 node used to integrate all the user interfaces available in MoveIt 2 into a series of ROS services and actions. This node needs three files:

- **URDF:** The description of the robot.

- **SRDF:** The semantic description of the robot, usually generated with MoveIt Setup Assistant.

- **MoveIt configuration:** Specific MoveIt configurations like joint limits, kinematics, motion planning, and perception. The configuration file is usually generated by the MoveIt Setup Assistant.

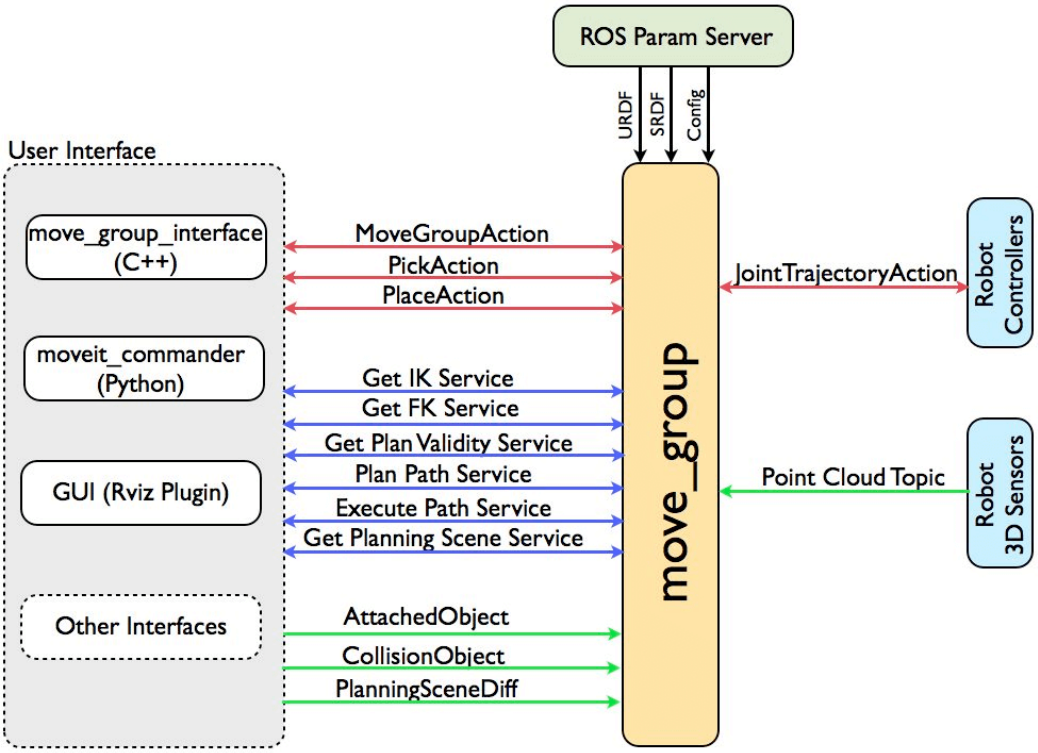


Figure 3.8: `move_group` node. Source [45]

This node obtains the current state information (joint states) from the `joint_state_publisher` node [24] and a TF listener node to locate the robot in the map.

3.3.3 Trajectory Execution

`move_group` node is also responsible of the trajectory execution through the Trajectory Execution Manager (TEM). It interfaces with the robot controllers using the *FollowJointTrajectoryAction* action client interface, which points to a ROS 2 controller action server integrated on the robot side.

3.3.4 Planning Scene Monitor

Planning Scene Monitor is a ROS 2 component that allow to read and write the state of a *Planning Scene* in a thread-safe manner, ensuring that motion planning is always performed in the most accurate representation of the robot's surroundings.
The *Planning Scene* is an object that represents the environment around the robot and the state of the robot itself, including attached objects.

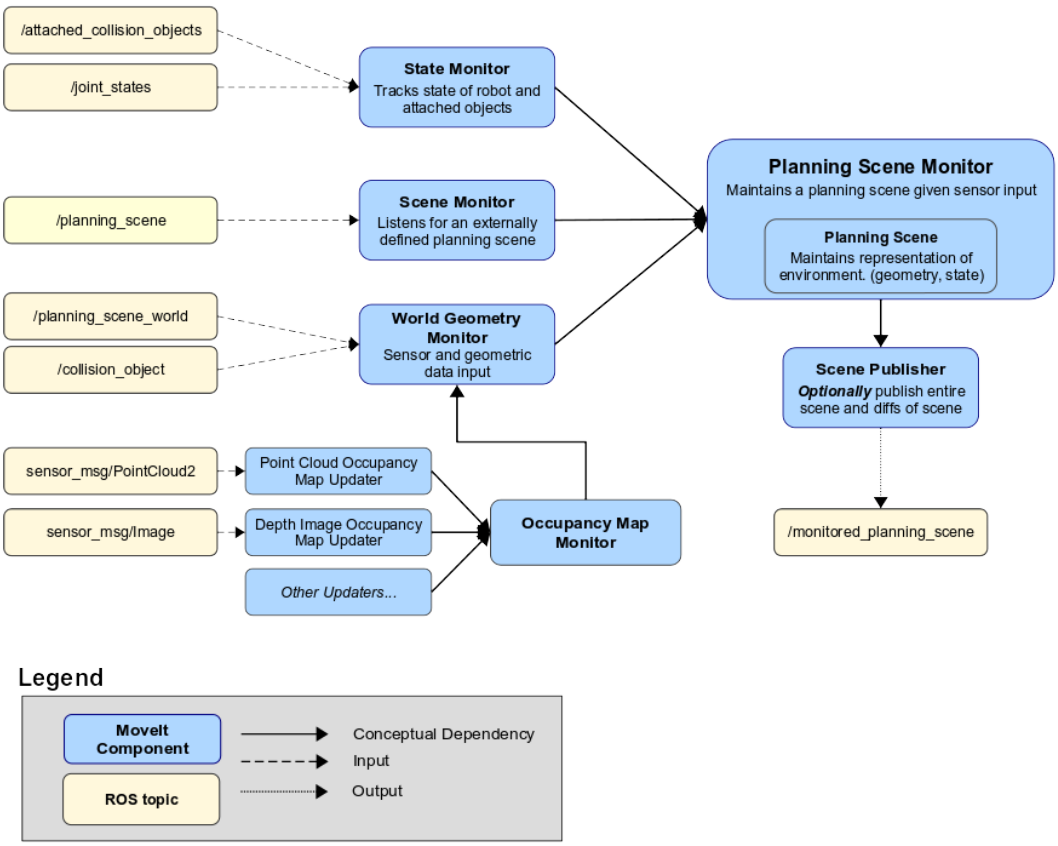


Figure 3.9: Planning Scene Monitor architecture. Source [45]

The Planning Scene Monitor gathers information from other components of MoveIt 2:

- *World Geometry Monitor:* Builds the world geometry using data from sensors on the robot and user inputs. It uses an occupancy map monitor to create a 3D representation of the environment, typically using OctoMap [48].
- *State Monitor:* Tracks the state of the robot's joints and attached objects in the environment to avoid collisions. The joint states are updated based on the data from the `joint_state_publisher`.
- *Scene Monitor:* Listens for updates to the external planning scene and merges them with the robot's internal state, ensuring a coherent environment model.

3.4 Fiducial markers for pose estimation

To adhere to the goals of this thesis, the most cost-effective approach has been selected to localize both human and LoCoBot in the environment. Despite the decreasing cost of LIDAR, only pre-installed sensors on the LoCoBot will be used, leading to the choice of using computer vision techniques for visual odometry using fiducial markers. This technology requires only an RGB camera to detect markers, and the analysis can be performed even on a relatively low-power computer.

Although LoCoBot includes an onboard camera, external cameras have been chosen to track both the LoCoBot and the human it follows. This choice is driven by two important factors: first, relying on the onboard camera would require a large number of markers throughout the environment, and sensor fusion would become essential for estimating the robot's pose whenever markers are not visible. Second, if an obstacle temporarily blocks the robot's view, it could lose sight of the human, disrupting its tracking capability.

Using a network of external cameras addresses both of these issues. With external cameras, only two markers are necessary, one for the human and one for the LoCoBot. In addition, complete coverage of the environment can be achieved. This latter hypothesis is assumed in this thesis, considering that both markers are always visible throughout the strategic positioning of the cameras.
Furthermore, the external cameras can be connected to external computers to estimate the pose, while the one on the LoCoBot focuses of navigation and interaction tasks.
Moreover, the same hypothesis allows to not consider the odometry of the wheels and the consequent sensor fusion needed to track a robot.

In this section, fiducial markers' technology will be explained (*Section 3.4.1*), including its fundamental concepts and the chosen coordinate systems will be described (*Section 3.4.2*). The ROS 2 package used for detecting these markers will be discussed in *Section 3.4.3*. Finally, the resulting TF tree that is published to link each detected marker to the map origin will be presented (*Section 3.4.4*), providing a comprehensive overview of how these markers contribute to the overall navigation and localization framework.

3.4.1 AprilTag 3

A fiducial marker is a recognizable pattern in an image that enables the calculation of the relative pose between the camera capturing the image and the marker itself. Commonly used

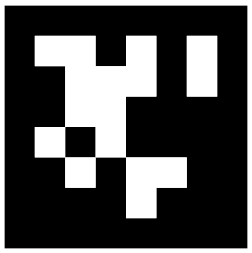


Figure 3.10: An example of AprilTag 3 marker of the tag36_11 family

in augmented reality, fiducial markers vary in design but generally consist of an image with a predefined size divided into pixels, each arrangement encoding a unique identifier.
The known dimensions of these markers enable precise pose estimation relative to the camera. For mapping and localization, setting either the camera or a marker as a static reference point for the map origin allows the position of other markers to be dynamically calculated within this reference.

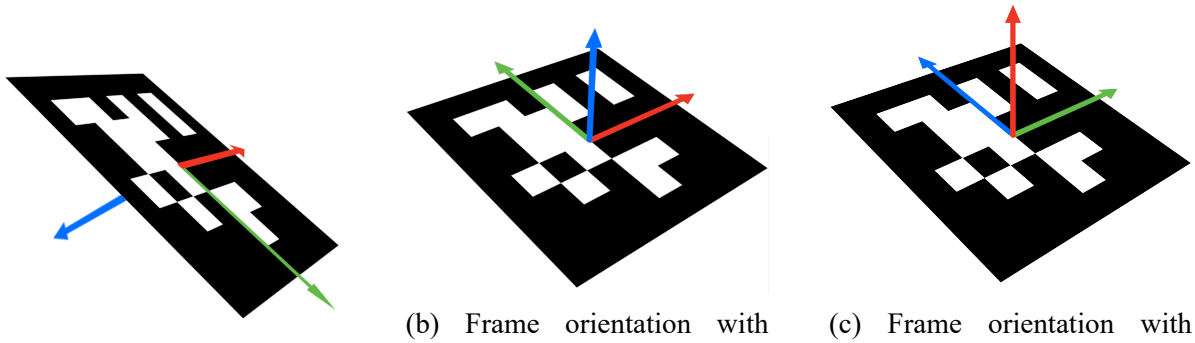
A previous study [49] comparing various fiducial markers demonstrated that AprilTag 3 [50] performed better than other popular markers, such as ArUco [51] and Stag [52]. Consequently, the AprilTag 3 family (*Figure 3.10*) was selected to localize both the LoCoBot and humans in this project. The chosen tag family, *tag36_11*, is widely implemented and compatible with ArUco detectors, such as OpenCV.

3.4.2 AprilTag 3 Coordinate System

The pose of the apriltag is determined by assigning the coordinate system with the origin at the camera center. The z-axis points from the camera center out the camera lens. The x-axis is to the right in the image taken by the camera, and y-axis is down. The tag's coordinate frame is centered at the center of the tag, with x-axis to the right, y-axis down, and z-axis into the tag.

3.4.3 AprilTag ROS

AprilTag ROS [53] is a ROS 2 wrapper for the AprilTag detector. This package includes a node that detects markers in a video stream and publishes the transforms from the camera frame to each detected marker's frame on the *tf* topic. In addition, a list of detected marker poses is published in a separate topic.
The configuration file allows the user to specify which tag IDs to detect and assign each a unique frame name. Other detector parameters, such as the tag family, can also be set in this file.
A key parameter, `z_up`, changes the orientation of the detected frame. If set to *false*, the frame



(a) Original frame orientation

(b) Frame orientation with `z_up` = *false*

(c) Frame orientation with `z_up` = *true*

Figure 3.11: AprilTag coordinate systems

matches the orientation in *Figure 3.11b*; otherwise, it matches the orientation in *Figure 3.11c*. The configuration implemented in this thesis is the one in *Figure 3.11b*. This configuration was chosen because the navigation map's z-axis points upward from the ground. As a result, the z-axis direction of a marker on the LoCoBot is aligned with the map.

3.4.4 TF Tree

By publishing a static TF that links the LoCoBot's fiducial marker to its base frame (i.e., *locobot/base_footprint*), the TF tree generated by the AprilTag node integrates with the one published by the `joint_state_publisher` and the `robot_state_publisher` of the LoCoBot.
As the *base_footprint* frame is the ground-projected center of the Kobuki base, by positioning the marker over the LoCoBot, only a z-axis translation between the ground and marker is required. A rotation on the z-axis may also be needed, depending on the position of the marker, as the LoCoBot x-axis should be facing towards the direction of movement, consistent with navigation standards.
In *Figure 3.12*, the TF tree generated by the AprilTag node and cameras is shown. The *locobot/base_footprint* node has an attached TF subtree generated by the LoCoBot's state publishers, not inserted in this image.

3.5 Camera Network

To track the positions of the human and the LoCoBot within the environment, a network of Azure Kinect DK cameras was installed. The Kinect comes with a ROS wrapper [54] responsible for configuring the camera, publishing TFs for camera frames, and streaming images and point clouds from the depth sensor. At the time of writing, Azure Kinect DK is discontinued, and the wrapper supports only the Humble release of ROS 2.

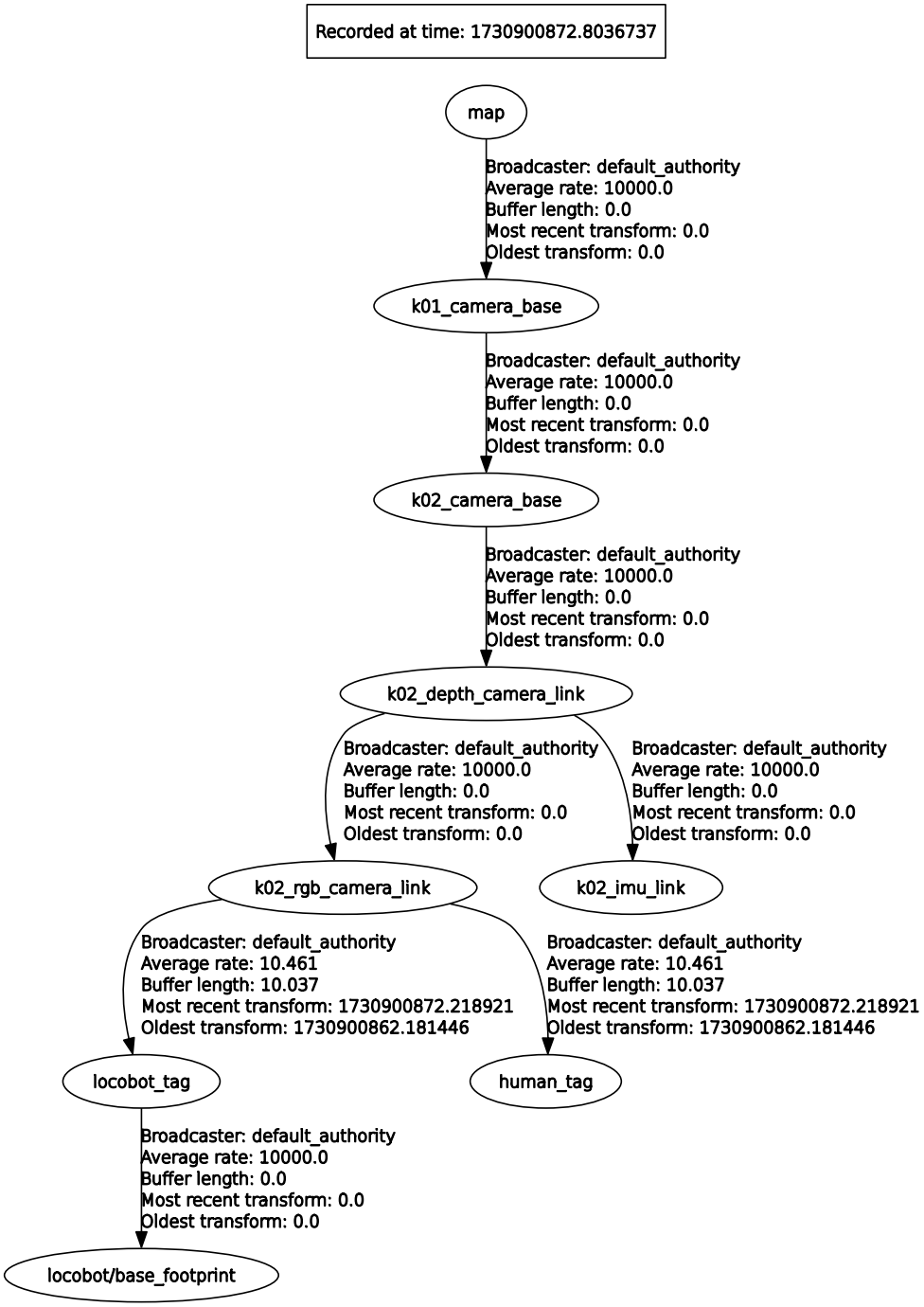


Figure 3.12: TF Tree of the camera and markers for the human and the LoCoBot

As point clouds are unnecessary for this application, each camera only publishes RGB images. In this thesis, a custom package [55] was utilized to address various issues found in the original wrapper. The section is structured as follows: first, the camera pose calibration method used is described in (*Section 3.5.1*). Next, the issue of image rectification is highlighted (*Section 3.5.2*), along with the challenge of network bandwidth usage when transmitting images (*Section 3.5.3*).

3.5.1 Camera Pose Calibration

A separate package [56] was used to determine the position of each camera on the map. This package is invoked by several launch files:

- **k01calib.launch.py**: Launches the Azure Kinect node to publish the RGB and depth streams for the camera named "K01", applies image rectification, and starts the AprilTag detector.
- **k02calib.launch.py**: Similar to *k01calib.launch.py* but for the camera "K02".
- **calib_master.launch.py**: Starts a node to perform multi-camera calibration.

The calibration method involves placing several AprilTag markers on the ground, with at least one visible to multiple cameras. By triggering the `start_calibration` and `stop_calibration` services, TFs from each camera to each visible marker are recorded to calculate the map frame's location. The marker used as center of the map can be specified in the parameters file, along with the map frame name. To improve accuracy during calibration, several factors were taken into account:

- *Calibration Time:* The calibration duration was set to 10 seconds to ensure sufficient data for accurate calibration.
- *Standard Deviation on the TF:* If the pose of a marker is detected with a standard deviation greater than the selected threshold, that marker is excluded from the calibration process.
- *Mean Position of the Camera:* The pose of the camera is calculated as the mean of the poses estimated by each marker.

The same calibration can be performed also with only one camera.

After calibration, a static TF from one camera to the map frame is published, and all cameras form a daisy-chained TF tree anchored at the first camera.
This setup provides a robust framework in which one camera can track the LoCoBot while another observes the human, or both cameras may observe the same subjects.

The speed of this process allows for daily recalibration in laboratory environments, while in real scenarios, cameras would be mounted permanently.

Efficiency of this technique

When two cameras detect the same marker and publish TFs for the same child frame, the resulting TF in the tree is defined by the most recently published one. This can lead to significant flickering in the estimated marker position, introducing errors that go beyond detection inaccuracies.
These discrepancies arise from imperfect calibration of the camera positions, causing each camera to estimate the marker position slightly differently.
To quantify this error, an empirical approach was adopted. Two cameras were placed in different locations, ensuring both could detect the LoCoBot marker within their field of view. Transform data from the *map* frame to the *locobot/base_footprint* frame was recorded every 200 ms over 5 seconds.

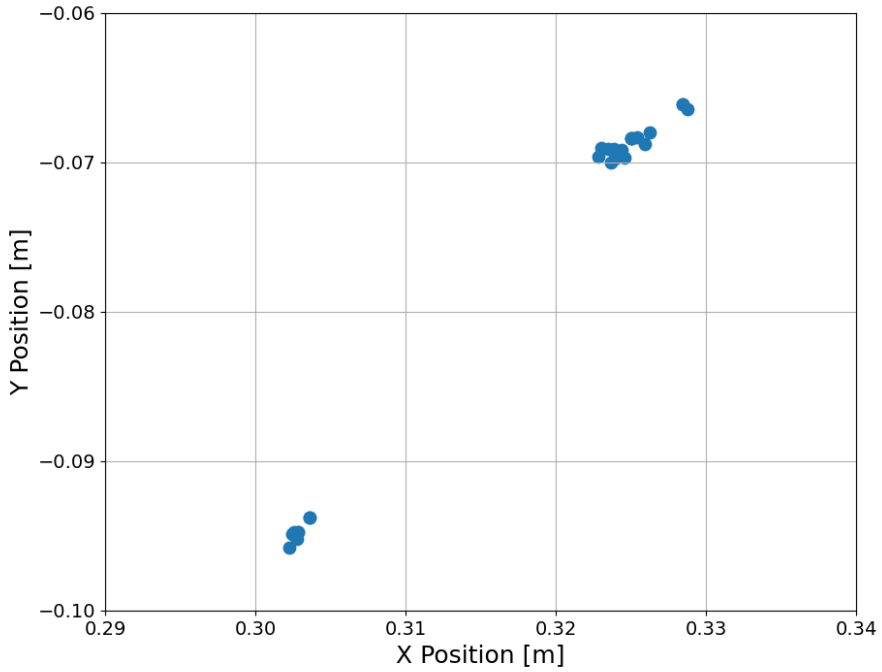


Figure 3.13: Estimated position over 5 seconds with two Kinect cameras

As illustrated in *Figure 3.13*, the data forms two distinct clusters, each representing the pose estimations of the marker in the map from one camera. While each camera provides consistent

estimates within its cluster, clusters positioning reveals the impact of calibration imperfections. The analysis shows that the positional difference between the clusters is less than 5 cm, which is deemed acceptable for the purposes of this work.

3.5.2 Image Rectification

Each camera sensor captures an image that is not planar, as the image is distorted by the lens. Thus, each camera publishes a *camera_info* topic with parameters needed for image rectification. To rectify the image, the `Image_proc` [57] package was used.

3.5.3 Bandwidth Usage

Due to the nature of the DDS protocol, all data in ROS 2 are broadcasted across the network. Using multiple cameras with Full HD video streaming at 30 fps requires considerable network bandwidth, since Kinect camera streaming at this resolution and frame rate can demand close to 1 Gbit/s. With multiple cameras, this would quickly overwhelm the network bandwidth and cause inefficient data transfer between the LoCoBot and the remote computer responsible for calculating the robot's position on the map.
To mitigate this issue, the nodes responsible for Apriltag pose estimation were loaded in the same process as the image rectification node (*Section 3.1.1*).

3.6 Conclusions

The explanation of ROS 2's framework, covering Topics, Services, and Actions, offered a clear understanding of node interactions within the ROS graph. Furthermore, the in-depth look at Nav2's navigation pipeline and MoveIt 2's planning interface established the basis for comprehending the customization and decisions made in the framework's development.
Moreover, this chapter emphasized a computer vision technique using fiducial markers for estimating object positions in space, a core element of the framework. Essential features were detailed to ensure proper camera position calibration and to preserve network bandwidth.
In sum, this chapter set the stage for further framework development and adaptation, confirming that all critical concepts and methods are comprehended and validated.

Chapter 4

Human-guided AMM framework

Figure 4.1 presents the components of the proposed framework discussed in this chapter. Initially, this chapter provides an introduction to the gesture recognition module (*Section 4.1*), a core component of the framework that enables human control of the robot. Human-robot interaction can be achieved through various communication methods, with modern approaches favoring user-friendly techniques such as speech and gesture recognition over traditional interfaces like keyboards.
Speech recognition typically employs large language models (LLMs) to interpret spoken commands, whereas gesture recognition uses image-based machine learning to identify hand gestures. While speech recognition offers greater flexibility for users, gesture recognition was chosen for this framework due to its lower computational requirements, the availability of an onboard camera, and the absence of a microphone.

Besides gesture recognition, this chapter delves into the development process of the state machine, emphasizing its role as the control node for the LoCoBot. As outlined in *Section 4.2*, the state machine is not merely a decision-making mechanism but also directly handles control tasks. By integrating the control logic within the state machine, human command inputs are efficiently processed and smoothly transformed into executable tasks for the navigation and interaction modules. A comprehensive description of the state machine's architecture, functionality, and individual states is presented in *Section 4.3*.
In addition, the customizations made to Nav2 and MoveIt 2 will be discussed, as these modules are directly linked to the state machine. The integration of Nav2 and MoveIt 2 is elaborated in *Sections 4.4* and *4.5*, respectively.
Finally, *Section 4.6* presents a simulation environment designed to evaluate the functionality of the state machine and the integration of all framework modules.

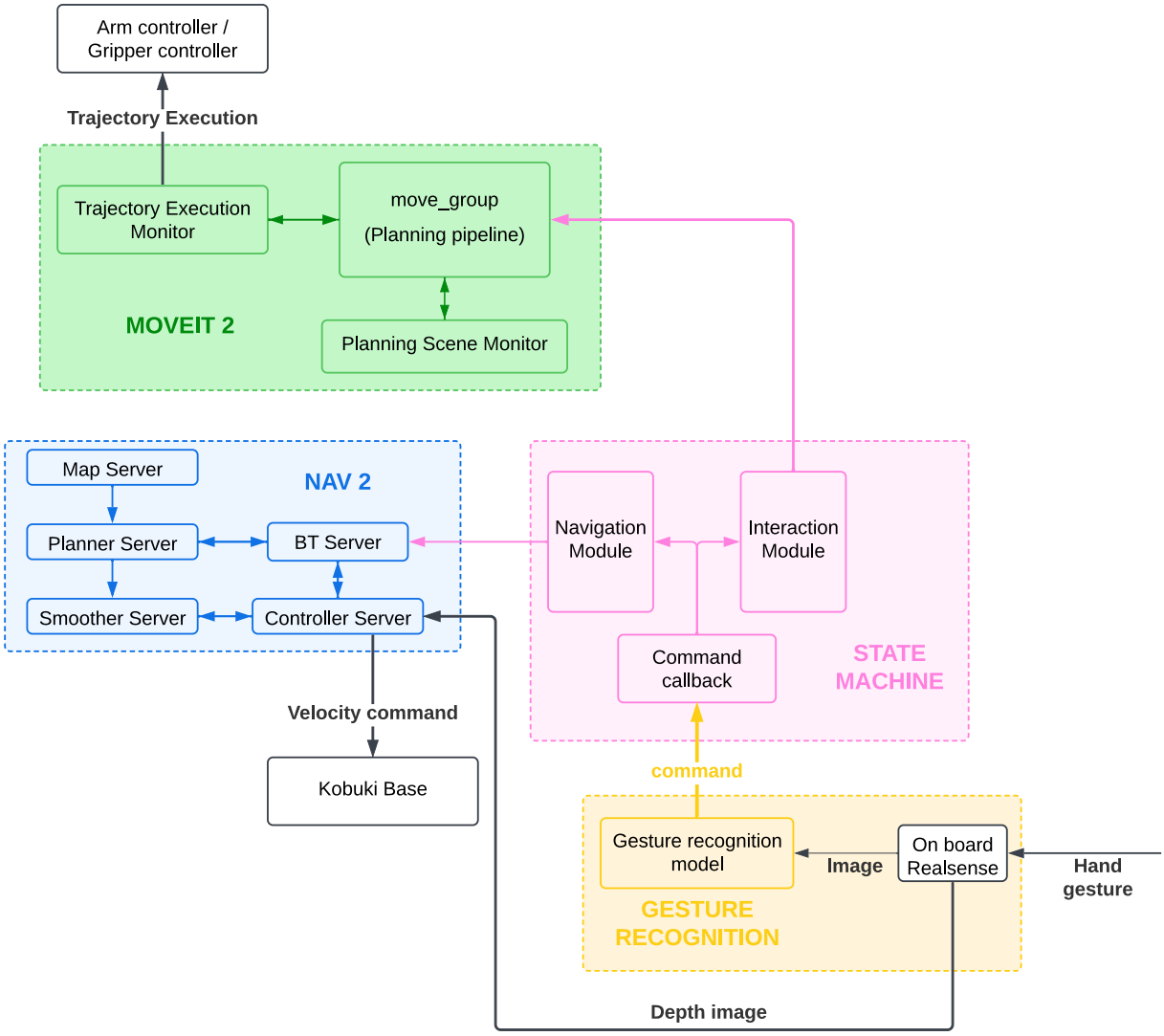


Figure 4.1: Overview of the framework components discussed in Chapter 4

The proposed framework can be found at: `https://github.com/Alessio-Lovato-Unipd/locobot_ws/tree/main`.

4.1 Gesture Recognition

The *MediaPipe* framework [58] was selected as gesture recognition model due to its real-time performance and low computational demands. This framework processes frames (or video streams) using a Convolutional Neural Network (CNN), extracting hand skeletons to recognize gestures.

The gesture recognition process was implemented as a ROS 2 node. The node subscribes to the onboard RealSense™ camera's image topic and interacts with the state machine via the *ControlStates* service provided by the `simulation_interfaces` package.

This node maps recognized gestures to specific state machine commands as outlined in *Table 4.1*.

Gesture	Command
Thumb_Up	NAVIGATION
Thumb_Down	INTERACTION
Open_Palm	IDLE STATE
Pointing_Up	OPEN_GRIPPER
Closed_Fist	CLOSE_GRIPPER
ILoveYou	ABORT

Table 4.1: Gesture to command mapping

The available parameters for the gesture recognition node are summarized in *Table 4.2*.

Parameter	Default Value	Description
`camera_topic`	`/camera/image_raw`	Topic for camera images.
`service_name`	–	Name of the service to interact with.
`minimum_score`	`0.6`	Minimum confidence score for valid gestures.

Table 4.2: Gesture recognition node parameters

4.1.1 Usability improvement

To improve usability, each image frame from the camera was flipped horizontally before being processed by the model. This adjustment allowed gestures to be performed more naturally without requiring users to kneel or adjust their position. However, this approach introduced some ambiguities, such as reversing the *Thumb_Up* and *Thumb_Down* gestures. Other gestures needed to be performed while pointing towards the ground.

4.2 LoCoBot Control Node

To simplify interactions with the underlying implementations of Nav 2 and MoveIt 2, all LoCoBot controls are grouped within a ROS 2 node named *LocobotControl*. This node is developed as a C++ class inheriting from the `rclcpp::Node` class.
The *LocobotControl* node encapsulates functions for navigation and interaction control while also providing real-time feedback on execution status. Two auxiliary classes, *ArmStatus* and *NavigationStatus*, are integrated within this node to monitor the status of each subsystem.
Table 4.3 lists the configurable parameters for this node.

Parameter	Default value	Description
`navigation_server`	`navigate_to_pose`	Name of the navigation server
`arm_interface`	`interbotix_arm`	Name of the arm interface in the SRDF
`gripper_interface`	`interbotix_gripper`	Name of the gripper interface in the SRDF
`timeout`	`2.0`	Timeout [s] for navigation server availability

Table 4.3: LocobotControl node parameters

4.2.1 ArmStatus Class

As its name implies, this class monitors the arm's status. It stores the last known poses of both the arm and the gripper, as well as flags for error and motion states.
The `updateStatus()` function is used to refresh the arm's status, and other member functions provide access to its pose, error flags, and motion state.
Additionally, predefined arm and gripper poses, specified in the SRDF, are stored in enumeration classes (*Table 4.4*). These classes also establish a pose that represents an indefinite state.

Enumeration Class	ArmPose	GripperState
Pose	`HOME`	`HOME`
	`SLEEP`	`RELEASED`
	`UPRIGHT`	`GRASPING`
	`UNKNOWN`	`UNKNOWN`

Table 4.4: Enumeration classes storing default poses of the robotic arm

4.2.2 NavigationStatus Class

The *NavigationStatus* class monitors the outcomes of navigation commands. It tracks the success, failure, or in-progress status of navigation and gathers feedback from the navigation server, such as estimated time of arrival and remaining distance. These details are accessible via member functions.

4.2.3 LocobotControl Class

This class implements the actual ROS 2 node. The constructor retrieves parameters, initializes the client for the navigation server, and creates an *ArmStatus* instance to store the arm's initial pose. A *NavigationStatus* instance is also initialized.
The destructor ensures that any ongoing motion is stopped before destroying the node.
Wrapper functions are provided to expose internal status information. Furthermore, the following public methods are implemented to control the LoCoBot:

- `void MoveBaseTo(const geometry_msgs::msg::PoseStamped &pose, std::optional<double> timeout = std::nullopt)`
  - **Return Value**: `void`
  - **Parameters**:

    `pose`: Target pose relative to the map frame.

    `timeout`: Timeout for server availability (default is node parameter value).
  - **Description**: Sends a goal to the navigation stack to move the robot to the specified pose.
- `bool SetArmPose(const ArmPose pose)`
  - **Return Value**: `bool`
  - **Parameters**:

    `pose`: Target arm pose.
  - **Description**: Moves the arm to a predefined pose.
- `bool SetGripper(const GripperState state)`
  - **Return Value**: `bool`
  - **Parameters**:

    `state`: Target gripper state.
  - **Description**: Moves the gripper to the specified state.
- `void StopArm()`
  - **Return Value**: `void`
  - **Description**: Stops ongoing arm or gripper motions.

- `bool cancelNavigationGoal()`
  - **Return Value**: `bool`
  - **Description**: Cancels all goals sent to the navigation server.

4.3 State Machine Node

To achieve high-level control over the LoCoBot, a state machine was used, with transitions between states triggered by the gesture recognition node. The state machine is implemented using a ROS 2 node that inherits from the *LocobotControl* class, making all the functions to control the LoCoBot available.
In addition, service servers and topic publishers are created to enable interaction with other nodes. The parameters configurable for the node are listed in *Table 4.5*.

Parameter	Default value	Description
`robot_tag_frame`	`locobot_tag`	Frame of the robot's tag
`map_frame`	`map`	Frame of the map (map origin)
`human_tag_frame`	`human_tag`	Frame of the human's tag
`follow_human`	`true`	Specifies whether the robot follows the human
`goal_update_topic`	`goal_update`	Topic to update the navigation goal
`state_topic`	`machine_state`	Topic where the current state is published
`sleep_time`	`100`	Sleep duration [ms] between state machine cycles
`tf_tolerance`	`2.0`	Time [s] to tolerate TF missing before error
`debug`	`false`	Publish internal state

Table 4.5: State Machine node parameters

The state machine defines two types of enumerated states: *Internal states* and *External states*. *Internal states* represent the machine's operation and transition logic, while *External states* provide high-level feedback similar to a ROS 2 action interface. These states are summarized in *Table 4.6*.
The constructor initializes the TF listener buffer and creates the *LastError*, *ClearError*, and *ControlStates* service servers. All these services are available in the `simulation_interfaces` package. Additionally, publishers are set up to broadcast the current state and update navigation goals. Finally, a thread is launched to execute the state machine's loop function, `SpinMachine()`, which ensures continuous operation. This thread is joined during node destruction to ensure proper resource cleanup.

Type	State	Description
Internal	IDLE	Waits for a new command
	SECURE_ARM	Ensures the arm is in safe position for navigation
	WAIT_ARM_SECURING	Waits for the arm to secure
	SEND_NAV_GOAL	Sends the navigation goal to the navigation stack
	WAIT_NAVIGATION	Waits for the robot to reach the navigation goal
	WAIT_ARM_EXTENDING	Waits for the arm to extend to the target position
	ARM_EXTENDED	Arm extended, allowing gripper usage
	WAIT_GRIPPER	Waits for the gripper to complete its movement
	WAIT_ARM_RETRACTING	Waits for the arm to retract
	ERROR	Indicates an error state
	ABORT	Aborts the current task
	STOPPING	Stops the state machine
External	SUCCESS	Task completed successfully
	FAILURE	Task failed to complete
	RUNNING	State machine is currently active
	INITIALIZED	State machine has been initialized

Table 4.6: State Machine states and descriptions

4.3.1 LastError Service

This service retrieves the last error encountered by the state machine and its current state. If no errors are present, the error message will be empty.

4.3.2 ClearError Service

The *ClearError* service invokes the `clearError()` function. It returns *false* if called when the machine is not in the `ERROR` or `ABORT` state. Otherwise, it resets user requests, clears errors in the *NavigationStatus* and *ArmStatus* classes, sets the machine to the `IDLE` state, and returns *true*.

4.3.3 ControlStates Service

This service interfaces with the gesture recognition node. Based on the received request mapped by the *Commands* (*Table 4.1*), the state machine modifies internal variables that govern state transitions.

Direct state changes via this service are avoided to maintain encapsulation. Instead, auxiliary variables such as `requestedInteraction_`, `requestedNavigation_`, `requestedAbort_` and `requestedGripperMovement_` are used. These requests are processed in the state machine loop, and mutexes ensure thread safety.

Navigation and interaction requests are accepted only when the machine is in the `IDLE` state,

while gripper commands are valid only in the ARM_EXTENDED state. These constraints prevent unintended transitions during other states.

4.3.4 SpinMachine() Function

This function operates the state machine in an infinite loop and invokes the nextState() function to update the machine's state. The loop terminates upon node destruction or when the ROS 2 context is shutdown. The current external state is published on every cycle, and the internal state is optionally logged if the debug parameter is enabled.

4.3.5 Internal States

Internal states represent the robot's functioning and are grouped into three categories: *Generic*, *Navigation Module*, and *Interaction Module*. Transitions between these states are depicted in *Figure 4.2*. A transition occurs automatically in the absence of any attached conditions.
The machine starts in ERROR state for safety, requiring user intervention to proceed. From any state, the machine can transition to ABORT upon user request or to ERROR in case of critical failures, such as missing TF of human or LoCoBot.
Each state executes specific tasks to manage robot operations, as described in the following sections:

Generic

Generic group covers states that, while not specialized, are essential for operation.

- **IDLE**: In this state, the LoCoBot waits for commands to start navigation or interaction. If navigation is triggered, the next state is SECURE_ARM. Otherwise, the arm is commanded to go to the *Home* position, and the next state is WAIT_ARM_EXTENDING.

- **STOPPING**: This state sends commands to stop navigation and arm movements. The next state is ERROR.

- **ERROR**: This state indicates that an error occurred. It does not perform any action. To exit this state, the *ClearError* service must be used, which clears all errors and returns the state machine to the IDLE state.

- **ABORT**: This state is triggered by a request from the human. It stops arm movement and navigation, clears all errors, and returns the state machine to the IDLE state.

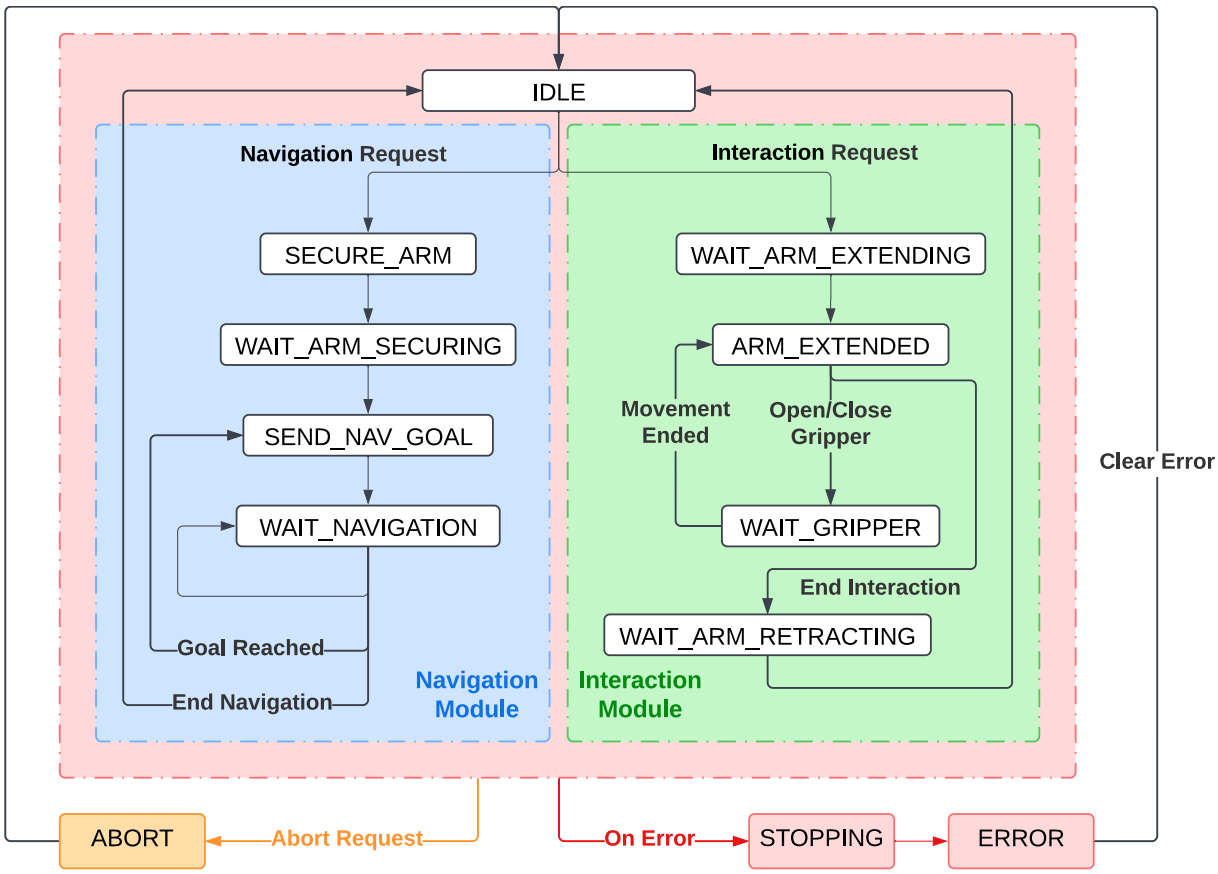


Figure 4.2: Internal States diagram

Navigation Module

The *Navigation Module* contains all the states dedicated to executing navigation tasks.

- **SECURE_ARM**: This state send the arm in a safe position for navigation (*Sleep* position). The next state is `WAIT_ARM_SECURING`.

- **WAIT_ARM_SECURING**: If the arm is moving, this state waits for its completion. If an error occurs, the next state is `STOPPING`. Otherwise, the next state is `SEND_NAV_GOAL`.

- **SEND_NAV_GOAL**: In this state, the pose of the human relative to the map frame is obtained and sent as a goal to the navigation stack. The state then transitions to `WAIT_NAVIGATION`.

- **WAIT_NAVIGATION**: If the `follow_human` parameter is set to *true*, the position of the navigation goal is updated every time this state is triggered.
  This state loops until navigation is stopped either by the human or due to an error. In the first case, the next state is `IDLE`. In the latter case, the next state is `STOPPING`. If navigation ends without either of those events, the next state returns to `IDLE` if `follow_human` is *false*, otherwise `SEND_NAV_GOAL`.

Interaction Module

The *Interaction Module* includes states related to manipulation tasks.

- **WAIT_ARM_EXTENDING**: This state waits for the arm to reach the *Home* position. If an error occurs, the next state is `STOPPING`. Otherwise, it transitions to `ARM_EXTENDED`.

- **ARM_EXTENDED**: In this state, commands to move the gripper can be sent. If a command is issued, the next state is `WAIT_GRIPPER`. It is also possible to return to the `IDLE` state, in which case the next state is `WAIT_ARM_RETRACTING`.

- **WAIT_GRIPPER**: This state waits for the gripper to complete its movement. If the movement succeeds, the next state is `ARM_EXTENDED`. If the movement fails, the next state is `STOPPING`.

- **WAIT_ARM_RETRACTING**: This state waits for the arm to reach the *Sleep* position. Once reached, the next state is `IDLE`. If an error occurs, the next state is `STOPPING`.

4.3.6 External States

External states provide feedback about the machine's overall progress. The default state at startup is `INITIALIZED`. When internal states `SECURE_ARM` or `WAIT_ARM_EXTENDING` are reached, the external state transitions to `RUNNING`. Errors trigger the `FAILURE` state, and upon clear request, the state resets to `INITIALIZED`. Lastly, the external state `SUCCESS` is set when the destructor of the state machine is triggered.

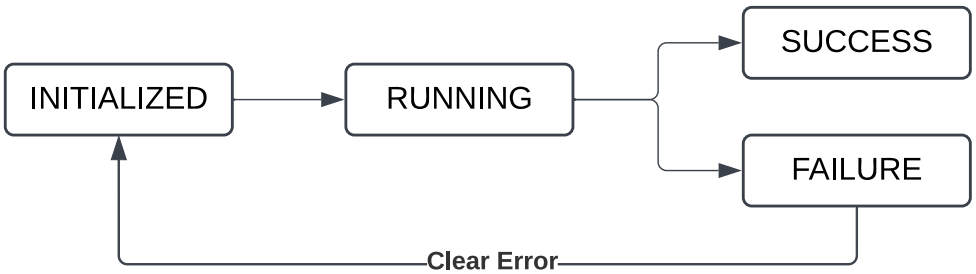


Figure 4.3: External states diagram

4.4 Nav2 configuration

The navigation stack tuning was the second most significant task in this thesis due to the vast number of customizable plugins offered by Nav2. Each "server" in the navigation system is highly configurable and can be selectively implemented. Since achieving perfect navigation was not within the scope of this thesis, the implementation used only standard plugins without custom development. Moreover, only the essential servers were utilized and tuned for the specific application. The selected servers were the *Map Server*, *Behavior Server*, *Controller Server*, *Planner Server*, *BT Navigator*, and *Velocity Smoother*. This section details the configurations and choices made for these servers.
A common parameter across all nodes was `use_sim_time`, which was set to *true* during simulations and *false* on the physical robot.

4.4.1 Map Server

The primary parameter for this node was the file path to the map to be loaded. The macro `$(find-pkg-share <package-name>)` was used to locate files within specific packages, providing flexibility and portability.

4.4.2 Map Saver

The map saver server was not implemented, as the environment was considered static with few static and dynamic obstacles that did not require mapping. Instead, navigation maps were generated by converting the simulation world files (*Section 4.6.1*) into the Nav2 map format using [59].

4.4.3 Behavior Server

This server was configured as recommended in [42], with the `local_frame` and `robot_base_frame` parameters set to *locobot/base_footprint*.
The server loaded the following plugins:

- *spin*
- *backup*
- *drive_on_heading*
- *wait*
- *assisted_teleop*

This configuration ensured compatibility with the default *BT Navigator*.

4.4.4 Controller Server

The primary goal of the controller server was obstacle avoidance and path-following. Two controllers were evaluated:

1. *Regulated Pure Pursuit (RPP)*: A local trajectory planner based on the *Pure Pursuit* algorithm, enhanced with collision detection and rotation-to-heading capabilities [60]. This latter feature is particularly useful, since LoCoBot can only detect obstacles in front of itself. Parameters were tuned for the application.
2. *Model Predictive Path Integral (MPPI)*: A model-based controller that dynamically deviates from the global path to avoid obstacles, returning to the planned trajectory when clear. The `Prefer Forward Critic` cost was increased to prioritize forward motion.

The `transform_tolerance` parameter configuration for both controllers was set to *0.2* in simulation, *0.5* with one external camera, and *0.8* with two external cameras. This was due to time difference present in each TF timestamp published by multiple devices.

Local Costmap

Both controllers shared the same *local costmap*, which included:

- A layer based on STVL, loading data from the onboard RealSense™ point cloud, with `voxel_decay` of 1.5 seconds, `voxel_size` of 5 millimeters and `obstacle_range` 1.5 meters.
- An inflation layer for maintaining safe distances.

4.4.5 BT Navigator

For interoperability, the default navigators, *navigate_to_pose* and *navigate_through_poses*, were retained. Parameters changes included:

- `robot_base_frame`: *locobot/base_footprint*
- `global_frame`: *map*
- `transform_tolerance`: Same as the parameter in *Controller server*

A custom behavior tree for the *navigate_to_pose* navigator was implemented for both controllers using BehaviorTree.CPP v4.6 and Groot2 [61].

Behavior Tree for Regulated Pure Pursuit (RPP)

The behavior tree developed for the RPP controller (*Appendix B.1*) included several features:

- A *ComputePath* sequence computes a path to the *human_tag* pose, truncate it to avoid collisions with the human target, and validate it. Failure at any step aborted the sequence.
- A *FollowPath* fallback sequence follows the computed path using the specified controller. If path-following fails, a wait action allows dynamic obstacles to clear.
- After two failed attempts to continue navigation, a recovery routine is invoked: the robot backs up 10 cm at 0.1 m/s to exit the collision area and recalculates the path. Although the robot operates without collision avoidance in this brief segment, it is assumed that there are no obstacles behind the LoCoBot since it passed that way less than 30 seconds earlier.
- The *Pipeline sequence* - implemented by Nav2 - allowed re-ticking actions to update the path every second.
- *Inverter* and *ForceFailure* nodes ensure continuous tree execution until the goal was reached or an abort was triggered.

Behavior Tree for Model Predictive Path Integral (MPPI)

Since the MPPI controller adapts paths in real time to avoid obstacles, recovery actions were unnecessary, resulting in a simpler tree (*Appendix B.2*).

4.4.6 Planner Server

Based on the path planner survey in [62], the *Smac Planner 2D* [63] was chosen: a cost-aware A* implementation optimized for circular footprint robots. Furthermore, this planner allows to plan paths in unknown areas of the map, taking advantage of the collision avoidance performed by the controllers.

Global Costmap

The global costmap included:

- STVL and inflation layers, similar to the local costmap. In this case, the `voxel_decay` time was adjusted based on the controller: higher for RPP, equal to the local costmap for MPPI.
- Static layer, which integrates known obstacles from the *Map Server*.

4.4.7 Velocity Smoother Server

The velocity smoother was tuned to achieve the desired maximum velocities and accelerations for the LoCoBot.

4.5 MoveIt 2 Configuration

The `Interbotix_xslocobot_moveit` package provides the essential configuration files, along with the URDF and SRDF necessary for the control of the WidowX-200 arm, minimizing the need for extensive custom adjustments.
Custom named poses for the arm and gripper were already added to the SRDF, allowing the `move_group` node to recognize predefined positions.
An external URDF has been integrated to simulate the Apriltag marker above the LoCoBot.

4.5.1 Loaded Components

The implemented motion planner is OMPL, configured with the following planning request adapters:

- Add time-optimal parameterization.
- Fix workspace bounds.
- Fix start state bounds.
- Fix start state collision.
- Fix start state path constraints.

The configuration also includes a *Planning Scene Monitor*, which primarily monitors joint states. Environmental changes, such as grasped objects or potential workspace obstacles, are not dynamically updated. This approach assumes a clutter-free workspace since the LoCoBot operates under human supervision, leaving safety evaluation to the user.

4.5.2 Grasping Position

Object grasping can be addressed in three ways:

1. Assuming fixed object dimensions for consistent grasping.
2. Using computer vision to adapt to variable object shapes.
3. Employing soft grippers to reduce the need for precise grasp positioning.

Given the fixed arm structure of LoCoBot, the third option is not feasible. Implementing vision-based grasping would require additional computational resources, diverging from this thesis's low-cost focus. Consequently, a consistent object size is assumed, enabling a fixed grasping position without requiring a collision model. Moreover, in this setup, objects are handed to the LoCoBot by a human rather than being picked autonomously.

4.6 Simulation Environment: Gazebo Classic Simulator

The simulation environment for testing the behavior of the state machine relies on *Gazebo Classic Simulator* [29], a widely used robotics simulation tool with an extensive library of plugins and ROS 2 compatibility.
The simulation model is configured using URDF files, which define the robot's links, joints, and kinematics. Again, these file were provided by the `Interbotix_xslocobot_descriptions` package.

Gazebo also supports *Actors*, which simulate human-like entities or mobile objects, critical for testing robot navigation in dynamic environments. Additionally, Gazebo's plugins enable communication with ROS 2 topics and services. For example, the `gazebo_ros2_control` plugin integrates with the `ros2_control` framework (*Appendix A*), allowing execution of differential drive or manipulator controllers.

4.6.1 Simulated world file

In Gazebo, simulation environments are configured using world files in SDF format, which define parameters, models, and spawn positions. For this project, the `Interbotix_xslocobot_sim` package launches a default world featuring the LoCoBot and a model of the *Trossen Robotics* office. To properly simulate the testing environment available in the university laboratory, several specific world files were developed:

- **big_arena**: Represents the maximum navigable space for the LoCoBot in the testing area.
- **small_arena**: A reduced version of *big_arena*.
- **static_obstacle**: Includes *big_arena* model in a shifted position and a static obstacle.
- **dynamic_obstacle**: Includes *big_arena* model with an actor simulating a moving obstacle.

The environment can be switched by setting the `world_filepath` parameter in the launch file of the `Interbotix_xslocobot_sim` package. *Figure 4.4* shows the *static_obstacle* world file. The other world files follow a similar configuration, differing only in dimensions or obstacle dynamics.

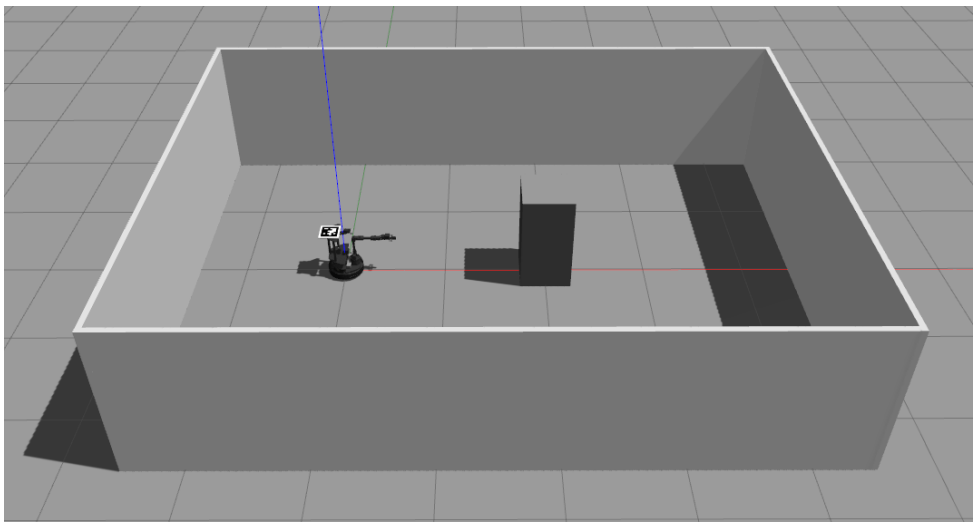


Figure 4.4: Example of Gazebo simulation: the *static_obstacle* world

4.6.2 Framework validation on simulated environment

To validate the framework in simulation, a RealSense™ camera was connected to a laptop, and commands were sent to the simulated model of the LoCoBot. The robot was able to navigate to a simulated human position, both static and dynamic. Additionally, the arm and gripper movements performed correctly. All tests met expectations, validating the functionality of the framework in the simulated environment.

4.7 Conclusions

This chapter provides a comprehensive overview of the framework, emphasizing the control functions for the LoCoBot and the logic behind the state machine. The customization of Nav2 focused on incorporating human-following capabilities and enhancing obstacle avoidance by leveraging the onboard depth camera. This should ensure reliable navigation even in dynamic environments with unmapped obstacles.

The validation in the simulation environment has shown promising results, providing a solid foundation for the tests with the physical robot, which are discussed in the next chapter.

Chapter 5

Experimental Results

Evaluating the capabilities demonstrated in the previous chapter in a real-world context is essential to determine if the implementation achieves the defined objectives even with the physical robot. The aim of this chapter is to assess the performance of the LoCoBot's state machine and verify whether the implementation meets the required specifications.
Before performing a general test to evaluate the overall completeness of the proposed framework, specialized tests must be conducted to validate each individual module. Of particular focus are the interaction module, which includes gesture recognition, and the navigation system. Camera calibration and pose estimation performance were previously assessed in *Section 3.5*.

Navigation tests, inspired by the standards outlined in the normative ISO 18646-2 [2], are presented in *Section 5.1*, while the interaction and gesture recognition tests (*Section 5.2*) are specifically tailored to the application. Finally, a general test to validate the functioning of the entire framework is conducted in *Section 5.3*.

5.1 Navigation tests

Autonomous navigation involves verifying several key aspects: obstacle detection, path planning, and obstacle avoidance in the presence of both static and dynamic objects.
Since all of those features were implemented using the Nav2 stack — whose capabilities are well-documented — it is assumed that, with the appropriate parameters defined in the previous chapter, everything performs as expected. The success of the subsequent tests will further validate this assertion.

5.1.1 Obstacle detection

The obstacle detection test aims to determine the area around the LoCoBot where obstacles can be detected.
According to the normative [2], the test involves measuring the accuracy of obstacle detection at various positions around the robot, both at the minimum and maximum detection ranges specified by sensor manufacturers. Since this thesis employs only a depth camera for environment perception, a modified version of the test was conducted.

Y_1 [mm]	X_1 [mm]	X_2 [mm]	α [deg]
500	600	1500	80

Table 5.1: Obstacle detection distances

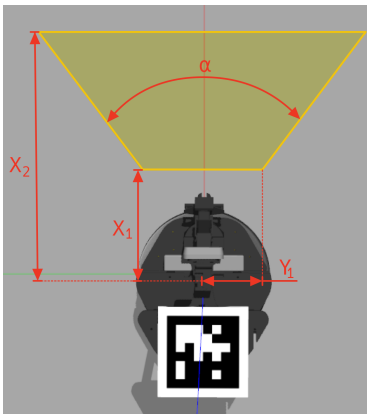


Figure 5.1: Obstacle detection coordinates

The test was conducted in a controlled, obstacle-free environment with the robot stationary. Objects of varying sizes were placed at different positions within the field of view of the depth camera, which is limited to the frontal area of the LoCoBot. The minimum and maximum detectable distances were measured from the robot's center.
The results, validated with a 1 m tall obstacle, are presented in *Table 5.1*.

The coordinate (X_1, Y_1), visible in *Figure 5.2* and relative to the LoCoBot's center, represent an external point of the camera's horizontal field of view, that extends up to the camera's sensor. The maximum detection distance Y_2 was software-limited to 1.5 m from the camera frame.
The minimum distance at which a 15 cm tall frontal obstacle could be detected was 70 cm, due to the angle of the vertical field of view.

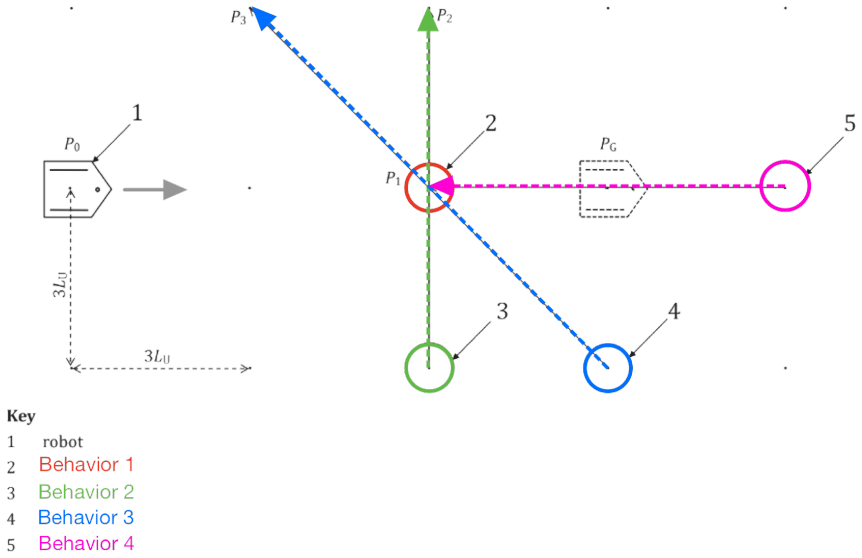


Figure 5.2: Test paths proposed by normative ISO 18646-2. Source [2]

Furthermore, to prevent the floor from being misidentified as an obstacle, a minimum obstacle height threshold of 8 cm was applied.
Nevertheless, challenges persist, particularly with inaccuracies in the robot's pose estimation. These inaccuracies, often stemming from image distortion during pose estimation, can lead to a misalignment between the robot's perceived point cloud and the real-world environment. Such discrepancies may result in portions of the floor being erroneously classified as obstacles.

5.1.2 Obstacle avoidance

This test evaluates whether the robot can successfully navigate from point to point in an environment with obstacles. Due to space constraints, the path length was set to 3,15 meters, instead of the 4,5 meters specified by [2]. All other dimensions were scaled accordingly.

Figure 5.2 illustrates the normative setup. A similar configuration was implemented, but to establish a baseline for comparison, a "Behavior 0" test (navigation without obstacles) was introduced, resulting in:

- **Behavior 0**: Linear path without obstacle
- **Behavior 1**: Linear path with static obstacle
- **Behavior 2**: Linear path with dynamic traversal obstacle
- **Behavior 3**: Linear path with dynamic obstacle

- **Behavior 4**: Linear path with dynamic frontal obstacle

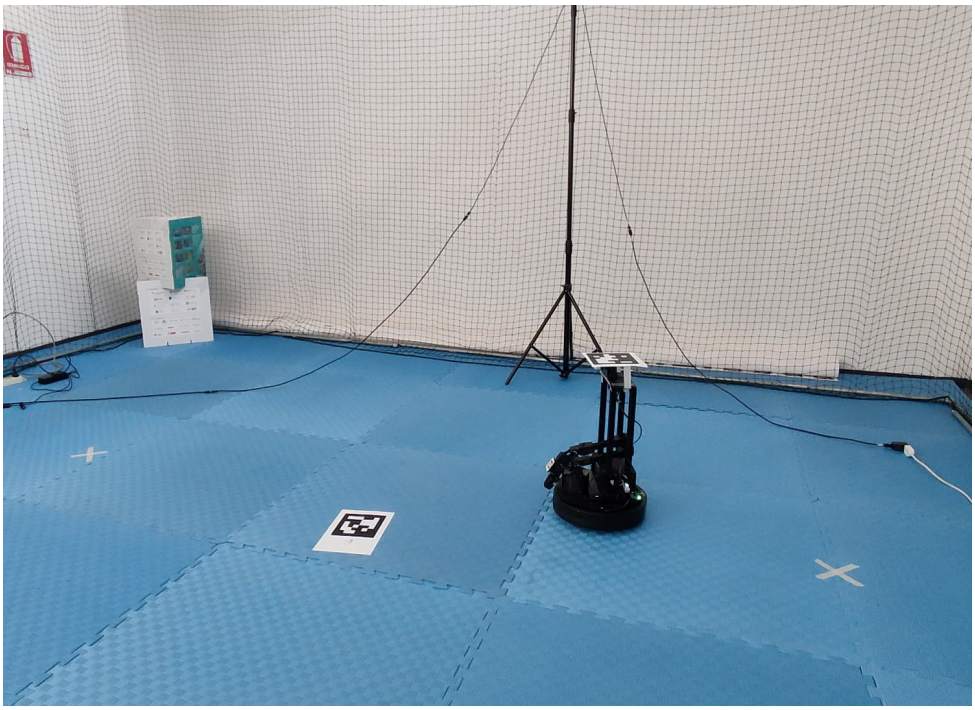
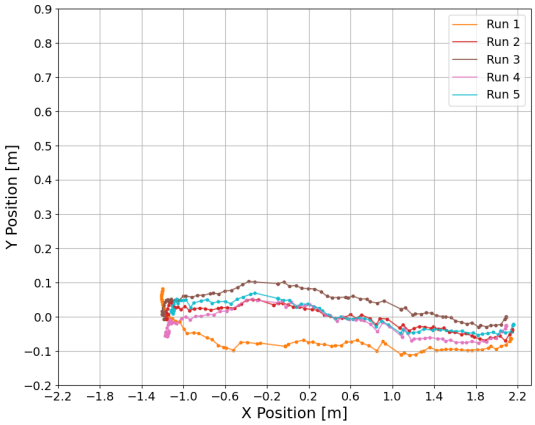


Figure 5.3: Testing area at the laboratory of the university

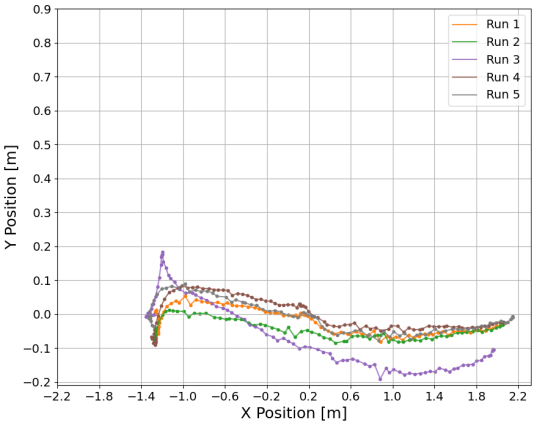
The test was conducted in the testing area at the laboratory of the university, visible in *Figure 5.3*. Manual repositioning of the robot was performed before each run but, to improve repeatability, the goal position was set programmatically via software to the same spot. Each behavior was executed five times to evaluate repeatability and similarities among runs while the LoCoBot's pose was tracked using a single camera, saving its position every 200 ms.
Tests were conducted using both the RPP and MPPI controllers to compare their performance.

In *Figure 5.4*, the comparison highlights visible differences in the paths taken with and without obstacles.
Especially with RPP, depending on obstacle distance and position, the robot stops and waits for the dynamic obstacle to clear the area, following the behavior implemented in *Section 4.4.4*. This feature, more perceptible in *Behavior 2*, is not visible from *Figure 5.5*, so it has been explicitly shown in *Appendix C*.

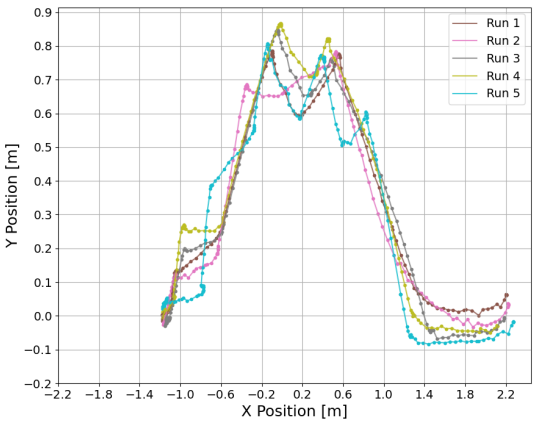
This test confirms that both controllers can avoid static and dynamic obstacles, each using distinct strategies. Firstly, as evident from *Figure 5.4* and *Figure 5.5*, the MPPI controller generates a smoother path. Moreover, the MPPI controller responds instantly to obstacles, dynamically replanning the robot's path to avoid them. While this approach can occasionally



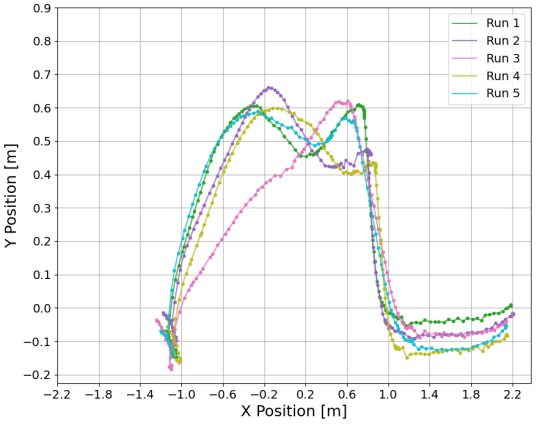
(a) Behavior 0 - RPP



(b) Behavior 0 - MPPI

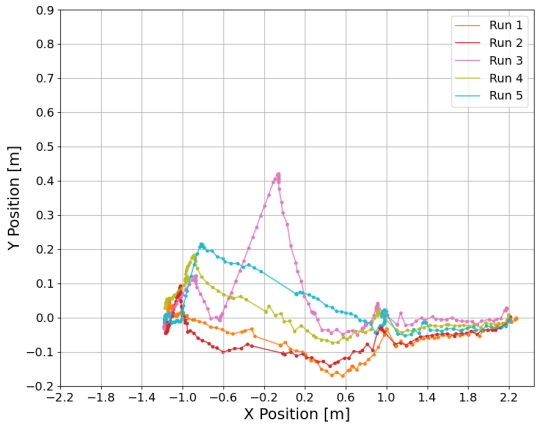


(c) Behavior 1 - RPP

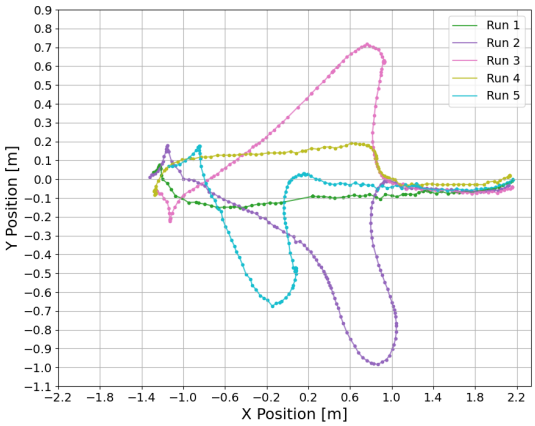


(d) Behavior 1 - MPPI

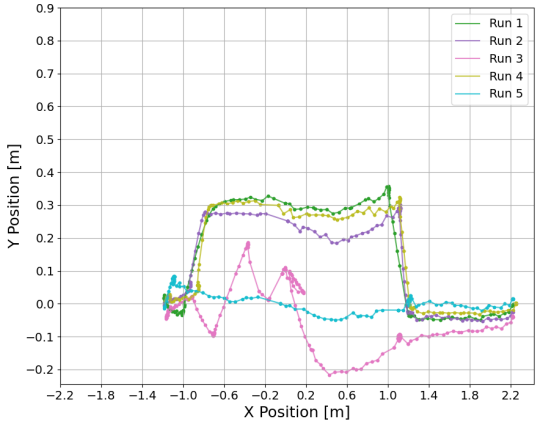
Figure 5.4: No obstacle and static obstacle tests



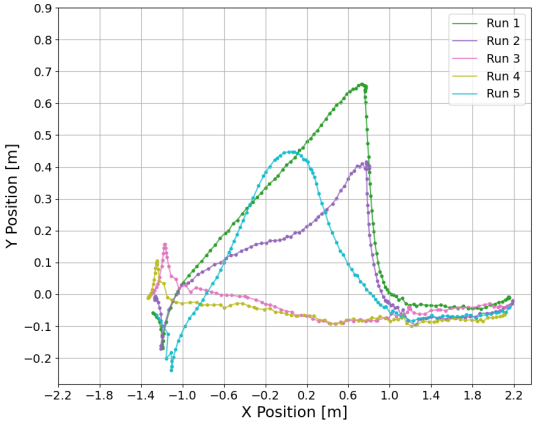
(a) Behavior 2 - RPP



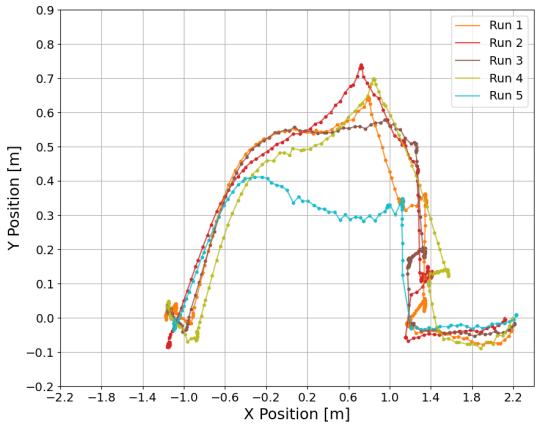
(b) Behavior 2 - MPPI



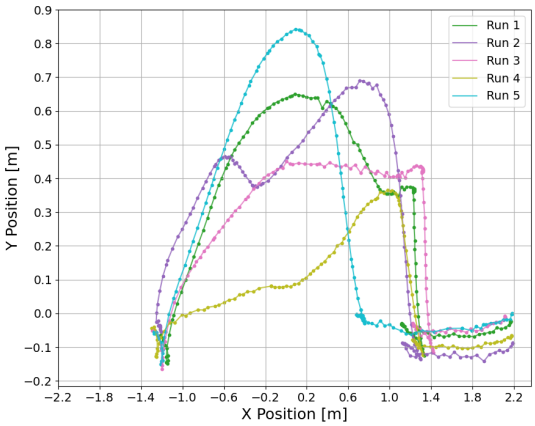
(c) Behavior 3 - RPP



(d) Behavior 3 - MPPI



(e) Behavior 4 - RPP



(f) Behavior 4 - MPPI

Figure 5.5: Dynamic obstacle tests

Run	Behavior 0		Behavior 1		Behavior 2		Behavior 3		Behavior 4	
	RPP	**MPPI**	**RPP**	**MPPI**	**RPP**	**MPPI**	**RPP**	**MPPI**	**RPP**	**MPPI**
1	17.8	18.4	31.9	27.6	27.1	20.5	26.7	26.0	75.1	38.5
2	18.0	16.5	29.1	28.6	31.1	28.3	28.3	26.2	43.6	27.8
3	17.3	18.9	31.6	27.3	32.5	26.2	42.5	21.8	57.2	34.2
4	17.5	19.0	34.0	28.2	29.3	24.0	28.3	21.7	45.2	29.4
5	17.0	18.7	42.0	30.2	30.3	29.5	29.0	22.2	31.3	30.2
Avg	17.52	18.3	33.72	28.38	30.06	25.7	30.96	23.58	50.48	32.02

Table 5.2: Time of each run and average time in seconds

lead to relevant deviations from the goal, especially in environments with multiple obstacles causing excessive detours, the controller's replanning frequency ensures consistently reliable performance. In contrast, the RPP controller adopts a more conservative strategy, waiting for dynamic obstacles to clear, which can save time in such scenarios.

The superiority of the MPPI controller in these tests is further demonstrated in *Table 5.2*, which presents the time taken to reach the goal for each run. This time is measured from when the navigation goal is accepted by the server to when the navigation server reports success. In the table, the best runs for each controller are highlighted in green, and the worst runs are marked in red. For each behavior, the best and worst results are underlined.

In the average value of each behavior ("Avg" in *Table 5.2*), with the exception of *Behavior 0*, the MPPI controller outperforms the RPP. Given that in this setup the LoCoBot had sufficient space in all directions to avoid the obstacle, the time difference between the averages, aside from some outliers, can be attributed to the waiting behavior of the RPP controller.

Moreover, the time discrepancies can also be partially attributed to specific test conditions. Near the goal, the camera detected the LoCoBot's pose as not parallel to the ground, leading the floor to be mistakenly identified as an obstacle. This highlights the MPPI controller's superior resilience to pose measurement inaccuracies.

These tests confirmed the proper tuning of parameters and the feasibility of implementing both controllers. The choice between them depends on the specific scenario. For this thesis, the MPPI controller is selected due to its smoothness, speed, and obstacle avoidance capabilities, especially given the low presence of dynamic obstacles in the tested scenarios.

5.2 Interaction tests

To validate the interaction module, it is crucial to determine whether the robotic arm performs tasks accurately as instructed via gesture recognition. This part of the testing includes evaluating the gesture recognition module and conducting a functional test. The base of the robot was not considered for the latter test, as only the camera and robotic arm were required.

5.2.1 Gesture recognition

The performance of the gesture recognition module was evaluated in various setup:

- **Base**: Gesture recognition was tested against a white background under a 500W frontal light.
- **Noise**: The background was random but consistent through all the tests.
- **Net**: The background was a white wall with a green net used to enclose the arena.
- **Loco_Noise**: Test conducted with the LoCoBot hardware against the same background as *Noise*.
- **Loco_Net**: Same setup as *Net*, but with the LoCoBot hardware.

The first three tests were conducted using a laptop with an Intel® Core™ i7-9750H CPU @ 2.60 GHz and a RealSense™ camera connected, while the remaining tests used the onboard NUC and camera. Tests were performed at 40 cm and 100 cm from the camera by three subjects. Each test included five gesture collection runs, with the gesture sequence for each run randomly selected from those used in this thesis.
All users were provided with video feedback from the camera.

KPI	Base	Noise	Net	Loco_Noise	Loco_Net
Total Accuracy [%]	85.71	95.92	86.57	92.06	95.18
Average Detection Time [ms]	53.66	53.95	51.43	70.44	71.15
Standard Deviation Det. Time [ms]	7.91	9.38	8.04	3.61	3.37
Average Confidence [%]	0.71	0.72	0.68	0.80	0.82
Standard Deviation Confidence [%]	0.13	0.11	0.12	0.13	0.10

Table 5.3: Tests at 40 cm

KPI	Base	Noise	Net	Loco_Noise	Loco_Net
Total Accuracy [%]	93.33	79.25	86.44	98.36	97.47
Average Detection Time [ms]	56.15	52.50	51.63	69.75	71.48
Standard Deviation Det. Time [ms]	7.80	8.10	9.61	4.60	3.36
Average Confidence [%]	0.74	0.68	0.71	0.79	0.80
Standard Deviation Confidence [%]	0.11	0.12	0.13	0.14	0.12

Table 5.4: Tests at 100 cm

As shown in *Tables 5.3* and *5.4*, no significant differences in detection time or accuracy were observed between tests performed with the laptop and the NUC. However, it is worth noting that the laptop's performance degraded significantly in battery mode (data not included in these results).

To simulate a realistic scenario, *Tester 2* was instructed to avoid precision in his hand poses, adopting a natural posture instead. He was also asked to press the "ready" button as quickly as possible to minimize the time spent positioning his hand. *Figure 5.6* demonstrates lower performances under this latter scenario, denoting reduced accuracy when the hand is improperly positioned — a common occurrence during live-feed detection without visual feedback. Detailed data for each tester and test are provided in *Appendix D*.

5.2.2 Arm movement

A qualitative analysis was conducted to evaluate the arm's response to gestures. While no quantitative data were collected, *Figure 5.7* illustrates that the arm successfully executed commands associated with specific gestures. Each gesture is marked by a red circle in the first frame of every command.

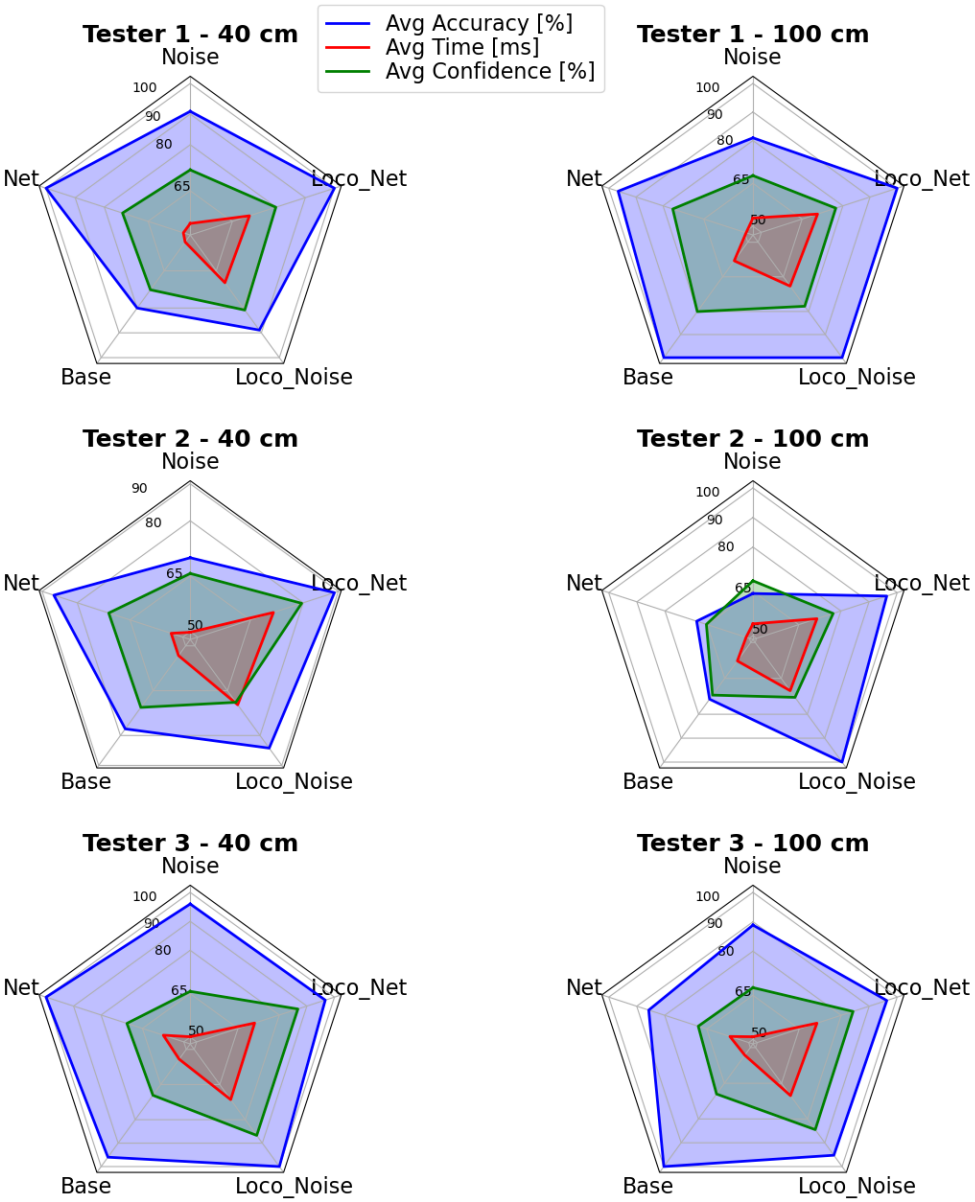


Figure 5.6: Gesture test differentiation by tester

Figure 5.7: Recognition test

LoCoBot consistently responded to the given commands, demonstrating the effective implementation of the interaction module. Furthermore, this test inspired the usability improvement cited in *Section 4.1.1*.

5.3 General test

After successfully passing all the previous specific tests for each module, a general test was conducted to verify the implementation of the state machine and ensure proper transitions between states in response to human gestures. To implement this test, a routine was developed to specifically evaluate each state transition. The routine includes the following steps:

1. Start the state machine and clear any error states
2. Request interaction with the LoCoBot
3. Open the gripper
4. Close the gripper (potentially passing an object to the LoCoBot)
5. Return to the IDLE state.
6. Request navigation
7. Navigate around, ensuring the LoCoBot follows the human
8. Interrupt navigation
9. Request interaction again
10. Open the gripper and take the object
11. Return the LoCoBot to the IDLE state

Additionally, during the navigation, the obstacle avoidance will be tested again, while the pose estimation method will be evaluated throughout the entire process. It is important to remember that if the TF of either the LoCoBot or the human is not detected for a certain period (5 seconds in this case), the state machine will trigger the ERROR state. The LoCoBot was tracked using two Kinect cameras.
To demonstrate the ease of operation of this framework, both trained individuals and untrained participants were considered. Both groups were aware of the optimal positioning in front of the camera and the command associated with each gesture. The use of the Apriltag marker was not considered a differentiating factor, as each participant was instructed to walk while always pointing the marker towards the camera. The only difference between these groups was the amount of time they had spent using the robot.

Even though the gesture recognition threshold for validating a gesture was set based on prior tests, some issues were encountered with the gripper interaction gestures, resulting in

recognition times of up to 10-20 seconds. This could be attributed to the fact that users were not aware of the camera's perspective, leading to suboptimal gesture performance due to a trial-and-error approach in positioning.

A more significant difference was observed during the navigation phase. Trained users knew the optimal positioning of the marker to ensure the robot followed them without being recognized as obstacles. However, the limited size of the testing area meant that the robot was often navigating close to static obstacles or near the human subject, resulting in continuous re-planning that also accounted for human avoidance. On the other hand, due to the proper tuning of parameters, the robot was able to follow the human with minimal delay.

A video of this test is available at `https://alessio-lovato-unipd.github.io/tesi_locobot/`.
However, *Figure 5.8* provides a general idea of the test execution, as these frames succession have been extracted from the video.

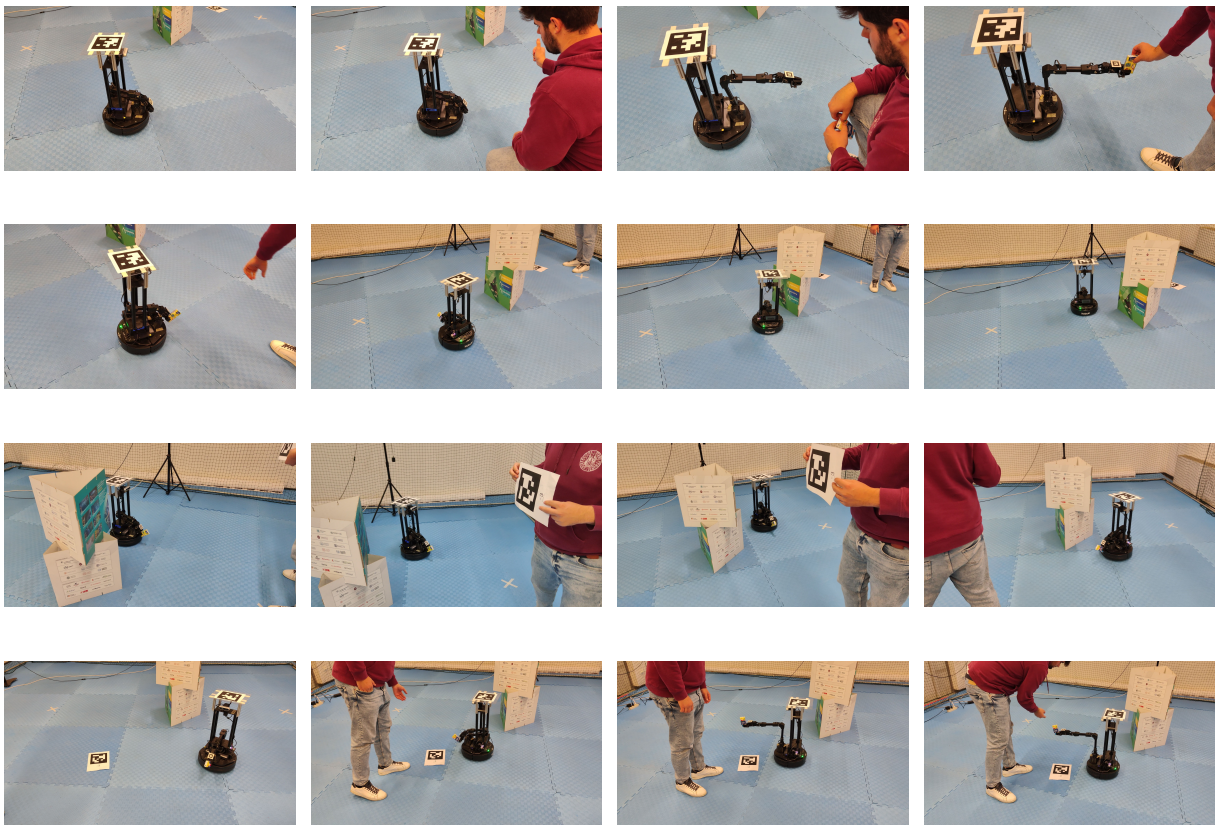


Figure 5.8: General test video frames

5.4 Conclusions

The validation tests for each individual module yielded promising results. Navigation testing demonstrated that, even if both controller were usable, the MPPI controller was the optimal choice for this implementation, given the environment's low percentage of dynamic obstacles. However, it was observed that navigation in narrow environments posed challenges, primarily because the human to follow was often perceived as an obstacle, hindering the path planning process.
The gesture recognition module, despite performing exceptionally well when tested in isolation from the state machine, exhibited minor issues during the comprehensive test. Nonetheless, it successfully executed the required tasks.
Overall, the general test was successful, as the state machine efficiently handled all state transitions. Additionally, this test affirmed the robustness of the pose estimation technology based on fiducial markers and the effectiveness of the calibration method.

In conclusion, the testing phase demonstrated that the entire framework is a valid solution.

Chapter 6

Conclusions

This thesis introduced the implementation of a human-guided framework for real-world applications involving human-robot interaction. The chosen platform was an autonomous mobile manipulator, specifically the LoCoBot WX200. While the AMM was developed for research purposes, the framework's concept can be adapted to other contexts, including industrial automation and service robotics.
Additionally, within the Industry 4.0 context, the proposed framework offers significant potential for scalability: state machines controlling individual robots can be expanded to connect with larger systems, facilitating precise management and coordination of extensive robot fleets.

The primary objectives of this framework included human-following capabilities, interaction through the robotic arm, and human-driven control. Central to the design was the need for intuitive control mechanisms, leading to the adoption of the hand gesture recognition model provided by *MediaPipe* as the main input method. The state machine at the core of the framework processes human commands and orchestrates actions through navigation and interaction modules.

Gesture Recognition

During the validation phase, some delays were observed in gesture recognition. Although minimal, these delays could affect user experience, potentially causing frustration due to slow task execution. Two improvements are suggested to address this issue: refining the gesture recognition model for greater robustness in real-world environments or exploring alternative interaction methods like voice recognition. Voice commands, while computationally heavier, would allow greater flexibility by eliminating the need for proximity to the robot during operation and resolving issues such as the robot misidentifying the user as an obstacle.

State machine

The state machine serves as the control logic for the AMM, handling both the internal states of the robot and its interaction with external modules. Developed using the ROS 2 framework, it communicates with two critical packages: Nav2 for navigation tasks and MoveIt 2 for manipulation tasks. Commands from the human, processed through the gesture recognition module, are translated into actions executed by the state machine.
This architecture enables seamless integration of navigation and interaction functionalities, while maintaining modularity and interchangeability.

Interaction module

Built on MoveIt 2, the interaction module enables control of the robotic arm. While minimally customized for this project, the module performed as expected, successfully executing planned movements. However, a limitation was noted: the arm's interaction pose is fixed.
While dynamically calculating optimal poses could enhance flexibility, it was deemed outside the scope of this thesis. Moreover, dynamically generated poses could introduce safety risks by introducing unpredictable arm movements, potentially catching users off guard.

Navigation module

The navigation module, a customization of the Nav2 stack, represents a core component of the framework. It incorporates features like human-following and obstacle avoidance, leveraging the onboard depth camera for real-time environmental perception. The module demonstrated robust navigation capabilities, effectively avoiding both static and dynamic obstacles, including unmapped ones. Human-following functionality, implemented using a clever usage of the state machine, was validated during the general test.
The pose estimation, necessary for the navigation, has been implemented through the localization module.

Despite these successes, a limitation emerged: the followed human was also identified as an obstacle, complicating close-range following in confined spaces. A potential solution involves introducing a filtering mechanism to merge depth and RGB visual data, excluding the human from being classified as an obstacle. However, implementing this solution would require a more powerful processing unit and an upgraded battery to sustain efficient long-term operations.

Localization module

This module estimates the poses of both the LoCoBot and the human in the environment using Apriltag fiducial markers. Pose tracking was achieved with two Kinect cameras, mitigating occlusion issues that could arise with a single camera. While this redundancy improved reliability, it also introduced the need for precise calibration of the cameras relative to the navigation map origin.
A calibration technique was developed to synchronize the camera data, minimizing positional discrepancies between the two perspectives. The results demonstrated effective pose estimation, enabling accurate navigation.

Final considerations

Although some areas for improvement have been identified, the framework was successfully validated through normative-inspired tests. These evaluations highlighted its strengths and limitations, confirming the functionality of the Nav2 and MoveIt 2 stacks within the proposed architecture.
Ultimately, the framework achieved the objectives of this thesis, demonstrating its capability for autonomous navigation, human-robot interaction during a comprehensive test.

Suggested improvements

An additional enhancement suggested, based on the conducted tests, is the integration of a LIDAR sensor. This would significantly improve navigation reliability by providing the robot with a comprehensive understanding of its surroundings, enabling reverse navigation and enhancing safety.

Bibliography

[1] J. Horáková and J. Kelemen, "The Robot Story: Why Robots Were Born and How They Grew Up," in *The Mechanical Mind in History*, The MIT Press, Feb. 2008, ISBN: 9780262256384. DOI: 10.7551/mitpress/7626.003.0013. eprint: https://direct.mit.edu/book/chapter-pdf/2284933/9780262256384_cal.pdf. [Online]. Available: https://doi.org/10.7551/mitpress/7626.003.0013.

[2] *Robotics — performance criteria and related test methods for service robots - part 2: Navigation*, ISO 18646-2:2024, 2024.

[3] A. S. Silva Oliveira, M. C. dos Reis, F. A. X. da Mota, M. E. M. Martinez, and A. R. Alexandria, "New trends on computer vision applied to mobile robot localization," *Internet of Things and Cyber-Physical Systems*, vol. 2, pp. 63–69, 2022, ISSN: 2667-3452. DOI: https://doi.org/10.1016/j.iotcps.2022.05.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2667345222000128.

[4] *Robots and robotic devices — collaborative robots*, ISO/TS15066:2016, 2016.

[5] M. Basiri, J. Gonçalves, J. Rosa, A. Vale, and P. Lima, "An autonomous mobile manipulator to build outdoor structures consisting of heterogeneous brick patterns," *SN Applied Sciences*, vol. 3, no. 5, p. 558, 2021, ISSN: 2523-3971. DOI: 10.1007/s42452-021-04506-7. [Online]. Available: https://doi.org/10.1007/s42452-021-04506-7.

[6] H. Engemann, S. Du, S. Kallweit, P. Cönen, and H. Dawar, "Omnivil—an autonomous mobile manipulator for flexible production," *Sensors*, vol. 20, no. 24, 2020, ISSN: 1424-8220. DOI: 10.3390/s20247249. [Online]. Available: https://www.mdpi.com/1424-8220/20/24/7249.

[7] *Development of an Autonomous Mobile Manipulator for Industrial and Agricultural Environments — hdl.handle.net*, https://hdl.handle.net/10589/223386, [Accessed 22-10-2024].

[8] S. Al-Hussaini *et al.*, *Human-supervised semi-autonomous mobile manipulators for safely and efficiently executing machine tending tasks*, 2020. arXiv: 2010.04899 [cs.RO]. [Online]. Available: https://arxiv.org/abs/2010.04899.

[9] L. Tagliavini, L. Baglieri, G. Colucci, A. Botta, C. Visconte, and G. Quaglia, "D.o.t. paquitop, an autonomous mobile manipulator for hospital assistance," *Electronics*, vol. 12, no. 2, 2023, ISSN: 2079-9292. DOI: 10.3390/electronics12020268. [Online]. Available: https://www.mdpi.com/2079-9292/12/2/268.

[10] J. Carius, M. Wermelinger, B. Rajasekaran, K. Holtmann, and M. Hutter, "Deployment of an autonomous mobile manipulator at mbzirc," *Journal of Field Robotics*, vol. 35, no. 8, pp. 1342–1357, 2018. DOI: https://doi.org/10.1002/rob.21825. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/rob.21825. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21825.

[11] *RB-KAIROS+ Manipulator - Pick & Place Robot | Robotnik® — robotnik.eu*, https://robotnik.eu/products/mobile-robots/rb-kairos-2/#rb-kairos-plus-ur-five, [Accessed 05-11-2024].

[12] *LoCoBot — robots.ros.org*, https://robots.ros.org/locobot/, [Accessed 28-11-2024].

[13] *Home | Trossen Robotics — trossenrobotics.com*, https://www.trossenrobotics.com/, [Accessed 28-11-2024].

[14] *DYNAMIXEL | All-in-one Smart Actuator — dynamixel.com*, https://www.dynamixel.com/, [Accessed 28-11-2024].

[15] *LoCoBot WidowX-200; Interbotix X-Series LoCoBot Documentation — docs.trossenrobotics.com*, https://docs.trossenrobotics.com/interbotix_xslocobots_docs/specifications/locobot_wx200.html, [Accessed 21-09-2024].

[16] *Differential wheeled robot - Wikipedia — en.wikipedia.org*, https://en.wikipedia.org/wiki/Differential_wheeled_robot, [Accessed 22-09-2024].

[17] *WidowX-200; Interbotix X-Series Arms Documentation — docs.trossenrobotics.com*, https://docs.trossenrobotics.com/interbotix_xsarms_docs/specifications/wx200.html, [Accessed 23-09-2024].

[18] ROBOTIS, *U2D2 Manual*, [Online]. Accessed 23-09-2024. Available: https://emanual.robotis.com/docs/en/parts/interface/u2d2/.

[19] *Depth Camera D435 — intelrealsense.com*, https://www.intelrealsense.com/depth-camera-d435/, [Accessed 28-11-2024].

[20] *ROS Package: Realsense2_camera — index.ros.org*, https://index.ros.org/p/realsense2_camera/, [Accessed 23-09-2024].

[21] *GitHub - Interbotix/interbotix_ros_core: Core ROS Packages for Interbotix Robots — github.com*, https://github.com/Interbotix/interbotix_ros_core, [Accessed 23-09-2024].

[22] *GitHub - Interbotix/interbotix_ros_toolboxes: Support-level ROS Packages for Interbotix Robots — github.com*, https://github.com/Interbotix/interbotix_ros_toolboxes, [Accessed 23-09-2024].

[23] *GitHub - Interbotix/interbotix_ros_rovers: ROS Packages for Interbotix Rovers — github.com*, https://github.com/Interbotix/interbotix_ros_rovers, [Accessed 23-09-2024].

[24] *ROS Package: Joint_state_publisher — index.ros.org*, https://index.ros.org/p/joint_state_publisher, [Accessed 18-09-2024].

[25] *ROS Package: Robot_state_publisher — index.ros.org*, https://index.ros.org/p/robot_state_publisher/, [Accessed 23-09-2024].

[26] *Rviz2: ROS 3D Robot Visualizer — github.com*, https://github.com/ros2/rviz, [Accessed 23-09-2024].

[27] *Interbotix X-Series Arms Documentation — docs.trossenrobotics.com*, https://docs.trossenrobotics.com/interbotix_xsarms_docs/ros_interface/ros2/overview/xs_sdk.html, [Accessed 15-10-2024].

[28] *ROBOTIS e-Manual — emanual.robotis.com*, https://emanual.robotis.com/docs/en/software/dynamixel/dynamixel_workbench/, [Accessed 15-10-2024].

[29] OSRF, *Gazebo — classic.gazebosim.org*, https://classic.gazebosim.org/, [Accessed 23-09-2024].

[30] *ROS Index — index.ros.org*, https://index.ros.org/r/gazebo_ros_pkgs/github-ros-simulation-gazebo_ros_pkgs, [Accessed 23-09-2024].

[31] *MoveIt 2 Motion Planning Configuration; Interbotix X-Series LoCoBot Documentation — docs.trossenrobotics.com*, https://docs.trossenrobotics.com/interbotix_xslocobots_docs/ros2_packages/moveit_motion_planning_configuration.html, [Accessed 23-09-2024].

[32] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, eabm6074, 2022. DOI: 10.1126/scirobotics.abm6074. [Online]. Available: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[33] *Discovery; ROS 2 Documentation: Humble documentation*, https://docs.ros.org/en/humble/Concepts/Basic/About-Discovery.html, [Accessed 12-09-2024].

[34] *Composition ROS 2 Documentation: Humble documentation*, https://docs.ros.org/en/humble/Concepts/Intermediate/About-Composition.html, [Accessed 21-10-2024].

[35] *Topics; ROS 2 Documentation: Humble documentation — docs.ros.org*, https://docs.ros.org/en/humble/Concepts/Basic/About-Topics.html, [Accessed 12-09-2024].

[36] *Services; ROS 2 Documentation: Humble documentation*, https://docs.ros.org/en/humble/Concepts/Basic/About-Services.html, [Accessed 12-09-2024].

[37] *Actions; ROS 2 Documentation: Humble documentation*, https://docs.ros.org/en/humble/Concepts/Basic/About-Actions.html, [Accessed 12-09-2024].

[38] T. Dirk, *ROS 2 middleware interface*, https://design.ros2.org/articles/ros_middleware_interface.html, [Accessed 12-09-2024], 2017.

[39] *Different ROS 2 middleware vendors; ROS 2 Documentation: Humble documentation*, https://docs.ros.org/en/humble/Concepts/Intermediate/About-Different-Middleware-Vendors.html, [Accessed 12-09-2024].

[40] G. Pardo-Castellote, "Omg data-distribution service: Architectural overview," in *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.*, 2003, pp. 200–206. DOI: 10.1109/ICDCSW.2003.1203555.

[41] S. Macenski, F. Martín, R. White, and J. G. Clavero, "The marathon 2: A navigation system," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020, pp. 2718–2725. DOI: 10.1109/IROS45743.2020.9341207.

[42] *Nav2; Nav2 1.0.0 documentation*, https://docs.nav2.org/, [Accessed 13-09-2024].

[43] *REP 105 – Coordinate Frames for Mobile Platforms (ROS.org)*, https://www.ros.org/reps/rep-0105.html, [Accessed 17-09-2024], 2010.

[44] S. Macenski, D. Tsai, and M. Feinberg, "Spatio-temporal voxel layer: A view on robot perception for the dynamic world," *International Journal of Advanced Robotic Systems*, vol. 17, no. 2, 2020. DOI: 10.1177/1729881420910530. [Online]. Available: https://doi.org/10.1177/1729881420910530.

[45] *MoveIt 2 Documentation; MoveIt Documentation: Humble documentation*, https://moveit.picknik.ai/humble/index.html, [Accessed 17-09-2024].

[46] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, 2012, https://ompl.kavrakilab.org. DOI: 10.1109/MRA.2012.2205651.

[47] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa, "Chomp: Gradient optimization techniques for efficient motion planning," in *2009 IEEE International Conference on Robotics and Automation*, 2009, pp. 489–494. DOI: 10.1109/ROBOT.2009.5152817.

[48] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," *Autonomous Robots*, 2013, Software available at https://octomap.github.io. DOI: 10.1007/s10514-012-9321-0. [Online]. Available: https://octomap.github.io.

[49] C. Matteo, L. Alessio, and P. Alberto, *Tecniche di localizzazione indoor per droni basate su fiducial markers — hdl.handle.net*, https://hdl.handle.net/20.500.12608/34419, [Accessed 06-11-2024].

[50] M. Krogius, A. Haggenmiller, and E. Olson, "Flexible layouts for fiducial tags," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019.

[51] S. Garrido-Jurado, R. Muñoz-Salinas, F. Madrid-Cuevas, and M. Marín-Jiménez, "Automatic generation and detection of highly reliable fiducial markers under occlusion," *Pattern Recognition*, vol. 47, no. 6, pp. 2280–2292, 2014, ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2014.01.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0031320314000235.

[52] B. Benligiray, C. Topal, and C. Akinlar, "Stag: A stable fiducial marker system," *Image and Vision Computing*, vol. 89, pp. 158–169, 2019, ISSN: 0262-8856. DOI: https://doi.org/10.1016/j.imavis.2019.06.007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0262885619300903.

[53] *GitHub - Adlink-ROS/apriltag_ros: A ROS2 wrapper of the AprilTag 3 visual fiducial detector — github.com*, https://github.com/Adlink-ROS/apriltag_ros, [Accessed 27-10-2024].

[54] *GitHub - microsoft/Azure_Kinect_ROS_Driver at humble*, https://github.com/microsoft/Azure_Kinect_ROS_Driver/tree/humble, [Accessed 21-10-2024].

[55] *GitHub - mguidolin/Azure_Kinect_ROS_Driver: A ROS sensor driver for the Azure Kinect Developer Kit. — github.com*, https://github.com/mguidolin/Azure_Kinect_ROS_Driver, [Accessed 06-11-2024].

[56] *GitHub - mguidolin/apriltag_calibration — github.com*, https://github.com/mguidolin/apriltag_calibration, [Accessed 06-11-2024].

[57] *ROS Package: Image_proc*, https://index.ros.org/p/image_proc, [Accessed 21-10-2024].

[58] F. Zhang *et al.*, *Mediapipe hands: On-device real-time hand tracking*, 2020. DOI: 10.48550/ARXIV.2006.10214. [Online]. Available: https://arxiv.org/abs/2006.10214.

[59] *GitHub - arshadlab/gazebo_map_creator — github.com*, https://github.com/arshadlab/gazebo_map_creator, [Accessed 16-11-2024].

[60] S. Macenski, S. Singh, F. Martín, and J. Ginés, "Regulated pure pursuit for robot path tracking," *Autonomous Robots*, vol. 47, no. 6, pp. 685–694, 2023, ISSN: 1573-7527. DOI: 10.1007/s10514-023-10097-6. [Online]. Available: https://doi.org/10.1007/s10514-023-10097-6.

[61] *BehaviorTree.CPP — behaviortree.dev*, https://www.behaviortree.dev/, [Accessed 18-11-2024].

[62] S. Macenski, T. Moore, D. V. Lu, A. Merzlyakov, and M. Ferguson, "From the desks of ros maintainers: A survey of modern & capable mobile robotics algorithms in the robot operating system 2," *Robotics and Autonomous Systems*, vol. 168, p. 104 493, 2023, ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2023.104493. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S092188902300132X.

[63] S. Macenski, M. Booker, and J. Wallace, "Open-source, cost-aware kinematically feasible planning for mobile and surface robotics," 2024. DOI: 10.48550/arXiv.2401.13078. arXiv: 2401.13078 [cs.RO]. [Online]. Available: https://arxiv.org/abs/2401.13078.

[64] *Ros2_control documentation - Humble; ROS2_Control: Humble Nov 2024 documentation — control.ros.org*, https://control.ros.org/humble/index.html, [Accessed 04-11-2024].

Appendix A

`ros2_control` package

ROS 2 leverages `ros2_control` [64] to standardize controller implementations. *Figure A.1* illustrates the structure of this framework.

Controller Manager

The *controller_manager* node manages the controller's lifecycles and hardware interfaces, supporting strategies such as position, velocity, and effort-based control. During each control loop, the manager sequentially reads the hardware states, updates active controllers, and outputs calculated commands to the hardware interfaces. Configuration parameters, loaded from YAML files, specify the controller types and joint names.

For differential drive, the *Diff Drive Controller* interprets velocity commands for the left and right wheels, enabling coordinated movement. For robotic manipulators, the *Joint Trajectory Controller* handles multi-DoF movement, generating smooth, time-parameterized commands for each joint based on desired positions, velocities, and accelerations.

Resource Manager

The Resource Manager abstracts hardware components and provides standardized access to physical devices, facilitating seamless integration into the ROS 2 ecosystem. It ensures smooth communication for command and state interfaces, enabling diverse hardware compatibility without requiring specific implementations.

Hardware Components

The `ros2_control` framework introduces two critical interfaces: the *Hardware Interface* for reading joint states and the *Command Interface* for sending commands and reading states. Hard-

ware components are categorized as:

- **Actuator**: Simple 1-DoF components, such as motors, with direct control and feedback.
- **Sensor**: Read-only components providing environmental data.
- **System**: Complex hardware, like multi-DOF robotic arms, supporting both reading and writing capabilities.

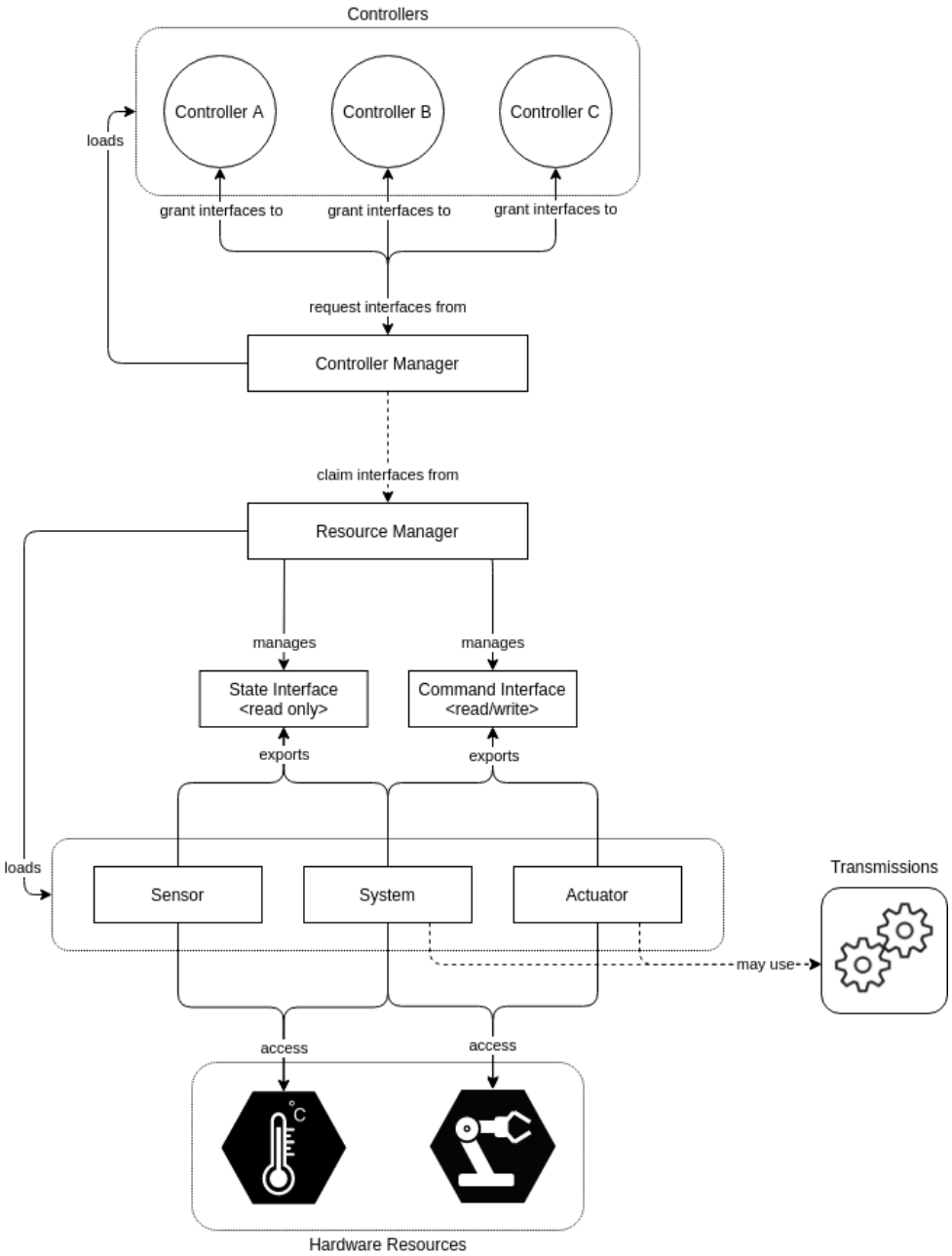
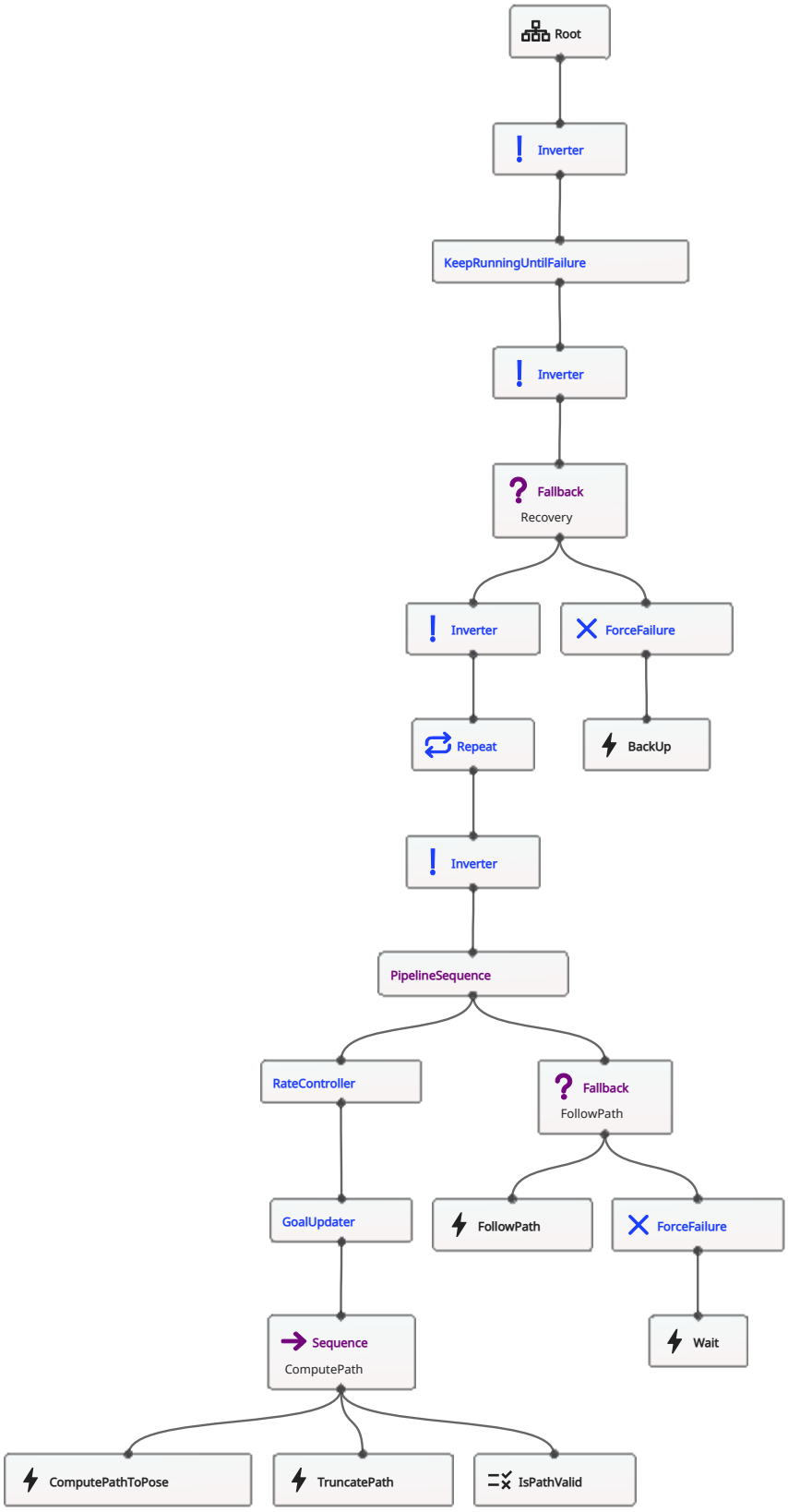


Figure A.1: Components architecture of ROS 2 control. Source [64]

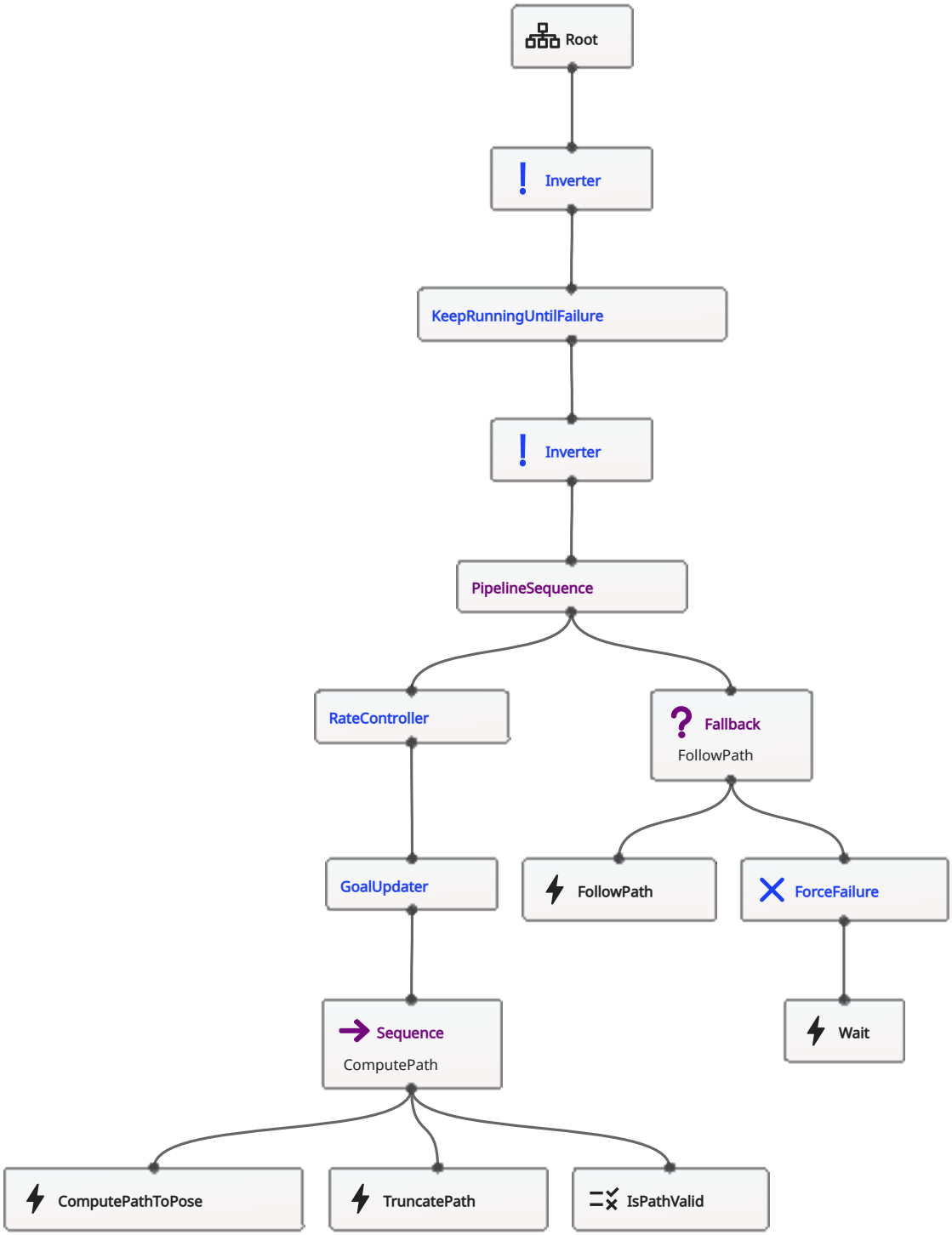
Appendix B

Nav2 Custom Behavior Trees

B.1 RPP Behavior Tree

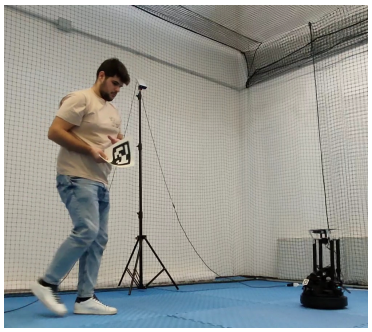


B.2 MPPI Behavior Tree

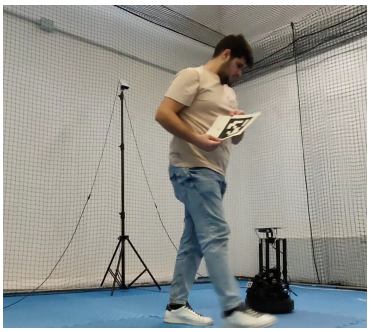


Appendix C

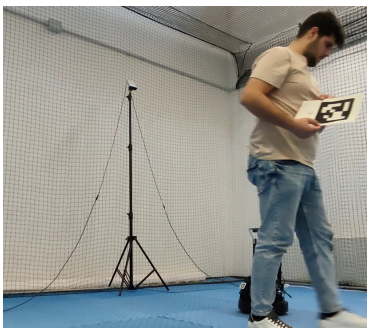
Frame sequence of Behavior 2 with RPP



(a) Behavior 3 - frame 1



(b) Behavior 3 - frame 2



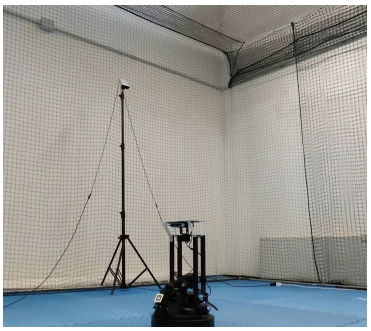
(c) Behavior 3 - frame 3

(d) Behavior 3 - frame 4

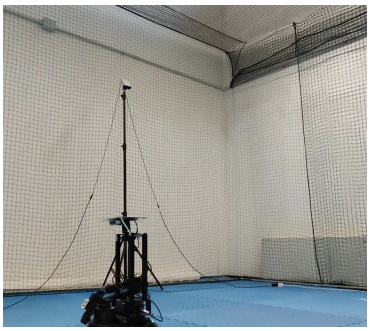
(e) Behavior 3 - frame 5

(f) Behavior 3 - frame 6

(g) Behavior 3 - frame 7



(h) Behavior 3 - frame 8



(i) Behavior 3 - frame 9

Appendix D

Gesture Recognition Tables

In this appendix are highlighted the data collected during the gesture recognition tests.

Metric	Base	Net	Noise	Loco_Noise	Loco_Net
Total Accuracy [%]	80.00	100.00	90.91	88.89	100.00
Average Detection Time [ms]	53.38	52.91	54.33	69.80	70.88
Standard Deviation Det. Time [ms]	7.37	10.56	7.95	4.33	3.50
Average Confidence [%]	0.73	0.74	0.72	0.81	0.80
Standard Deviation Confidence [%]	0.15	0.12	0.10	0.12	0.10

Table D.1: Analysis for Tester 1 at 40 cm

Metric	Base	Net	Noise	Loco_Noise	Loco_Net
Total Accuracy [%]	77.78	86.67	70.00	84.21	88.89
Average Detection Time [ms]	53.26	53.30	49.83	69.71	71.54
Standard Deviation Det. Time [ms]	9.01	8.46	7.67	3.63	3.82
Average Confidence [%]	0.71	0.71	0.66	0.69	0.80
Standard Deviation Confidence [%]	0.15	0.12	0.10	0.14	0.10

Table D.2: Analysis for Tester 2 at 40 cm

Metric	Base	Net	Noise	Loco_Noise	Loco_Net
Total Accuracy [%]	96.00	100.00	96.00	100.00	96.67
Average Detection Time [ms]	54.17	57.44	50.16	71.42	71.04
Standard Deviation Det. Time [ms]	7.78	7.47	8.02	2.88	2.87
Average Confidence [%]	0.70	0.71	0.66	0.87	0.87
Standard Deviation Confidence [%]	0.10	0.08	0.14	0.06	0.08

Table D.3: Analysis for Tester 3 at 40 cm

Metric	Base	Net	Noise	Loco_Noise	Loco_Net
Total Accuracy [%]	100.00	96.55	80.95	100.00	100.00
Average Detection Time [ms]	58.05	49.42	52.83	69.02	70.69
Standard Deviation Det. Time [ms]	6.92	9.75	8.40	4.16	2.93
Average Confidence [%]	0.80	0.76	0.68	0.78	0.77
Standard Deviation Confidence [%]	0.11	0.14	0.16	0.15	0.12

Table D.4: Analysis for Tester 1 at 100 cm

Metric	Base	Net	Noise	Loco_Noise	Loco_Net
Total Accuracy [%]	73.68	68.75	64.29	100.00	96.30
Average Detection Time [ms]	57.70	51.17	54.03	70.14	71.48
Standard Deviation Det. Time [ms]	6.80	7.48	6.37	5.06	4.00
Average Confidence [%]	0.72	0.65	0.69	0.73	0.77
Standard Deviation Confidence [%]	0.14	0.09	0.09	0.14	0.14

Table D.5: Analysis for Tester 2 at 100 cm

Metric	Base	Net	Noise	Loco_Noise	Loco_Net
Total Accuracy [%]	100.00	85.71	88.89	95.24	96.30
Average Detection Time [ms]	53.18	56.71	50.92	70.18	71.34
Standard Deviation Det. Time [ms]	8.56	10.18	9.06	4.76	3.10
Average Confidence [%]	0.70	0.68	0.68	0.84	0.84
Standard Deviation Confidence [%]	0.06	0.12	0.09	0.10	0.10

Table D.6: Analysis for Tester 3 at 100 cm

Ringraziamenti

Ci tengo innanzitutto a ringraziare le professoresse Michieletto e Reggiani per tutto l'aiuto, i preziosi consigli, il supporto ricevuto e le possibilità fornitemi.

Ci tengo a ringraziare anche il professor Minto, che mi ha gentilmente dedicato molto tempo durante lo sviluppo di una parte di questa tesi.

Un grazie in particolare a Mattia, Massimiliano, Paride, Pietro e Marco, che mi hanno aiutato con lo sviluppo della tesi, mi hanno ascoltato parlare a vanvera per mesi e hanno rallegrato le giornate in laboratorio.

Ringrazio Alberto, Filippo, Francesca, Matteo, Pietro e Sofia per avermi sempre aiutato quando non avevo voglia di studiare o non riuscivo a capire qualcosa. Sicuramente, senza di voi l'università sarebbe stata molto più noiosa e lunga.

Vorrei ringraziare i miei amici (Calcroci, un sabato da cretini) perché in questi anni hanno sempre sopportato.

Un grazie al gruppo Jappe, che periodo abbiamo vissuto insieme!

A big thanks to all the Erasmus friends I made in Sweden, those six months changed my life.

In particolare, però, il ringraziamento più speciale va ai miei genitori, perché senza i loro insegnamenti, la loro perseveranza e pazienza non sarei chi sono ora. Mi avete insegnato tutto quello che so e nulla potrà mai ripagare quanto avete fatto, e fate, per me.