UNIVERSITY OF GENOVA

MASTER PROGRAM IN ROBOTICS

Enhancing Disaster Response with PDDL: Quadruped Robot Planning in Search and Rescue Environments

by

Isabel Cebollada Gracia

Thesis submitted for the degree of *Robotics Engineering* (31° cycle)

October 2024

Zoe Betta	Co-Supervisor
Carmine Tommasso Recchiuto	Supervisor
Antonio Sgorbissa	Supervisor

Thesis Jury:

Antonio Sgorbissa, Giovanni Indiveri, Matteo Lodi, Nicoletta Noceti, Renato Zaccaria

Dibris

Department of Informatics, Bioengineering, Robotics and Systems Engineering

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Isabel Cebollada Gracia
October 2024

Acknowledgements

I would like to express my most sincere thanks to my co-supervisor, Zoe Betta, for her continuous guidance, support, and valuable insights throughout this period. I would also like to extend my sincere thanks to my supervisors, Carmine Tommasso Recchiuto and Antonio Sgorbissa, for their unwavering availability and assistance whenever I needed it.

Most importantly, I want to thank my parents, without whom none of this would have been possible. Thank you for always helping me, trusting me unconditionally, loving me, and supporting every decision I have made throughout my life. All of this hard work has been possible because of you.

I want to thank my friend Jesús, who has walked alongside me during all my life, continually supporting me. You always have the right words to brighten my day. Thanks to my friends Cintia and Sonia, who have always been available to listen and support me, sharing their trust in me and making me feel at home. I would also like to thank the friends made during this journey, Carmine and Bariş. They have become my family in Italy and made this trip something I will never forget. I am so lucky to share this experience with you all.

Lastly, but by no means least, I would like to thank all my grandparents for the sacrifices they made throughout their lives, allowing me to choose the path I wanted to pursue. I would also like to express my gratitude to my aunts, Isabel and Pili, for believing in me more than I have believed in myself.

I would like to express my appreciation to all the people who had a positive impact on my life, as well as all my professors for providing me with the necessary knowledge.

Thank you all, you have made me the person I am today. This also belongs to you.

Abstract

After a disaster caused by a catastrophe, the response time is very short due to the conditions in which the victims may find themselves, but at the same time, the safety of the rescuers must be prioritized. To address this need, this project proposes a system capable of providing useful information for rescuers and assisting in the search and rescue of individuals through the use of the Spot robot, a quadruped robot. This type of robot is one of the most commonly used robots for search and rescue situations due to its ability to carry heavy weights and its agility on complex terrains.

The system generates a sequence of actions for a successful search. As the robot moves, it provides an updated map that includes both static and dynamic obstacles, along with its own position. When the robot detects a person, it initiates an assessment that includes pose and consciousness estimation and a report of these status. If the person is considered conscious, the robot creates a new plan to guide them to the exit. Conversely, if the person is not conscious, the robot reports their status, including consciousness level, pose, identifier, and location. The robot can also return to charge if the battery is insufficient to execute the current plan.

In this way, rescuers gain knowledge of the situation, including how many people are in each area, their condition, and information about the obstacles present through the robot's map creation. The robot assists conscious individuals who are able to move, working alongside the rescuers and saving them time to focus on those who may need more assistance, allowing a quicker and safer performance.

The framework used to develop the system has been ROS2, despite the fact that the available tools were only on ROS1. This has lead to the use of Docker in order to use ROS2 due to hardware limitations and increase portability of the developed system.

We tested the proposed system in an indoor environment in both simulated and real scenarios, demonstrating good performance in terms of time while showing high reliability in assessing individuals. Thus, this solution helps rescuers by collaborating in the search for people and providing useful information for their interventions.

Table of contents

List of figures

Nomenclature

Acronyms / Abbreviations

AI Artificial Intelligence

AMCL Adaptive Monte Carlo Localization

BT Behavior Tree

COLIN COntinuous LINear numeric change

CTL Computation Tree Logic

DDS Data Distribution Service

DOF Degrees Of Freedom

EAP Enhanced Autonomy Payload

EOL End Of Life

FF Fast Forward

FOV Field Of View

FPS Frames Per Second

GPS Global Position System

GUI Graphical User Interface

GXP General Expansion Payload

HTN Hierarchical Task Network

IMU Inertial Measurement Unit

LiDAR Light Detection and Ranging

LPG Local Search for Planning Graphs

LTL Linear Temporal Logic

LTS Long-Term Support

OPTIC Optimal Temporal and Incremental Planner

PDDL Planing Domain Definition Language

POPF Partial Order Planning Framework

RCNN Region Based Convolutional Neural Networks

ROS Robot Operating System

RTAB-Map Real-Time Appearance-Based Mapping

RT Real Time

SAR Search and Rescue

SDK Software Development Kit

SLAM Simultaneous Localization and Mapping

SPP-Net Spatial Pyramid Pooling

SSD Single Shot Detector

STRIPS Stanford Research Institute Problem Solve

TAMP Task and Motion Planning

TCP Transmission Control Protocol

TFD Temporal Fast Downward

UAV Unmanned Aerial Vehicles

UDP User Datagram Protoco

UGV Unmanned Ground Vehicles

YOLO You Only Look Once

Chapter 1

Introduction

1.1 Context

Technology is continuously evolving and becoming increasingly integral to our lives. Various fields, such as robotics, have emerged over the years with the aim of assisting us in tasks that are either mundane or dangerous, always with the goal of enhancing the quality of our lives.

Each year, catastrophes ——whether natural or human-made— occur, impacting millions of people. These situations demand swift action with minimal errors, as the primary objective is to ensure the safety of all individuals in danger as quickly as possible. Achieving this often requires individuals to put themselves at risk to aid those in need.

As a result, there is a pressing need to enhance performance in this field through the use of systems that can provide safer and faster interaction. By employing Search And Rescue (SAR) robots, performance can be improved not only by assisting injured individuals but also by reducing the risks faced by workers in these situations. These robots play critical roles in disaster areas by locating victims, assessing their condition, facilitating their evacuation, and clearing obstacles, among other tasks.

Among the different types of SAR robots that exist (Ulloa et al. (2023)), Spot by Boston Dynamics is a notable example. Due to its quadrupedal morphology, it can navigate through uneven terrain and other complex environments efficiently. Ensuring that this robot can perform critical tasks in disaster situations, such as locating people and assessing their condition, not only improves the safety and efficiency of rescuers but also allows for a faster evaluation of the scenario, which is crucial in these situations.

Quadrupedal robots have been tested in various scenarios, such as in Ramezani et al. (2020), where the ANYmal quadruped was equipped with LiDAR to enable mapping capabilities for autonomous inspection. However, this system requires manual teleoperation initially

to build a foundational map of the environment, upon which the robot can autonomously navigate. Similarly, Spot's autonomous operation during the DARPA Subterranean Challenge is discussed in Bouman et al. (2020). While Spot demonstrated excellent mobility and autonomous exploration capabilities, it encountered limitations in complex environments, such as the need for remote control in certain areas due to sensor limitations and difficulties in autonomously mapping cluttered environments. In Queralta et al. (2020), Spot's integration into multi-robot search and rescue (SAR) systems is highlighted, outlining challenges in perception and coordination. The paper mentions difficulties in maintaining autonomous control, especially in environments with low visibility or heavy clutter. In such cases, human intervention is necessary for complex task coordination and to overcome obstacles.

Although Spot is currently being utilized for actuation in SAR operations, it still faces certain limitations. One key limitation is its inability to adapt to unexpected situations due to a lack of autonomous high-level reasoning. As a result, the robot struggles to prioritize actions based on their importance. Additionally, in complex environments, Spot often cannot navigate independently and requires teleoperation by humans. The aim of this project is to improve Spot's performance by incorporating Planning Domain Definition Language (PDDL). This enhancement will equip the robot with autonomous capabilities to prioritize tasks based on the dynamic environment and will enable it to operate fully autonomously, responding to complex scenarios without human intervention.

1.2 Objectives

The objective of this work is to develop a planner capable of computing an efficient sequence of actions that allow Spot robot to successfully perform search and rescue operations in post-catastrophe environments. The primary goal of the planner is to search for people in different locations and, if someone is found, assess and report their condition.

Another key goal of this project is to implement all functionalities in the latest version of the framework used, Robot Operating System 2 (ROS2), as all previous work has been done in ROS1. This transition would represent a significant and beneficial advancement for the future, as the old version will no longer be supported, paving the way for ROS2.

The final objective is to create the entire system withing a Docker container, providing excellent portability. This will allow any system to run the developed application with just a few commands, regardless of the operating system or its version.

The different actions chosen, taking into account the robot's capabilities and the given problem, are aimed at creating the most effective plan to achieve the desired behavior. These actions are as follows:

- **Move**: This will allow the robot to navigate between the different points in the environment.
- **Search**: Spot will look for humans.
- **Evaluate**: Each person found will be assessed to determine their condition.
- **Check battery**: This step ensures the battery is sufficient before continuing with the plan.
- **Report**: The status of each individual will be reported to the professionals, providing them with better knowledge of each person's location and condition.
- **Dialog**: A conversation will be initiated with the person found to assess their level of consciousness.

With these actions, the plan must be able to generate the most efficient sequence of steps based on the current knowledge, operating in the shortest possible time and providing the most accurate information about each person found. This enables professionals, such as firefighters or doctors, to take action if necessary, or alternatively, provide guidelines to the individuals if their condition is stable. The planner will adapt according to the current knowledge available to Spot, enabling it to create fast and accurate plans.

1.3 Project structure

The work is divided into 6 different chapters, structured as follows:

- Chapter 1: Introduction. This chapter provides a brief introduction to the thesis, offering context about the scenarios where the work is applicable and giving a general overview of its scope.
- Chapter 2: State of Art. This chapter discusses current methods used with SAR robots and their performance in disaster situations.

- Chapter 3: Materials. This chapter outlines the hardware used in the development of this thesis, including the robot and the integrated payloads. It also details the software tools utilized during development and explains how they were integrated into the system.
- Chapter 4: Actions and controller. This chapter provides an overview of the software architecture of the thesis, offering a detailed explanation of the PDDL actions and the controller that manages these actions. It also describes the tools utilized for each action and explains how they were integrated to ensure the proper functioning of the overall system.
- Chapter 5: Experiments. The chapter presents the experiments conducted and their results.
- Chapter 6: Conclusions. This chapter summarized the conclusions of the work and outlines potential future developments.

Chapter 2

State Of The Art

Summary

This chapter provides an overview of the current state of the art in search and rescue robots used for interventions in disaster situations, focusing on robots designed to search for and locate people who may be injured or in need of assistance due to natural or man-made catastrophes. It also highlights the various planning systems available and their respective advantages. Solvers for these planning systems are presented, along with an introduction and an explanation of situations they are best suited for. Finally, different navigation methods are explained, outlining their advantages, drawbacks, and the scenarios in which they are most effective.

2.1 Robots employed for search and rescue operations

After a catastrophe such as an earthquake or a fire, there is an urgent need to act quickly and effectively due to the dangerous conditions faced by those affected. Unfortunately, many lives are often lost during rescue operations because of the limited time available to act, leaving individuals trapped or without enough time to escape the disaster. According to of Labor Statistics (2017), 232 fire and rescue workers suffered fatal work injuries between 2011 and 2015.

These environments are typically characterized by uneven, unfamiliar terrain with numerous uncontrollable obstacles that can hinder escape or directly endanger lives. While professionals such as firefighters respond as quickly as possible, their first priority is ensuring their own safety before beginning rescue operations. This, combined with the challenge of

identifying where victims are located, makes it crucial to prioritize rescuing those who are still alive. However, this is not always easy, as finding survivors quickly can be difficult if they are trapped under debris or are severely injured.

Therefore, despite the best efforts of rescuers, additional technological tools are necessary to improve performance and minimize the number of fatalities and injuries. By integrating technology that can work alongside rescuers, providing vital information on the locations and conditions of victims, robots play a crucial role in search and rescue operations.

In these tasks, robots can be either teleoperated, Marques et al. (2007) or autonomous. In both cases, robots must prioritize human safety, ensuring that their actions do not pose additional risks to people (Dadheech (2022)). Teleoperated robots require reliable communication between the controller and the robot itself to transmit commands and information efficiently. This creates latency problems, that can be avoided if the operator is close to the robot, but this entails safety risk for the operator. However, when using a teleoperated robot, there is still a chance that the operator may miss a victim due to poor lighting or obstacles. On the other hand, autonomous robots must be equipped with capabilities such as terrain navigation, obstacle avoidance, human recognition and decision-making based on the situation; this requires a very advanced system with high computational power and reliability.

Different types of robots are suitable for varying environments and conditions. One category is underwater robots (Wang et al. (2022)), which are particularly useful in drowning scenarios, one of the most common causes of accidental death worldwide (WISQARS (2022)). The need for robots in such situations is high due to the short window of time for action and extreme conditions like temperature or low visibility. Underwater robots are equipped with wireless communication systems to rely information to the surface and high-resolution cameras for quick and precise identification. These robots send real-time data to rescuers on the surface, enabling them to make informed decisions. However, they are specifically designed for search and rescue operations in water environments and are limited to that context.

Another type of robots used in search and rescue situations are aerial robots, more commonly known as Unmanned Aerial Vehicles (UAVs) (Lyu et al. (2023)). The most well-know UAVs for this task are drones, which, equipped with various sensors, are capable of performing SAR operations over large areas. Due to their flying capability, drones can cover vast regions quickly, making them particularly useful for SAR operations in outdoor environments where speed is critical.

Among the various sensors drones can be equipped with, a localization system, typically GPS, is essential due to their operation in large, outdoor environments. For object recognition,

drones rely on high-resolution cameras capable of performing either one-step or two-stage detection models (Mishra et al. (2020)). One-step detection models process images directly, often resulting in faster performance. Examples of such models include You Only Look Once (YOLO) and Single Shot Detector (SSD). In contrast, two-stage detection models first generate thousands of object proposals from the input image. These proposals are then analyzed by a neural network to extract features, with a classifier determining the presence of objects based on the extracted information. Some examples of two-stage methods are Region Based Convolutional Neural Networks (RCNN) and Spatial Pyramid Pooling (SPP-Net). Additionally, drones are also equipped with thermal cameras (Yeom (2024)), which improve the ability to locate individuals in large areas quickly. Once victims are detected, precise localization is required. For this purpose, drones use Light Detection and Ranging (LiDAR) sensors for precise localization. Additionally, Real-Time (RT) processing is crucial for these operations to ensure that information is received and acted upon as quickly as possible.

Another category of robots used in SAR operations are Unmanned Ground Vehicles (UGVs), also known as ground robots. While they are generally slower than drones, UGVs are better suited for indoor environments where precise navigation in confined spaces is essential. Ground robots can be further classified into three subtypes: wheeled, tracked and legged. In addition to their suitability for confined environments, UGVs offer advantages in terms of endurance and load capacity. Due to their larger battery capacity, they can operate for extended periods without needing recharges, which makes them ideal for long-duration missions. Moreover, their ability to carry heavier payloads enables them to transport crucial rescue equipment or supplies, adding to their overall effectiveness in search and rescue operations. These features make UGVs a versatile and reliable choice for navigating complex terrain and performing essential tasks in areas that result difficult to reach.

Wheeled robots (Chung and Iagnemma (2016)) are known for their speed, simplicity, robustness and maneuverability. However, their mobility is limited by the wheel-terrain contact interface. There are four primary combinations of wheel and terrain types: rigid wheels on rigid terrain, rigid wheels on deformable terrain, deformable wheels on deformable terrain and deformable wheels on rigid terrain. However, the effectiveness of navigation varies depending on these combinations and can be further influenced by obstacles such as ramps or stairs on the terrain.

Tracked robots offer greater mobility on uneven, debris-filled or muddy terrains. Their tracks allows them to traverse difficult ground and climb over obstacles such as rocks or fallen structures, which are very common in disaster scenarios. Although slower than wheeled

robots, tracked robots are better equipped to handle the unpredictable and varied terrains typically found in disaster situations.

Both wheeled and tracked robots have their advantages and drawbacks, and the decision on which to use depends on the context of the mission. However, legged robots offer significant advantages over the other two types due to their ability to navigate diverse terrains. Legged robots may be composed by a different number of legs, depending mainly by the purpose of the robot, but one of the most common configuration is quadrupedal robots. They are popular for SAR missions because of their dog-like physical structure, allowing them to climb stairs, move through narrow and low-height spaces and traverse rubble, mud, or water. Their versatility and adaptability to various terrains make them the preferred choice in environments where ground conditions are uncertain due to disaster damage.

Ground robots are equipped with a range of sensors to facilitate SAR operations, including LiDAR, high-resolution cameras, speakers and thermal cameras to ensure efficient performance in challenging conditions.

In summary, for search and rescue operations in disaster situations such as earthquakes, ground robots are generally more suitable compared to other types of robots. Underwater robots are limited to water environments and are not applicable in land-based disasters. Drones excel in open-field scenarios where rapid are coverage is crucial for locating victims, but they may lack precision and struggle in confined or complex environments. In contrast, ground robots can navigate more carefully and effectively in unfamiliar and cluttered environments, such as those with low ceilings, narrow passages and debris. Among ground robots, legged robots are considered the best option (Li et al. (2023)) due to their superior terrain adaptation capabilities and promising applications. Quadrupeds, in particular, can be equipped with heavy sensory systems (Cruz Ulloa et al. (2023)), allowing them to traverse challenging terrains and fit into tight spaces.

For this project, the quadrupedal legged robot Spot from Boston Dynamics (BostonDynamics (2024b)) was selected. Spot is well-suited for the task due to its versatile capabilities. It can carry various payloads to perform different functions, be remotely controllable or operate autonomously with an onboard computer. Additionally, its legged design allows it to efficiently navigate complex and narrow environments, making it ideal for the intended search and rescue operations.

2.2 Planning Languages

Planning languages are systems used in Artificial Intelligence (AI) and robotics to represent and solve planning problems. They provide a structured framework for specifying actions, states, goals, constraints and predicates, which are essential for generating sequences of basic procedures to achieve specific objectives. These languages are designed to facilitate the automation of decision-making processes, allowing systems to perform complex tasks efficiently and effectively, thus enabling autonomy in task execution.

There are several types of planning languages, each suited to different problem objectives. One of the earliest planning languages is the Stanford Research Institute Problem Solver (STRIPS) (Geffner (2000)). STRIPS is limited by certain constraints: it defines actions only through preconditions and effects, and it only allows simple goal statements. Although STRIPS lacks many features of modern planners, such as the use of types, temporal planning or current actions, it has served as the basis for the creation of many modern planning systems and languages.

Another type of planning language is Task and Motion Planning (TAMP) (Garrett et al. (2021)), a framework that integrates task planning and motion planning to address problems involving both high-level goals and low-level actions. TAMP requires detailed knowledge of both tasks and motion planning, making it complex and computationally expensive. Additionally, its modularity can be compromised due to the tight coupling between tasks and motions. This language is particularly well-suited for manipulator robots, where specific trajectories must be executed according to a plan. However, it is less appropriate for situations requiring high-level goals with well-defined actions that do not go into the specifics of motion.

Hierarchical Task Network (HTN) Planning (Georgievski and Aiello (2015)) is another language that decomposes complex tasks into smaller and manageable subtasks. HTN is tailored for domains where hierarchical decomposition is natural, but this can make the language less flexible and harder to generalize across different domains. While HTN is effective in cases requiring task decomposition, it can be time-consuming in situations where decomposition is unnecessary. HTN is procedural, focusing on how to plan tasks, and relies on a predefined set of methods, which limits its suitability for dynamic environments that require frequent replanning.

Another planning language is Linear Temporal Logic (LTL), which is more of a formalism than a traditional planning language. LTL is used for specifying and reasoning about the temporal properties of a system. It extends classical propositional logic by introducing temporal operators such as *next*, *globally*, *finally* and *until*. These operators allow for

the expression of properties that describe how a system's state evolves over time. LTL formulas are interpreted over paths to determine which formulas are satisfied. Due to these characteristics, LTL is widely used in model checking, where a system model (typically a finite state machine) is verified against a specification (an LTL formula). In this context, defining a goal may not be as intuitive as it is with other languages, as it involves reasoning about the system's behavior over time rather than specifying goals directly.

Computation Tree Logic (CTL) (Dal Lago et al. (2002)) is another temporal logic, like LTL, used to define system properties. Unlike LTL, CTL allows reasoning about multiple possible future paths from any given state, making it useful for systems represented by state-transition graphs.

Finally, Planning Domain Definition Language (PDDL) (Ghallab et al. (1998)) is a standardized and widely-used language in AI. PDDL separates the planning problem into two files: the domain and the problem. The domain file describes the environment through objects, things that are present in the environment, predicates, properties that can be associated to objects, actions, how the environment can be modified. The problem file specifies the initial state as the objects in the environment and their state through the definition of predicates that are true, it also defines the goal state, as the set of predicates that must be true at the end of the task. The actions in the domain file are defined through different conditions and effects such as preconditions, predicates that must be satisfied for that action to be executed, conditional effects, where a consequence is applied when a predicate is true, or numeric fluents, variables which apply to zero or more objects and maintain a value throughout the duration of the plan. PDDL, an evolution of STRIPS, allows more complex planning including concatenated goals, conjunctive and disjunctive actions and more. We chose it as the planning language for this thesis due to its advantages over other planning languages:

- **Standarization**: PDDL is the standard in AI planning, providing widespread support, collaboration and a large community.
- **Modularity**: Due to its syntax, PDDL offers modularity and reusability of domain files across different problems.
- **Types definition**: PDDL allows typing, enabling hierarchical structures among objects and facilitating action definitions constrained by types.
- **Complexity**: It supports the definition of complex problems, including negative predicates, numeric fluents, durative actions and complex goals with multiple predicates.

- **Replanning**: PDDL enables replanning, which is necessary in dynamic environments where conditions may change, requiring updates to the plan.

In summary, PDDL offers significant advantages over the other languages discussed. As an evolution of STRIPS, it introduces additional features such as hierarchies, types and complex preconditions and effects. While HTN focuses on task decomposition, PDDL provides a more general framework that does not rely on predefined hierarchies and allows for replanning. Unlike TAMP, PDDL does not require direct integration with motion planning, which is unnecessary for the scope of this project. For the same reasons, PDDL is a better choice than CTL and LTL, as it is specifically designed for planning rather than for reasoning about temporal properties or model checking.

2.3 Solvers

PDDL supports various solvers, each suited to different domains and problem definitions. Selecting the appropriate solver depends on the specific needs of the domain and the problem being addressed.

Fast Forward (FF) (FF (2024)) is an heuristic-based solver that uses forward search techniques combined with heuristics to generate plans. It starts from the initial state, explores possible actions and states to reach the goal, and uses heuristics (commonly based on the relaxed planning graph) to estimate the cost of reaching the goal from a given state. While FF is fast and efficient, its heuristic-based approach may result in suboptimal plans, as it prioritizes the speed of solution over plan quality.

Local Search for Planning Graphs (LPG) (Gerevini and Serina (2002)) is another classic planner that applies local search algorithms to planning graphs in order to generate plans. LPG iteratively refines candidate plans by making small adjustments and evaluating their impact, using heuristic functions to guide the search. LPG also handles numerical and temporal constraints, making it useful in contexts where timing or resource management is critical.

COntinuous LINear numeric change (COLIN) (Coles et al. (2014)) is a solver designed for planning problems involving continuous and linear numeric changes, making it useful in contexts that require such variable handling. Although it is effective for these problems, COLIN can become computationally complex in large-scale scenarios.

METRIC-FF (Metric-FF (2024)) is an extension of FF that incorporates metric features. Like FF, it uses forward search with heuristic functions to efficiently reach the goal, but it

also supports numerical and temporal constraints. METRIC-FF employs a metric planning graph, which aids in resource management and optimization.

Optimal Temporal and Incremental Planner (OPTIC) (Optic (2024)) is a temporal planner that focuses on generating optimal plans by managing time and resources. It is specifically designed for handling time constraints and can incrementally adjust plans as new information becomes available. However, OPTIC may be less effective in scenarios with fewer temporal constraints.

The final temporal planner solver discussed is the Partial Order Planning Framework (POPF) (Chen and Pan (2024)). As it name suggests, POPF uses partial order planning techniques to generate plans where the sequence of actions is not fully specified but is constrained by a set of precedence relations. This approach allows for flexibility in the order of actions while ensuring that all constraints are satisfied. POPF employs heuristic search techniques to estimate the cost of reaching the goal and guides the search process to find plans more efficiently. It is well-suited for complex scenarios, as it can handle temporal and numeric constraints. However, it may encounter issues related to computational complexity when dealing with large-scale problems.

Each solver offers different features that might be better suited depending on the context in which they are used. However, POPF has been chosen as the solver for the planning part of this project due to several features that make it particularly well-suited for search and rescue operations in disaster scenarios:

- **Flexibility**: POPF allows for flexible action sequencing, which is crucial in SAR operations where the environment is constantly changing and new information may emerge during the planned execution.
- **Handling complex constraints**: POPF manages both temporal and numerical constraints, essential for SAR where time and resource management are critical.
- **Efficiency**: POPF's heuristic search enables it to find feasible plans quickly, which is critical in time-sensitive operations.
- **Plan optimization**: POPF's optimization capabilities allow for cost minimization and effective resource management.

2.4 Navigation

Navigation is a key component in robot performance, as it directly affects movement within the environment. How the navigation is implemented can vary based on the design, environment and intended application.

One of the most common navigation methods is the line-following algorithm, where the robot adheres to a predefined path typically marked by a line, usually of a contrasting color (often black), on the ground. These robots are equipped with infrared sensors that detect the color contrast between the line and the surrounding ground. The microprocessor uses this information to adjust the robot's direction if the line moves off-center. This approach is a basic form of navigation commonly used for educational purposes or in factory for transporting goods between different points.

Navigation usually comprises also obstacle avoidance algorithms, which are crucial for robots operating in dynamic environments. Robots are equipped with sensors —such as ultrasonic, infrared, bump sensors, cameras, or LiDARs— that detect nearby obstacles. This allows robots to react, based on the sensor input, by adjusting its movement based on the detected obstacle position.

Path panning algorithms, together with obstacle avoidance ones allow a robot to navigate to a goal position, and orientation, in space. An example is the vector fields algorithm, which creates a histogram grid based on sensor data and obstacle density, guiding the robot to navigate towards the goal and avoiding obstacles.

Several path-planning algorithms are grid based and they require an environment represented as a grid map with each cell marked as occupied, free or unknown, this is called a grid-map. Based on this grid, a cost map is created reflecting the difficulty or proximity to unknown space or obstacle cells. Path-finding algorithms such as A*(Ju et al. (2020)) or Dijkstra (Wang et al. (2011)) are used to calculate the shortest path between two points. A* is generally more effective than Dijkstra in large grids because it incorporates cell movement costs, whereas Dijkstra finds the shortest path without considering these costs. This method requires the initial position of the robot to be known or manually set and is not well-suited for environments with dynamic obstacles. Therefore, grid-based navigation is commonly used for indoor applications, such as in vacuum cleaners or service robots, where the environment is more predictable and there is not the need to account for the presence of humans moving around.

GPS-based navigation is another technique used by robots to determine their position on the Earth's surface and navigate to specific locations using the Global Positioning System

(GPS). This method relies on a constellation of satellites orbiting the Earth, which constantly broadcast signals containing their time and position. The robot, equipped with a GPS receiver, collects information from multiple satellites and calculates the time difference between the sending and receiving of signals. This allows the robot to determine the distance to each satellite. By using triangulation with these distances, the robot can pinpoint its exact position (latitude, longitude, and altitude). The calculated coordinates are mapped onto a digital map, enabling the robot to always know its location relative to the map. The robot then navigates by moving towards predefined waypoints specified by GPS coordinates. During movement, it continuously recalculates its position and adjusts its path accordingly. Although GPS-based navigation is very accurate and cost-effective, it is primarily used outdoors due to GPS signals' inability to penetrate structures, but it is well-suited for covering large areas.

Behavior-based navigation emphasizes the use of independent behaviors to achieve navigation tasks. It is organized into multiple layers, with each layer corresponding to different behaviors: lower layers handle basic functions, while higher layers manage more complex tasks. The robot selects the behavior to follow based on current sensor inputs, and this behavior is updated as the inputs change. When multiple behaviors are active, the system prioritizes them according to predefined rules.

Given the project's focus on developing an indoor search and rescue system, where the environment is unknown and dynamic obstacles (both people moving in the environment, and debris that are not present before the mission) are present, grid-based navigation techniques have been employed for global path planning, working in conjunction with costmaps, which are grid-based structures that, differently from the grid maps, represent the environment with values indicating the cost of traversing the area, with high values corresponding to a high risk of collision. However, since these techniques are limited in handling dynamic obstacles, a local planner has also been utilized to incorporate real-time obstacle avoidance. In parallel, the robot's position is continuously estimated using SLAM techniques, specifically through the use of RTAB-Map (Labbé and Michaud (2018)), which will be further explained in the next chapter.

Simultaneous Localization and Mapping (SLAM) (Durrant-Whyte and Bailey (2006)) is a vital technique in robotics and autonomous systems that enables robots to create a map of an unknown environment while simultaneously tracking their own location within that map. By using sensors such as Inertial Measurement Units (IMUs), cameras or LiDAR, the robot can estimate its position and orientation. This estimation is continuously updated as the map is constructed. The mapping process involves representing the environment in a grid (either 2D or 3D) where each cell is marked as free, occupied or unknown based on

sensor data. Localization and mapping are correlated, meaning that accurate performance in one relies on accurate performance in the other. Consequently, both tasks are performed iteratively to refine the map and the robot's position and orientation. Although SLAM can be computationally intensive, it is needed for autonomous robots operating in unknown or dynamic environments.

Chapter 3

Materials

Summary

This chapter outlines the materials used in the development of the project, encompassing both hardware and software. It provides a detailed overview of the chosen robot and its capabilities. Additionally, the integrated payloads are discussed, highlighting the additional functionalities they offer to enable the robot to accomplish the designated tasks.

3.1 Spot from Boston Dynamics

The robot used for this thesis is the Spot (Figure 3.1), developed from Boston Dynamics. Spot is a versatile and agile quadruped robot designed for a wide range of applications, including research purposes.

Figure 3.1 Spot robot from Boston Dynamics.

Spot's dimensions are 1100 mm in length and 500 mm in width, which remain fixed, while its height is variable. When sitting, Spot's height is 191 mm, and during movement, it can range from a range between 520 mm to 840 mm, with a default walking height of 610 mm. It is important to note that these dimensions refer to the robot alone, without any attached payloads.

It can not only modulate its height, but it can also rotate through the hips and knees (Figure 3.2) along the roll, pitch and yaw axes. It has two actuators in each hip and one in each knee, giving it a total of 12 Degrees Of Freedom (DOF). The movement limits are $\pm$ 45 degrees for the X-axis rotation in the hip from vertical, $\pm$ 91 degrees for the Y-axis rotation in the hip with a 50 degree bias from vertical, and a flexion-extension range from 14 to 160 degrees for the knee. This modulation allows Spot to adapt its posture according to the environment, enabling it to navigate through narrow passages or low-height spaces.

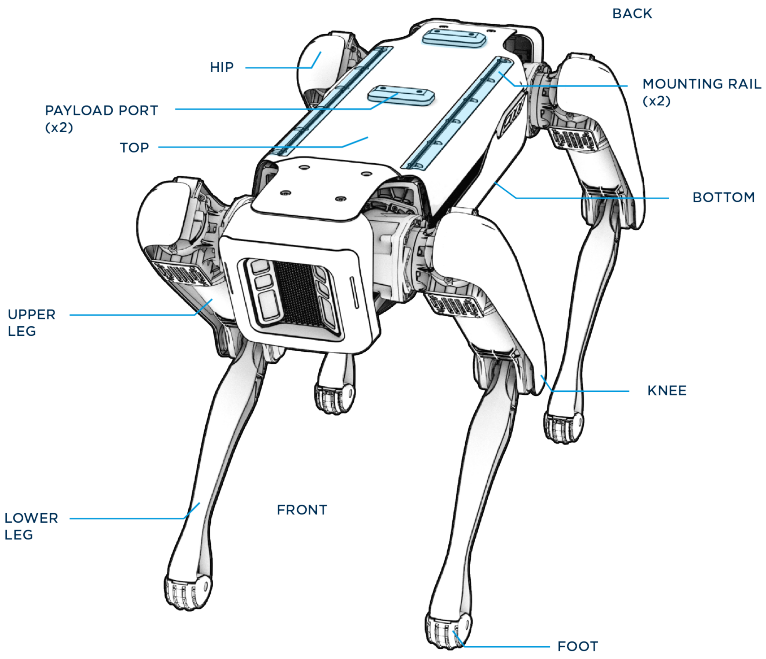


Figure 3.2 Spot anatomy.

The weight of the robot is 31.7 kg without payloads and including the battery, which weights 4.2 kg. The battery has a capacity of 605 Wh, providing an average runtime autonomy without payloads of 90 minutes when the robot is moving, and around 180 minutes if the robot is on standby mode with the motors turned off. The battery takes approximately 120 minutes to recharge.

Due to its quadrupedal design, Spot can navigate through complex terrain, making it an excellent choice for search and rescue operations where surfaces are often uneven and obstructed with obstacles. Additionally, Spot can walk on sloping surfaces within a range of $\pm 30º$, allowing it to access locations that other robots might not reach. Furthermore, Spot is capable of climbing stairs. The stair step and riser dimensions must be within 175 x 255-280 mm, with a maximum step height of 300 mm, enabling the robot to navigate across multiple floors.

The robot's operating temperature range is between -20ºC and +45ºC, and its maximum speed is 1.6m/s.

Spot is equipped with five black-and-white stereo cameras that provide a 360-degree field of view and a detection range up to 4 m, requiring a minimum light level of 2 Lux. Two of these cameras are positioned at the front, one on each side, and one at the back of the robot. Spot uses this perception system to automatically avoid collisions with obstacles, allowing it to navigate without hitting dynamic obstacles, though effectiveness can vary depending on light levels and the visibility. Additionally, Spot features a safety tool called E-stop, which prevents the robot from moving until one or more pre-determined clients confirm that they can communicate with the robot by sending a special message.

Spot is teleoperable via a tablet controller connected to the robot through a WiFi connection operating at a frequency of 2.4 GHz. 2.4GHz. The WiFi connection can be implemented either by having the tablet connected directly to a network generated by the robot or by connecting both the robot and the tablet to the same network. The controller (Figure 3.3) features two joysticks, a d-pad, four mode buttons, two trigger buttons and a 7-inch touchscreen. Using this controller, commands can be sent to the robot, enabling it to sit, stand, move, turn or rotate along its yaw pitch and roll axes.

In situations where the battery is very low, Spot is equipped with a safety feature that automatically sits it down before shutting it off if it was standing. This precaution helps protect the robot and its attached payloads from potential damage.

Extra payloads can be attached to the robot to enhance its functionality. These payloads can be classified into three types: passive, active and radiating payloads. Passive payloads are non-actuated attachments used for sensing and signal processing, drawing power from the Spot's power supply, for example an external camera. Active payloads are attachments that incorporate actuated motion independent from the robot's movement, allowing for separate control. Their power source can be either the Spot's power supply or an independent source, such as with the Spot arm. Radiating payloads are those that unintentionally emit radiation,

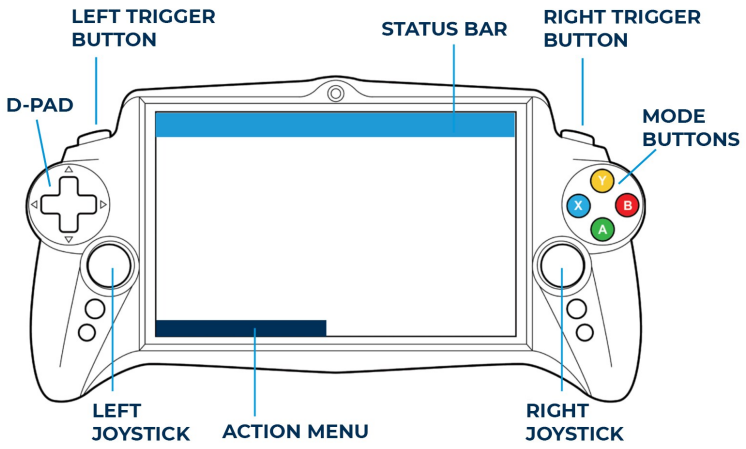


Figure 3.3 Spot tablet controller.

such as radio transmitters. Passive payloads have been integrated into the robot to enhance its capabilities, and these will be detailed in the next section.

3.2 Payloads from Boston Dynamics

Even though Spot can operate effectively with its built-in capabilities, additional payloads can be attached to enhance its functionality for specific tasks and information gathering. While users have the flexibility to add various payloads, Boston Dynamics offer pre-approved payloads that are designed to meet all operational requirements and are ready for use with the robot BostonDynamics (2024a). Spot is equipped with two ports for connecting two different payloads. These ports provide power, communication, time synchronization and safety system integration via a DB25 connection. It is important to note that Spot will not function with a payload unless it is properly connected through these ports, and the robot must have a battery installed for the payloads to operate.

The total combined payload capacity distributed on top of the robot is 14 kg and must not be exceeded. This weight includes all attached payloads. For optimal performance and stability, the combined center of mass of the payloads should be positioned between the front and rear hips, which enhances Spot's agility and reduces the risk of tipping over. Regarding payload dimensions, the maximum recommended width is 190 mm. Exceeding this width may interfere with the robot's legs and reduce its overall mobility. The maximum allowable length for payloads is 850 mm, and payloads should not extend beyond the front or the rear of the robot to avoid diminishing maneuverability. Additionally, the height of the payload

affects the robot's ability to stand and raises the center of mass. Payload height should be kept as low as possible to avoid negatively impacting the robot's movement.

The payloads from Boston Dynamics used in this project is the Spot Enhanced Autonomy Payload (EAP) BostonDynamics (2022). This payload integrates a Spore CORE module with a LiDAR assembly, significantly enhancing the robot's sensing capabilities.

The Spot CORE (Figure 3.4a) provides computing capabilities, network connectivity and data interfaces. It weights 2000 g and has dimensions of 250 mm in length, 190 mm in width and 84 mm in height. It operates within a temperature range of 0ºC to 50ºC and hosts the software necessary for the LiDAR service to interface with the robot. The Spot CORE is composed of:

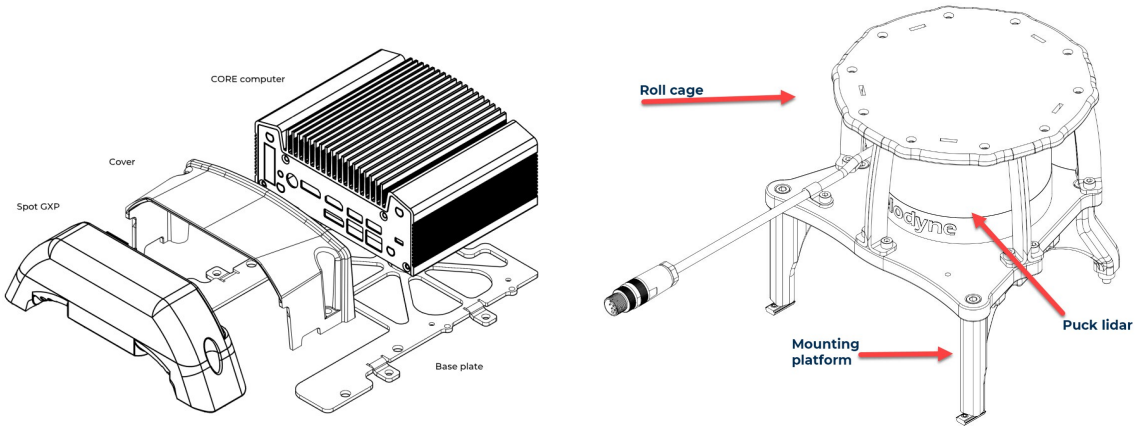
- Spot CORE computer running Ubuntu 18.04 Long Time Support (LTS) Desktop with 16 GB RAM and 512 GB storage.
- Spot General Expansion Payload (GXP), which provides regulated power and data interfaces.
- Spot CORE cover that protects the CORE computer and the GXP.
- Base plate.
- LiDAR cable that connects the CORE computer and the GXP to the LiDAR payload.

On the other hand, the LiDAR sensor (Figure 3.4b) enhances the robot's perception from a depth of 2-4 meters to approximately 120 meters. The LiDAR used is a VLP-15 LiDAR sensor, which has an accuracy of up to ± 3 cm and operates within a temperature range of -10ºC to 60ºC. This sensor is highly beneficial for navigation tasks in unknown environments, as it provides information about the surroundings from a greater distance, overcoming the limitations of a 4-meter range of the onboard cameras, that can restrict navigation in dynamically changing environments.

The EAP —which combines both the CORE and the LiDAR— (Figure 3.5) is mounted on the rear part of the robot and connected to the rear payload port. The connection between the CORE and the LiDAR is made via an M12 LiDAR cable, which provides the necessary data and power communication. The total weight of the EAP is 3.6 kg.

3.3 Custom Payloads

Besides the payloads that Boston Dynamics offers for direct integration with the robot, users can add customized payloads to extend the robot's sensing capabilities for different tasks.



(a) Spot CORE payload. (b) Spot LiDAR payload.

Figure 3.4 Boston Dynamic payloads used in the project.

Figure 3.5 Spot Enhanced Autonomy Payload.

Specifically, this project includes an additional payload: the ZED2 camera from StereoLabs StereoLabs (2024). The ZED2 camera weights 166 g and has dimensions of 174.9 mm in length, 29.8 mm in width and 31.9 mm in height. It operates within a temperature range of -10ºC to 45°C.

Compared with Spot's stereo cameras, which provide black and white images with a resolution of 640x480, the ZED2 camera is a stereo camera that offers high-definition video and RGB images. It supports various resolutions and frame rates: 2208x1242 at 15 Frames Per Second (FPS), 1920x1080 at 30 FPS, 1280x720 at 60 FPS or 672x376 at 100 FPS. Designed for depth perception, motion tracking and spatial AI, the ZED2 enables advances applications such as object tracking, object detection, spatial mapping, global localization, positional tracking, and neural depth sensing. Its Field Of View (FOV) is wide, with 110º horizontally, 70º vertically and 120º diagonally. The depth range extends from 0.3 m to 20 m, with an accuracy of less than 1% up to 3m and less than 5% up to 15 m (Abdelsalam et al. (2024)).

The ZED2 is connected to a NVIDIA Jetson computing board, which incorporates algorithms from the ZED SDK. This setup allows the camera to utilize the available features and perform AI tasks.

The camera is attached to the robot using a custom holder made with a 3D printer. This holder secures the NVIDIA Jetson, the camera and a power bank that serves as a power supply for the Jetson. The camera and the Jetson are connected via a cable, and the NVIDIA Jetson is connected to the Spot CORE through an Ethernet cable. This connection allows communication between the robot and the camera via the IP address 10.0.0.2. Access to the camera is initiated via an SSH connection from the Spot CORE, but the actual communication is handled through sockets, allowing for the control and use of camera tools on the NVIDIA Jetson. The various socket communication methods will be thoroughly detailed in Section3.7.

It is important to note that the camera has a different reference frame compared to the robot, so this must be considered when interpreting the information from the camera. Spot with all the attached payloads can be seen in Figure 3.6.

Figure 3.6 Spot with all payloads attached.

3.4 ROS2

Robot Operating System2 (ROS2) ROS2 (2024a) is an open-source and flexible framework designed for robot software development. It offers a comprehensive set of tools and libraries to enable developers to build sophisticated robotic systems. Key features of ROS2 includes:

- Middleware abstraction: ROS2 abstracts the underlying communication layer using the Data Distribution Service (DDS). This allows processes to communicate over a reliable network based on the application needs.
- Node-Based architecture: ROS2 employs a modular design based on nodes. Each node performs a specific function, which facilitates maintenance, re-usability and the development of more complex systems.
- Real-time support: ROS2 includes features that enable the real-time operation, making it suitable for applications where precise timing is critical.
- Publish/Subscribe messaging: ROS2 provides an asynchronous messaging model that enhances scalability and flexibility. Nodes can publish data on specific topics, while other nodes can subscribe to these topics to receive the data.

- Services and actions: In addition to topics, ROS2 offers services and actions for node communication. A service consists of a pair of messages (request and response) where a service client node requests information from a service server node. Actions are used for more complex tasks, allowing a node to send a goal to another node and receive results, status updates and feedback.
- Life-cycle management: ROS2 provides mechanisms for managing the states of nodes, which simplifies their management.
- Cross-platform compatibility: ROS2 supports multiple operating systems, including Windows, macOS and Linux.
- Security features: ROS2 includes encrypted communication and authentication mechanisms to ensure secure and confidential data exchange between nodes.

ROS2 is the successor of ROS1 ROS (2021), which has been and is still widely used in robotics research and development. Created in 2007, ROS1 has seen numerous updates and new versions released annually. However, significant features were either missing or in need of improvement. Incorporating these changes would have required extensive modifications, potentially destabilizing ROS1. As a result, a new version, ROS2, was developed to address these issues and provide the needed enhancements.

Currently, the last ROS1 version is ROS Noetic Ninjemys which was released in 2020 and has an End Of Life (EOL) scheduled for 2025. After this version, no further updates or releases for ROS1 will be made, meaning support will eventually cease. In contrast, the latest ROS2 version is ROS2 Jazzy Jalisco, but this project used the penultimate version, ROS2 Humble Hawskill. It has an EOL date of May 2027, and unlike ROS1, future versions will continue to be released.

This is one of the main reasons for developing the project using ROS2. The Spot Core operates with Ubuntu 18.04, which only supports ROS1 Melodic. Even though additional adaptations were needed to enable the use of ROS2 with the Spot robot, it was the preferred framework for developing the project due to its long-term support (LTS), better distributed system support, and improved security.

3.5 Docker

As mentioned in the previous Section 3.4, the Spot Core operates with Ubuntu 18.04, which only supports ROS1 Melodic for code development. To use ROS2 Humble, it is necessary to set up a Docker container running Ubuntu 20.04, which allows the installation of the desired ROS2 version.

Docker (2024) is an open platform that enables the packaging and execution of applications within a isolated environments known as containers. These containers include all the necessary components to run the application, allowing multiple containers to operate simultaneously on the same host machine without requiring additional installations. Furthermore, containers can be packaged into images and shared via Docker Hub, making it easy to deploy and use them on different machines with Docker installed.

An image is a read-only template that contains the instructions for building a Docker container. Typically, images are created by starting with a base image and then adding additional customizations to achieve the desired configuration.

A container is a runnable instance of an image. It can be created (only once), started, stopped, moved, or deleted using Docker API commands. Containers can be run with customized flags depending on the application's needs and can also generate new images based on their current state. A container is defined by its image and any configuration options specified during its creation or startup. Once a container is removed, any changes to its state that are not stored in persistent storage are lost.

Docker operates using a client-server architecture (Figure 3.7), where the client interacts with the Docker daemon. The daemon is responsible for building, running, and distributing Docker containers. The client and daemon can run on the same machine or be connected remotely. They communicate using a REST API over UNIX sockets.

For this project, it is crucial that the Docker container includes a Graphical User Interface (GUI) to visualize information such as the robot's navigation map and obstacles detected during exploration. To enable GUI support, you first need to execute the command `xhost +` in the Spot core terminal. This command allows all users to connect to the X server and display graphical applications on the screen. Once this is set up, you can run the Docker image using the command in Listing 3.1, where docker_name is the desired name for the container and image_name is the name of the image from which the container will be created. Note that this command needs to be executed only once to create and start the container. For subsequent launches, you can use the command `docker start docker_name` to start

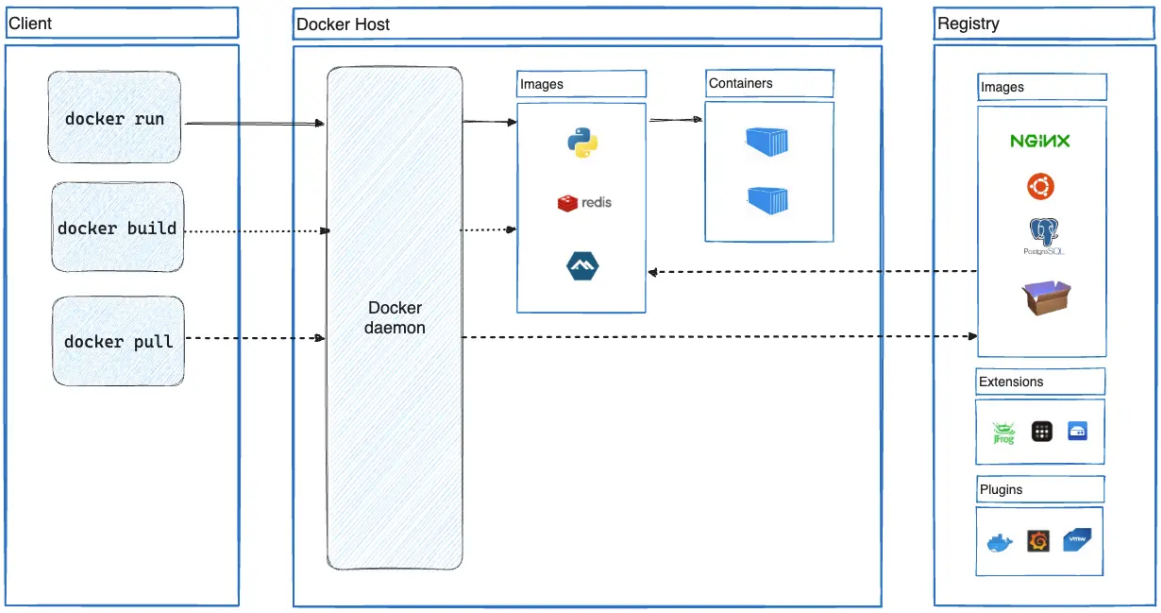


Figure 3.7 Docker architecture.

the container. To access the running container from different terminal windows, use the command `docker exec -it docker_name /bin/bash` in the secondary terminal.

```
sudo docker run -it --name=docker_name -v /tmp/.X11-unix:/
    tmp/.X11-unix -e DISPLAY=unix$DISPLAY -e GDK_SCALE -e
    GDK_DPI_SCALE --net=host --privileged image_name
```

Listing 3.1 Docker run command.

3.6 Spot Software Development Kit (SDK)

To facilitate the development of software applications and solutions for the Spot robot, Boston Dynamics provides a toolkit that allows developers to interact with the robot. The SDK BostonDynamics (2024c) offers the necessary tools and interfaces to control Spot, access its sensors, and integrate it with other systems. The Spot SDK is composed by documentation, the Python client library, the payload documentation and the Spot API.

The Python client library for Spot is supported on platforms macOS 10.14 (Mojave and Catalina), Windows 10 and Ubuntu 18.04 LTS. It is compatible with Python versions 3.6 through 3.10.

The Spot API (Figure 3.8) enables applications to control the robot, access sensor data and integrate payloads. It operates on a client-server model, where client applications interact

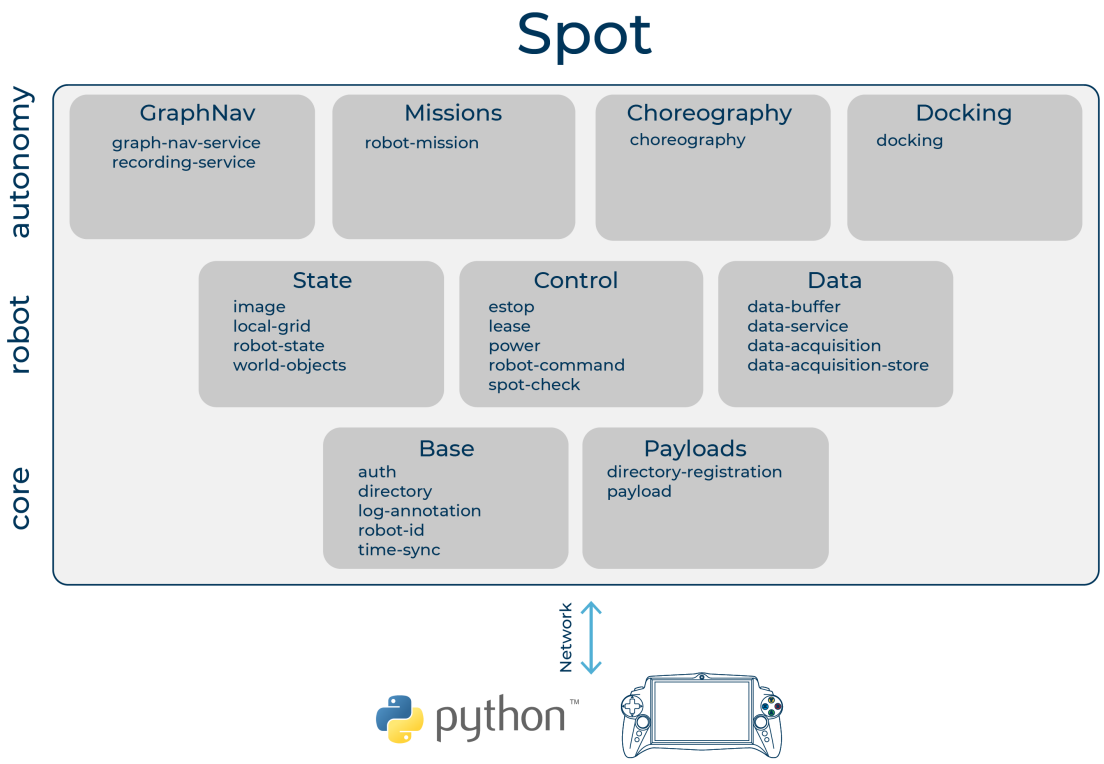


Figure 3.8 Spot API.

with services running on the Spot robot over a network connection. This network can include direct WiFi or Ethernet connections, intranets or the Internet.

The SDK includes various examples to demonstrate the functionalities of the robot. The interface between the robot and ROS1, provided by Boston Dynamics, is not directly accessible from the Docker container. As discussed in Subsection 3.5, using ROS2 for this project requires working on Ubuntu 20.04, and since Spot CORE is incompatible with ROS2, Docker is needed to bridge this gap. The interface for having the robot data available on ROS, is available only for ROS1. Therefore, it is necessary to retrieve the required information directly from the SDK using the appropriate commands. This involves re-mapping all essential information to ensure they are available for ROS2, usually they are mapped in ROS2 topics, allowing prevention towards data loss.

3.7 Sockets

To establish communication between the custom payload of the ZED2 camera and the Docker container hosting the project on the Spot Core, a communication mechanism is required. The camera has its own processor to handle the data acquisition, but this data must be transmitted to the robot's computing unit, specifically, to the Docker container responsible for managing and processing this information.

Sockets enable communication between different machines or processes over a network by allowing Python programs to send and receive data through Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) connections Al-Dhief et al. (2018). In this model, one machine or process acts as a server, and the other acts as a client. Python provides a socket module Python (2024), which serves as a low-level interface for network communication.

The UDP connection does not establish or maintain a persistent connection between the client and server. Instead, it sends individual packets, known as datagrams, from one host to another with minimal overhead. Each datagram is treated as an independent unit of data. UDP does not guarantee delivery, meaning datagrams may not reach their destination, might arrive out of order, and are not reordered or checked for errors. There are no built-in mechanisms in UDP to ensure data integrity or correct any issues that may arise during transmission.

The TCP connection instead is designed to provide reliable, ordered, and error-checked delivery of data. It establishes a connection between the client and server through a process known as the three-way handshake, ensuring that both parties are ready to exchange data. TCP guarantees that data sent from one end is received exactly as it was transmitted, in the correct order, and without errors. If any data is lost during transmission, TCP handles its re-transmission. Additionally, TCP ensures that packets are reassembled in the correct sequence at the receiving end.

Due to the characteristics of the TCP connection and the need for reliable communication between the camera payload and the Spot core, TCP has been selected as the connection protocol for the project. This choice ensures that all information is sent in the correct order and that no data is lost, which is crucial for search and rescue operations during a disaster.

3.8 Navigation2

The navigation stack designed for ROS2 is called Nav2. It enables mobile robots to navigate through complex environments. Nav2 supports various functionalities, including moving

from one point to another, adding intermediate poses, and performing tasks such as object following, localization, mapping, path planning, path following and obstacle avoidance. It provides additional capabilities for developing highly reliable autonomous systems. Nav2 creates an environmental model from sensor and semantic data, dynamically plans paths, calculates motor velocities, avoids obstacles and orchestrates higher-level robot behaviors.

Nav2 uses Behavior Trees (BTs) (Colledanchise and Ögren (2017)) to enable intelligent navigation through various independent modular servers. Each server handles different navigation tasks, and communication between the servers is facilitated via BT over a ROS2 interface, such as services or actions. Some of the servers are the following ones:

- SLAM: Integrates with SLAM algorithms to build a map while simultaneously localizing the robot.
- Adaptive Monte Carlo Localization (AMCL): Used for localization by employing particle filters to estimate the robot's position on a known map.
- Global Planner: Calculates an optimal path from the start to the goal position, typically using algorithms like A* or Dijkstra's.
- Local Planner: Manages real-time obstacle avoidance and path execution, ensuring the robot follows the global plan while reacting to dynamic obstacles.
- Controller Server: Executes velocity commands to follow the planned path, interacting with the robot's hardware.
- Recovery Behaviors: Provides mechanisms for the robot to recover from situations where it gets stuck or cannot find a valid path.

The architecture of Nav2 is illustrated in Figure 3.9. The Controller, Planner, Behavior and Smoother servers host a variety of algorithm plugins to accomplish different tasks. At runtime, the Planner, Smoother and Controller servers are configured with specific algorithm names and types. Each of these servers exposes an action interface corresponding to its designated task. When the Behavior Tree (BT) triggers a corresponding node, it invokes the action server to perform the task, which then calls the chosen algorithm by its name. This name is mapped to a specific algorithm plugin configured in the server.

In the Behavior Server, each behavior has a unique name, and each plugin also exposes its own dedicated action server. This design accommodates the diverse range of behavior actions. Additionally, the Behavior Server subscribes to the local costmap, receiving real-time updates from the Controller Server to perform its tasks. This approach minimizes the

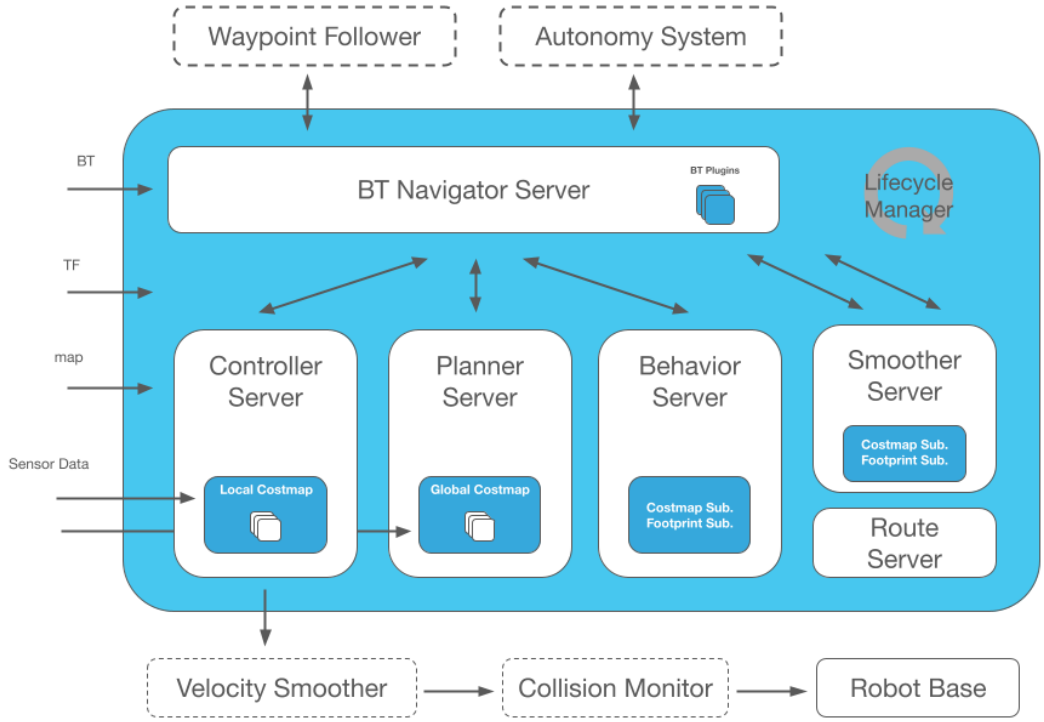


Figure 3.9 Nav2 architechture.

need for multiple instances of the local costmap, which would otherwise be computationally expensive.

In ROS2, planners and controllers serve distinct roles. Planners are responsible for computing a path or route to achieve a specific objective function. For example, a planner might compute a route to a goal or plan to cover all free space. Planners have access to a global environmental representation and buffered sensor data. In contrast, controllers focus on computing feasible control efforts by utilizing a local environment representation. Their goal is to follow the globally computed path by generating a locally feasible path at each update iteration. This ensures that the robot can adapt to changes in the local environment while adhering to the global plan.

3.9 RTAB-Map

Real-Time Appearance-Based Mapping (RTAB-Map) is a graph-based SLAM approach designed for RGB-D, stereo, and LiDAR sensors, utilizing loop closure detection. It employs a bag-of-words technique to assess the likelihood that a new image is from a previously visited location or a new one. Upon confirming a loop closure hypothesis, a new constraint is

added to the graph, and a graph optimizer minimizes map errors. RTAB-Map can be used with stereo cameras, 3D LiDAR or laser rangefinders. For this thesis, RTAB-Map has been utilized with a 3D LiDAR to create a map from point cloud data. Its seamless integration with ROS2 facilitates incorporation into ROS2-based robotic systems, providing various ROS2 nodes and topics for sensor interaction, data processing, and result visualization.

RTAB-Map offers a wide range of parameters to tailor the map-building process to specific user goals. These parameters include settings for detection range and height, optimizer strategies, and occupancy grid configurations. For a complete list of parameters and their descriptions, you can visit the RTAB-Map Parameters Documentation.

3.10 PlanSys2

The thesis focuses on developing a planner to compute an efficient sequence of actions for search and rescue environments using Ubuntu 20.04 LTS and ROS2. For this purpose, ROS2 provides PlanSys2 Martin et al. (2021), a task planning framework designed to offer high-level planning and decision-making tools for robotics. PlanSys2 is a PDDL-based planning system that leverages ROS2's advanced features, providing developers with a structured, reliable, and efficient solution. It incorporates the latest concepts from ROS2, which is currently the de facto standard in robotics.

As discussed in the Section 3.4, ROS2 introduces the concept of LifeCycle Nodes, which have a well-defined lifecycle consisting of states and transitions from creation to destruction. These states are observable, and transitions can be triggered both internally and externally. Each state transition has specific responsibilities, including memory allocation, configuration, communication management, and error handling. In the context of PlanSys2, all components and actions that the robot may perform are implemented as LifeCycle Nodes.

The Plansys2 architecture (Figure 3.10) features a modular design where each component clearly defines its interfaces. The Planner node serves as the central component, responsible for invoking the plan solver that contains the planning algorithm. The default plan solver is POPF, which has been selected for this project, although Temporal Fast Downward (TFD) is also available. Additionally, other plan solvers can be integrated and utilized for plan generation. Whenever the Planner node needs to generate a plan, it calls the Domain Expert and Problem Expert nodes to provide the necessary domain and problem information, respectively.

The Domain Expert node reads the PDDL domain file created by the user and stores it in memory. It supports multiple domains that can be combined to form a single domain,

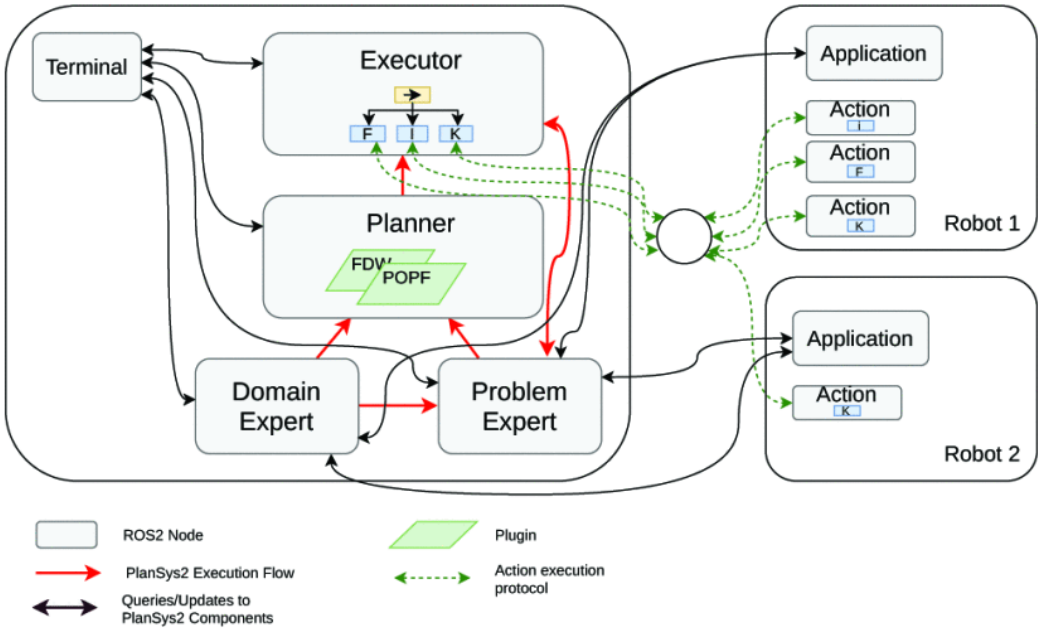


Figure 3.10 PlanSys2 architecture.

which enhances modularity. This node provides the domain to the Planner along with various request information about it.

The Problem Expert node contains the problem definition that needs to be solved by the plan. It includes instances, predicates, functions and goals, all of which must be written in accordance with the Domain Expert node's specifications. This node assembles the PDDL problem when requested by the Planner node.

Finally, the Executor node is responsible for implementing the plans created by the Planner node. It receives requests to execute the plan based on the current state of the Problem Expert node. The Executor queries the Planner node for a plan, and if a valid plan is available, it proceeds to execute it. The plan is converted into a Behavior Tree (BT), which manages the execution by verifying requirements and applying effects.

Each individual action defined in the domain file is implemented as a ROS2 node. Additionally, the application incorporates a controller to interact with the Problem Expert node, facilitating the execution or cancellation of plans through the Executor node.

Chapter 4

Actions and controller

Summary

This chapter introduces the software architecture used in the project development, covering the operating system, the communication mechanisms between the robot and the operating system, and the tools employed to achieve the desired behavior through a planner. It also details the actions developed for the PDDL domain file, including their preconditions, effects, and impact on the robot's performance. It provides a comprehensive explanation of how each action operates and outlines the development efforts undertaken to achieve the desired behavior for each action. Additionally, the chapter presents the controller responsible for managing the creation and cancellation of plans based on the robot's current perception of the environment.

4.1 Software architecture

This section provides an overview of the project architecture, explaining how the different components communicate and interact to achieve the desired behavior of the Spot robot.

As shown in Figure 4.1, various nodes have been developed. Nodes in purple indicate those that were fully developed for this project, while blue nodes represent modules already provided by ROS2, such as PlanSys2, Nav2, RTAB-Map and the Spot SDK that have been used for the project. The pink node represents the controller node, which, although also fully developed as part of this work, manages the information from the various actions and modifies the PDDL plan problem based on the current data. The green area refers to the camera processor, located outside the Spot CORE on the NVIDIA processor of the ZED2

camera, where a node has been implemented to establish a TCP socket connection with the nodes integrated within the Spot CORE. Except for this node, which cannot be allocated within the Spot CORE, all other nodes are located in a Docker container that hosts Ubuntu 20.04 with ROS2 integrated, providing fast portability to other Spot COREs if needed. Some of the most important topics are also reflected in the architecture, along with the information that is sent or received from the Spot SDK and other modules to create the map or achieve navigation. A more detailed explanation of each action will be provided later in this chapter, where the purple action nodes are further elaborated.

Straight black lines represent the acquisition of information through the SDK, which is obtained by querying the SDK for the necessary data. For the check action, the battery status is requested to retrieve this information. In the case of the move action, odometry and point cloud data are requested to publish the robot's position on a custom `/odom` topic and to facilitate environmental mapping, respectively; this information is also sent to RTAB-Map.

For other actions, the SDK is used to modify the Spot robot's behavior. In both evaluate and search actions, the robot's orientation is adjusted based on the specific goals of each action. The black arrows connecting PlanSys2 with each action represent the communication through PlanSys2, activating each action when deemed necessary and upon completion of the previous action. In contrast, the arrows from each action back to the PlanSys2 block indicate the feedback being returned during execution and the final result obtained upon completion of the action.

PlanSys2 sends the results of the actions to the controller, which, based on the acquired knowledge, may adjust the problem knowledge, modify the goal, or leave it unchanged. Additionally, the striped arrows refer to publisher/subscriber nodes, specifying the topic associated with each arrow. For the move action, there is a publication on the `/goalpose` topic, which contains the goal point. The local planner, working alongside Nav2, subscribes to this topic and publishes updates to a new topic, `/goaltospot`, indicating subpoints to reach in order to arrive at the final position.

The camera information server publishes two topics: `/evaluation` and `/position`, providing the Evaluate and Search nodes with the necessary camera data to perform their respective actions. Another method for sending data occurs through a TCP socket connection from the camera processor node to the camera information server. Lastly, all nodes are encapsulated within the ROS2 Docker container, except for the camera processor node, which is not hosted on the Spot CORE.

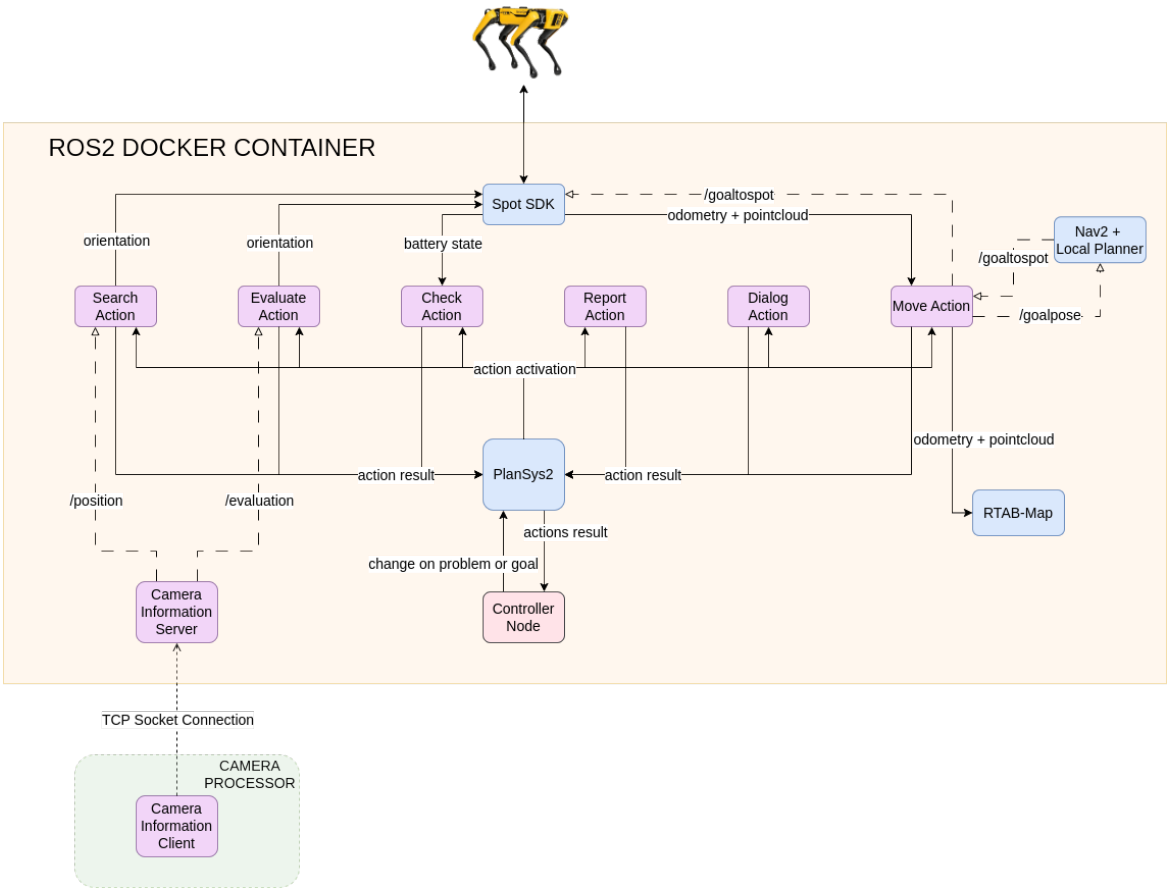


Figure 4.1 Project architecture.

4.2 Domain

The goal of the project is to develop actions that enable Spot to navigate through an environment after a disaster in search of people. Once a person is found, the robot will prioritize evaluating their condition and reporting their status. If the person can move, Spot will guide the person to the exit, and then it will continue with the plan. Additionally, before proceeding to a new location, Spot must check if it has sufficient battery to continue its exploration. If the battery is low, the robot should halt its exploration and return to the charging station.

4.2.1 Types

In PDDL, types are used to define categories or classes of objects within a planning domain. They help to organize and specify the objects that can be used in planning tasks by distinguishing between different kinds of entities. In addition, types can also be defined as constants, which allows for the definition of objects that are present in the problem. Types make the domain definition more structured, allowing for more precise modelling of actions and their applicability. It is also possible to establish a hierarchy between types. Figure 4.2 shows the hierarchy of the types for this problem, which are the following:

- **robot**: This type refers to the robot, specifically for this project Spot, which is the entity responsible for exploring the environment and searching for people.
- **person**: This type represents individuals that the robot may encounter during exploration. All detected individuals will be classified under this type.
- **location**: This type denotes various locations that the robot can navigate to or from.
- **state**: This is a constant type representing the different poses a person can be found in. It can have one of four values: stand, sit, lay, or unknown if the evaluation is not reliable enough to determine the person's position.
- **consciousness**: This constant type indicates the different states of consciousness a person might exhibit. Possible values are conscious, unconscious, or confused.

4.2.2 Predicates

Predicates in PDDL describe the properties of objects and their relationships within a planning domain. They help to establish conditions that can be either true or false, reflecting the state

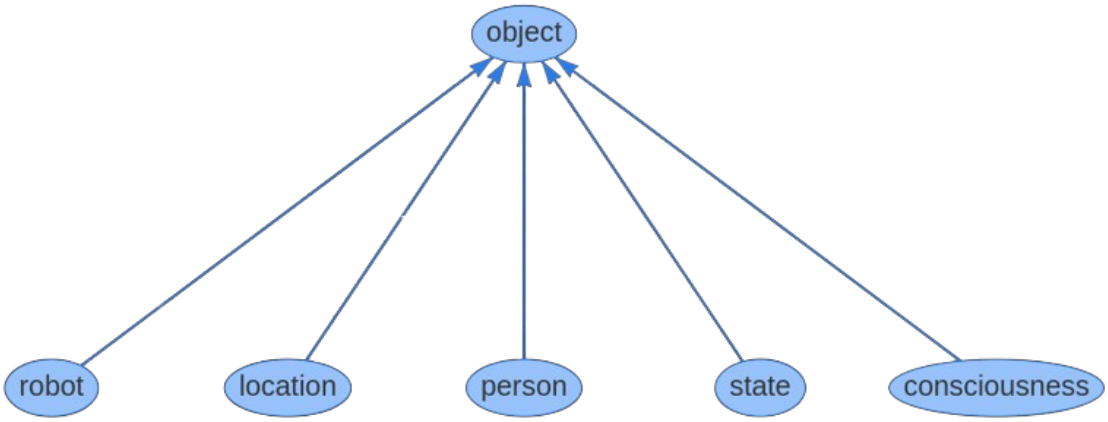


Figure 4.2 Types hierarchy.

of the world at any given time. By default, predicates not explicitly stated are assumed to be false due to the closed world assumption. The predicates declared for this problem domain are:

- `robot_at`: This predicate is used to specify the current location of the robot. It takes two parameters: the robot type and the location type. It is defined as `(robot_at ?r - robot ?l - location)`, where `?r` represents the robot and `?l` represents the location where the robot is currently situated.

- `connected`: This predicate indicates that two locations are physically connected, meaning the robot can move from one location to the other. It takes two parameters, both of which are of the location type. It is defined as `(connected ?lo1 ?lo2 - location)`, where `?lo1` and `?lo2` represent the two connected locations.

- `person_detected`: This predicate is used to track the locations where people have been found. It takes two parameters: one of type person to represent the detected individual and one of type location to indicate where the person was detected. It is defined as `(person_detected ?p - person ?l - location)`.

- `searched`: This predicate indicates that a specific location has been explored by the robot. It takes two parameters: one of type robot to represent the robot performing the search and one of type location to denote the location that has been searched. It is defined as `(searched ?r - robot ?l - location)`.

- `person_evaluated`: This predicate becomes true when a person is evaluated, it is a requirement to satisfy for each person found. It is defined as `(person_evaluated ?p - person)`

- `person_reported`:This predicate becomes true once a person's status has been reported. It marks the final stage of the assessment, taking place after both the evaluation and dialogue have been completed. It is essential for tracking whether each detected person has undergone evaluation. It is defined as `(person_reported ?p - person)`.

- `battery_unchecked`: This predicate indicates that the robot's battery level has not yet been checked. Since negative preconditions are not available in the planner, this predicate helps manage battery checking requirements. It is defined as `(battery_unchecked ?r - robot)`.

- `battery_checked`: This predicate indicates that the robot's battery level has been checked. It is defined as `(battery_checked ?r - robot)`. This predicate is used in conjunction with the `battery_unchecked` predicate, as negative preconditions are not supported. Depending on the action, either the positive or negative predicate will be utilized to satisfy the conditions.

- `is_free`: This predicate indicates whether the robot is free and not currently engaged in any other actions. It is defined as `(is_free ?r - robot)`. This helps to ensure that the robot does not perform multiple actions simultaneously, maintaining a single-task focus.

- `next_move`: This predicate specifies the next location where the robot will move. It is used to determine the destination during the check action. It has two parameters: one for the robot and one for the location, defined as `(next_move ?r - robot ?l - location)`. This helps in managing the robot's movement plans and ensuring it moves to the correct location.

- `person_state`: This predicate tracks the evaluation status of a person during the search. It records the state of a person once they are evaluated. It has two parameters: one for the person and one for the state in which they are found. The predicate is defined as `(person_state ?p - person ?s - state)`. This helps in maintaining and updating the information about each person's condition as the robot performs its search.

- `dialog_finished`: This predicate indicates that the dialogue or interaction with a person is completed. It becomes true once the conversation or evaluation with a person has concluded. It is defined as `(dialog_finished ?p - person)`. This predicate ensures that each person found is fully evaluated and their interaction is properly concluded as part of the action requirements.

- `person_dialog`: This predicate tracks the details of a person's evaluation following a dialogue, including their state of consciousness. It is added to the problem when a person is evaluated, recording both the person's identification and their state of consciousness. It is defined as `(person_dialog ?p - person ?c - consciousness)`. This predicate helps maintain the context of the person's state after the evaluation process.

4.3 Actions

In this section all the actions developed for the thesis plan are presented and explained. Each action is divided into three sections. The first one is the parameters section. It details the arguments that the action accepts, including the number and types of each argument. This information allows for precise modelling of the action. The second section is the condition, which is the set of predicates that must be true for the action to be executed. These conditions ensure that the action can only be performed when all prerequisites are satisfied. Finally, the effects section, which describes the predicates that will become true as a result of executing the action. These effects reflect the changes made by the action in the planning domain.

4.3.1 Move to

One of the main actions that need to be implemented is an action that enables the robot to move between different locations in the environment to explore and search for people. The parameters are three: one for the robot type and two for location types, indicating that the robot will move from an initial location to a final location. The conditions that must be true to execute the action are the following: both locations must be connected, the robot must be located at the initial location, the next move should be to the final location, and the battery must be checked to ensure it has enough charge to complete the motion. The effects of this action will be that the robot is no longer at the initial location but is at the final location, the next move will no longer be the destination since the robot has already moved there, and the battery will not be checked anymore. The complete action can be found in Listing 4.1.

```
(:durative-action move
    :parameters (?r - robot ?r1 ?r2 - location)
    :duration ( = ?duration 5)
    :condition (and
        (at start(battery_checked ?r))
        (at start(connected ?r1 ?r2))
        (at start(robot_at ?r ?r1))
        (at start(next_move ?r ?r2))
        )
    :effect (and
        (at start(not(robot_at ?r ?r1)))
        (at end(robot_at ?r ?r2))
        (at end(not(next_move ?r ?r2)))
        (at end(battery_unchecked ?r))
        (at end(not(battery_checked ?r)))
    )
)
```

Listing 4.1 Move action.

4.3.2 Search

This action enables the robot to search for people in the specified location by rotating and recording the individuals it finds and their positions. The action has two parameters: one for the robot and one for the location. The conditions required to execute the action are that the robot must be available, meaning it should not be performing any other actions, and it must be located in the specified location. As a result of the action, the robot will not be available at the beginning but will become available again once the action is completed. The predicate indicating that the location has been searched will become true, signifying that the robot has completed the exploration. Additionally, the battery will be marked as unchecked, both predicates for this are changed accordingly. The complete action code can be found in Listing 4.2.

4.3.3 Evaluate

The evaluate action focuses on a person rotating so that the robot is facing towards the detected person. After focusing, the robot will assess the condition of the person. This action has three parameters: the location, the person, and the robot. The conditions required to execute the action are: the robot must be available at the start, a person must have been detected in that location, and the robot must be present at the same location. As a result, the

```
(:durative-action search
    :parameters (?r - robot ?l - location)
    :duration ( = ?duration 5)
    :condition (and
        (at start(is_free ?r))
        (over all(robot_at ?r ?l))
    )
    :effect (and
        (at start(not(is_free ?r)))
        (at end(is_free ?r))
        (at end(searched ?r ?l))
        (at end(battery_unchecked ?r))
        (at end(not(battery_checked ?r)))
    )
)
```

Listing 4.2 Search action.

robot will be unavailable until the action is completed, and the predicate `person_evaluated` will become true. The full action code can be found in Listing 4.3.

```
(:durative-action evaluate
    :parameters (?l - location ?p - person ?r - robot)
    :duration ( = ?duration 5)
    :condition (and
        (at start(is_free ?r))
        (at start(person_detected ?p ?l))
        (over all(robot_at ?r ?l))
    )
    :effect (and
        (at start(not(is_free ?r)))
        (at end(is_free ?r))
        (at end(person_evaluated ?p))
    )
)
```

Listing 4.3 Evaluate action.

4.3.4 Dialog

The dialog action initiates a conversation between the person and Spot to assess the person's level of consciousness. The parameters required for this action are the robot, the location, and the person. The conditions that must be true include the robot being available, the person having already been evaluated, and the robot being located in the specified location. As a result of this action, the robot will be unavailable until the dialog is complete, and a new predicate indicating that the dialog has finished will become true. The full action code can be found in Listing 4.4.

```
(:durative-action dialog
    :parameters (?l - location ?p - person ?r - robot)
    :duration ( = ?duration 5)
    :condition (and
        (at start(is_free ?r))
        (at start(person_evaluated ?p))
        (over all(robot_at ?r ?l))
    )
    :effect (and
        (at start(not(is_free ?r)))
        (at end(is_free ?r))
        (at end(dialog_finished ?p))
    )
)
```

Listing 4.4 Dialog action.

4.3.5 Report

The report action involves compiling all the collected information about each person and writing it to a text file, ensuring that the information is continuously updated. The parameters required for this action are the location, the person, and the robot. The conditions that must be met include the robot being free to perform the action, the dialog with the person being completed, and the robot being in the specified location. As a result of this action, the robot will remain occupied until the action is finished, and the predicate `person_reported` will become true for that person. The full action code can be found in Listing 4.5.

```
(:durative-action report
    :parameters (?l - location ?p - person ?r - robot)
    :duration ( = ?duration 5)
    :condition (and
        (at start(is_free ?r))
        (at start(dialog_finished ?p))
        (over all(robot_at ?r ?l))
    )
    :effect(and
        (at start(not(is_free ?r)))
        (at end(is_free ?r))
        (at end(person_reported ?p))
    )
)
```

Listing 4.5 Report action.

4.3.6 Check

The check action is responsible for monitoring the robot's battery level to determine whether the exploration should be halted if the battery is insufficient. The parameters required for this action are the robot, the current location, and the next location. The conditions that must be satisfied include the robot being free, the robot being located at the current location, both locations being connected, and the battery being unchecked. This is why two predicates are necessary: it cannot simply be written as `(not(battery_checked ?r))` because negative preconditions are not accepted, so the negative predicate `(battery_unchecked ?r)` is used to represent the same condition and is necessary for the action. As effects, the robot will remain occupied until the action is completed, the `battery_checked` predicate will become true, the `battery_unchecked` predicate will become false, and the predicate for the next move will become true with the destination location as its argument. The full action code can be found in Listing 4.6.

```
(:durative-action check
    :parameters (?r - robot ?from ?to - location)
    :duration ( = ?duration 5)
    :condition (and
        (at start(battery_unchecked ?r))
        (at start(is_free ?r))
        (at start(robot_at ?r ?from))
        (at start(connected ?from ?to))
    )
    :effect (and
        (at start(not(is_free ?r)))
        (at end(is_free ?r))
        (at end(not(battery_unchecked ?r)))
        (at end(battery_checked ?r))
        (at end(next_move ?r ?to))
    )
)
```

Listing 4.6 Check action.

4.4 Controller

For the development of the project, PDDL actions are required to be used with PlanSys2 to create a coherent sequence of actions based on the current state of the environment. It's important to recognize that the initially created plan may become obsolete at any point, necessitating replanning to develop a plan better suited to the current situation. Due to this requirement, it is necessary not only to have actions that execute different parts of the plan but also a controller that manages the plan. This controller can request the plan from the Planner node of PlanSys2, cancel the plan if needed, replan to obtain a more updated plan, or continue executing the appropriate action at the right time. The controller has been developed in C++ and manages the outcome of each action, responding accordingly.

At the beginning, the controller is responsible for initializing the knowledge required to solve the problem. It adds the instance Spot to the robot type and various locations to the location type. Additionally, it includes different instances representing the positions in which people can be found (standing, lying down, sitting, or unknown) in the state type, as well as different levels of consciousness (conscious, unconscious, confused). It includes the necessary instances for battery checking. After these instances have been added to the Problem Expert node, the controller adds predicates according to the initial scenario. This involves connecting physically connected locations, allowing Spot to move between them,

setting a predicate that places the robot at its initial location and define that the robot is free and the battery is not checked. Finally, the goal is defined: the initial goal is to search all locations for people. Once all the locations have been explored, the robot's mission is complete.

In the main part of the code, after the predicates and instances have been initialized, the plan execution begins. Depending on the action feedback, the controller performs different tasks. These are the various actions the controller takes based on the feedback received when it calls the `getFeedback()` method from the Executor client node:

- Move to: For the action that moves the robot from one point to another, the controller keeps track of the robot's current position in a variable whenever the robot changes its location.

- Search: For the action that searches for people in a location, the controller keeps track of all the people found in that room based on the feedback provided by the action. If no one is found, it continues with the plan. However, if one or more people are found, the controller saves information about how many people were found and their positions. It then performs a replan, which involves adding an instance of type "person" for each person found and creating a predicate that links each person to their location to maintain the information. Additionally, the replan updates the goal to include predicates for evaluating, reporting, and having a dialog with each person found. Thus, the new goal is not only to search all locations but also to evaluate each person, establish a dialog with them, and finally report their state.

- Evaluate: For the action that evaluates the condition of the person found in the location, once the action has finished and the person's condition has been assessed, the controller adds a new predicate to the problem to indicate that the person has been evaluated along with their state. It also updates the internal information of the controller with the person's condition and publishes the newly obtained information to the topic `/reported_info`.

- Dialog: For the action that conducts a dialog with the person, once the action is completed and the person's state (conscious, unconscious, or confused) is received, a new predicate is added to the PDDL problem to indicate the identification of the person along with their state. The controller also updates its internal information with the person's condition and publishes the newly obtained information to the topic `/reported_info`.

- Report: Once the report action has finished, if the assessment has not been completed for all the people found in the location, the controller performs a replan. This replan adds to the goal that the next person must be evaluated, reported, and have a dialog conducted. If all people have been reported and at least one person is conscious, the controller performs a different replan. This replan guides the conscious individuals to the exit to receive assistance as soon as possible. It is important to note that this is a simplification; in a more general case, it will be necessary to verify whether the person is in a condition to move.

- Check: For the action that checks the battery to determine if the robot can continue with the exploration or needs to return to charge, the controller depends on the feedback provided by the action. If the battery level is sufficient, it continues with the plan. Otherwise, the controller performs a replan by removing the current goal and updating it to only include going to the battery charging location. This involves canceling the ongoing plan and starting a new plan to adapt the actions to the current situation.

Therefore, the goal of the controller is to manage the feedback from the various actions and respond accordingly. It replans when necessary to update the plan with the newly obtained information or updates the problem instances based on the performance of the actions.

For example, the code below represents the initial plan created for an environment with three locations, labeled a, b and c, and one additional location called battery_location where the battery charger is located.

```
Action: (search spot a)
Action: (check spot a b)
Action: (move spot a b)
Action: (search spot b)
Action: (check spot b c)
Action: (move spot b c)
Action: (search spot c)
```

The goal of the plan is:

```
(:goal (and (searched spot a)
            (searched spot b)
            (searched spot c))
)
```

This goal means that the exploration must be completed in all locations. If, during the search action in location b, the robot finds a person, the controller will perform a replan. The new goal to be satisfied will be:

```
(:goal (and (searched spot c)
            (searched spot b)
            (searched spot a)
            (person_evaluated p0)
            (dialog_finished p0)
            (person_reported p0))
)
```

The new plan generated after the replan would be:

```
Action: (evaluate b p0 spot)
Action: (dialog b p0 spot)
Action: (report b p0 spot)
Action: (check spot b c)
Action: (move spot b c)
Action: (search spot c)
```

If, before moving from b to c, the check action indicates that the battery is too low, the goal would be updated to:

```
(:goal (and (robot_at spot battery_location)))
```

The plan of actions after the replan would be:

```
Action: (move spot b battery_location)
```

4.5 SLAM and Navigation for move action

The move action in the PDDL plan is implemented in ROS2 using Python under the filename move_action_node.py. As described in the previous section, the various points to which the robot can move are referred to as locations. In the move action code, these locations are initialized with their respective coordinates and stored as PoseStamped() messages. They are then saved in a dictionary where the key represents the name of the location as defined in the PDDL problem, and the value is the PoseStamped() coordinates corresponding to that location.

PlanSys2 utilizes the action executor client, which provides a topic named `/actions_hub`. This topic allows the action code to retrieve information about the predicates, parameters, and the action being executed. In this way, when the move action is active, the parameters are accessed through this topic, enabling the retrieval of details such as the start and destination locations for the robot's movement. Using this information, along with the dictionary initialized at the start that contains the locations' coordinates, the system can determine the

precise coordinates of the destination. This coordinate information is then published on a topic called `/goal_pose`, which conveys the location coordinates where the robot needs to move.

The `/goal_pose` topic is connected with the navigation stack, as detailed in Section 3.8. This topic is used to send goal positions and orientations to the navigation system, directing the robot to navigate to a specific location within its environment. Although this information can be published manually from an interface such as RViz, in the context of the system's autonomous behavior, it is handled through the move action by publishing to this topic. The data published on `/goal_pose` is processed by the planner and controller nodes within Nav2, which are responsible for generating and executing a path to the goal while avoiding obstacles.

To achieve the desired navigation behavior for the Spot robot, a custom local planner has been developed in C++. This planner works alongside the global planner's path to determine the next waypoint, continuously guiding the robot toward its final goal.

The custom local planner is necessary due to the need for integration between the navigation stack's controller and the Spot robot's SDK. The controller computes the next goal point based on the global path, factoring in obstacles and safety distances. The local planner then publishes this calculated point on a topic called `/goaltospot`, enabling communication with the SDK. This process ensures that the robot navigates safely and efficiently toward its next target while adapting to its environment.

For the robot to move as intended, additional integration with the Spot SDK is necessary beyond just computing waypoints. To facilitate this, a separate file named gotopoint.py has been created using ROS2 and Python. This script establishes a connection with the Spot robot by utilizing the SDK's commands for authentication, setting mobility parameters, and configuring the client.

In gotopoint.py, a subscription is set up for the topic `/goaltospot`, where the messages are of a custom type called Trajectory, which includes data of the PoseStamped type. Upon receiving a message on this topic, the script converts the PoseStamped data into a RobotCommandBuilder command —a specific command format provided by the SDK for controlling the robot's movement. The script then sends this command to the Spot robot, effectively directing its movement based on the computed waypoints and ensuring that it navigates as planned.

Simultaneously, it is essential to define the SLAM component to enable the robot to build a map while it is moving and to localize itself while being aware of surrounding obstacles. Predefining the map is not feasible due to the dynamic nature of the environment,

especially after a catastrophe, where the environment can be cluttered with unknown obstacles. Therefore, creating and updating the map in real-time as the robot navigates is the most effective approach.

To achieve this, RTAB-Map, as detailed in Section 3.9, is used in conjunction with the navigation stack. RTAB-Map utilizes the point cloud data obtained from the LiDAR sensor attached to the Spot robot. For RTAB-Map to generate a coherent and continually updated map, it requires a constant input of point cloud data from the LiDAR. This data allows RTAB-Map to process and integrate the information into the map, ensuring it remains accurate and reflective of the current environment. Consequently, the robot can navigate effectively, localize itself, and avoid obstacles based on the most recent map updates. Different steps must be taken to receive point cloud information for performing SLAM.

The first step is to obtain the point cloud data from the LiDAR. Since Docker with ROS2 is being used, an initial connection between the SDK and Docker is required to access this data. This connection allows for the publication of point cloud information under a new topic so it can be utilized within Docker. For this purpose, a file named vel2.py has been created. Written in ROS2 with Python, this file establishes a connection with the Spot SDK, following the authentication steps similar to those in the gotopoint.py file. After retrieving the point cloud data using the appropriate SDK commands, vel2.py constructs a PointCloud2 message and compiles the data into a cloud format. This point cloud data is then associated with the `odom` frame, which serves as the fixed world reference frame for building the map and for robot movement. The processed point cloud information is published under the topic `/points2`, making it available for use within Docker.

Since basic information about the robot, such as its position and orientation, is not directly accessible due to the use of Docker and ROS2, the vel2.py file also handles publishing this data. Specifically, it creates and publishes an Odometry message under a new topic called `/odom`. This message contains the position and orientation of the Spot robot as read from the SDK. The Odometry message has its header frame set to `odom`, which serves as the fixed world reference frame. The child frame is set to `bodyy`, which is linked to the Spot robot and updates as the robot moves. This setup ensures that the robot's position and orientation are accurately represented and available for other components within the ROS2 environment.

The point cloud data is published to the topic with the header set to the fixed frame `odom`, which serves as the reference frame. However, as the robot moves, it is crucial to ensure that the new points in the point cloud are accurately positioned and not affected by the robot's movement offset. To address this issue, a second script named 2_pointcloud.cpp has been developed in ROS2 using C++.

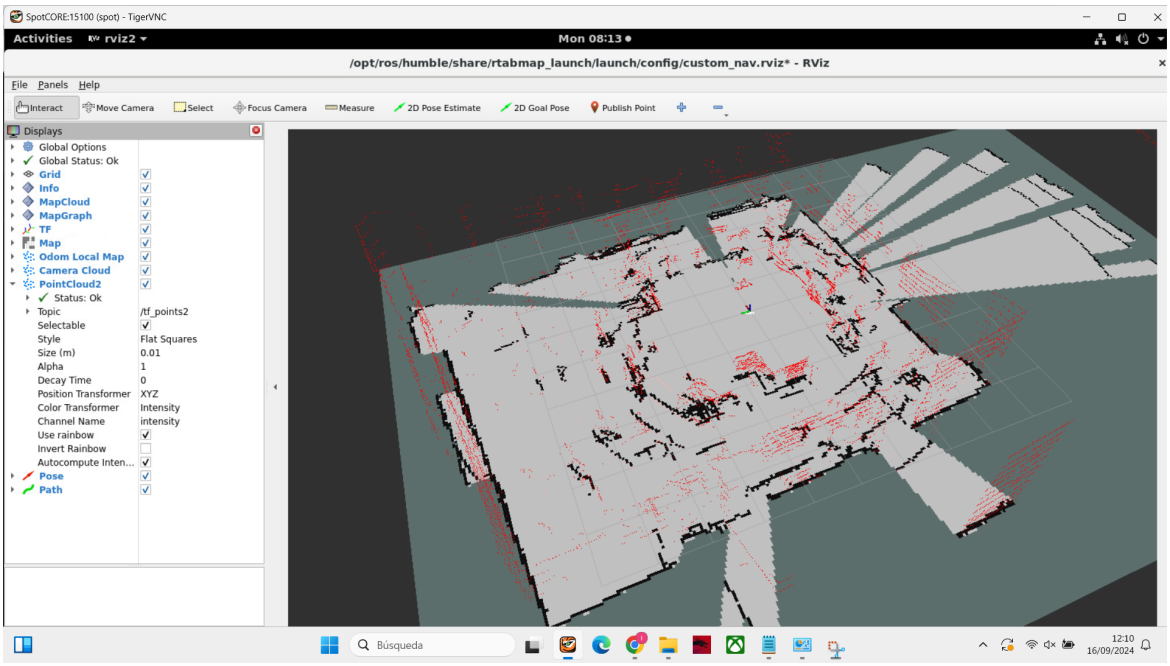


Figure 4.3 Built map and point cloud obtained through the topic /tf_points2 on RViz.

This script subscribes to the `/points2` topic, where the raw point cloud data is published. It then performs a transformation of the point cloud data from the original reference frame `odom` to the frame `bodyy`, which is directly linked to the robot. The transformed point cloud data is published under a new topic called `/tf_points2`. This ensures that the point cloud information accurately reflects the robot's movement and maintains coherence between frames, providing correctly positioned data for further processing.

Finally, RTAB-Map is configured to receive the point cloud data from the topic `/tf_points2` (Figure 4.3). RTAB-Map, along with other parameters, is set up to optimize the map-building process according to the project's requirements. By launching the scripts gotopoint.py, vel2.py, 2_pointcloud.cpp, the navigation stack, and RTAB-Map, the move action is completed. The overall architecture of the move action is illustrated in Figure 4.4.

4.6 Battery information for Check action

During the execution of the mission, it is crucial for the system to monitor the robot's battery state. This is because the robot may need to return to a designated charging point or continue with its search based on the battery level. The check action in the PDDL domain addresses this need.

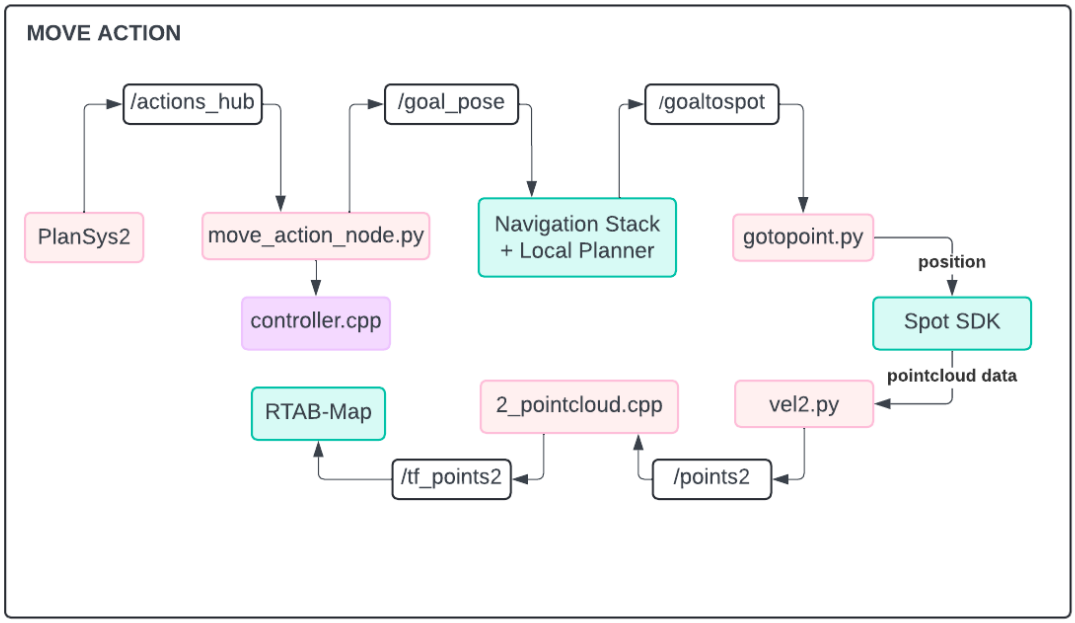


Figure 4.4 Structure of the move action architecture.

The check action is implemented in ROS2 with Python and is named check_action_node.py. This file sets up a ROS2 subscriber that listens to the topic `/get_battery_state`, from which it receives information about the battery percentage. When the check action is initiated, it subscribes to this topic to obtain the current battery status. This information is essential for making real-time decisions about whether the robot should return to a charging station or proceed with its assigned tasks.

For this action, another file named getbattery.py has been created, written in ROS2 with Python. This file is responsible for obtaining battery information from the robot, which involves establishing a connection with the robot's SDK. It follows the same process as other files, including authentication and using SDK commands.

The getbattery.py file creates a publisher for the topic `/get_battery_state`. This topic allows access to the robot's battery percentage through a subscription. The file publishes the battery percentage to this topic, making the information available for real-time decision-making.

Once the battery information is obtained in the file check_action_node.py by subscribing to the `/actions_hub` topic, the action parameters are retrieved. As outlined in Section 4.3.6, these parameters include the robot and the locations involved—both the robot's current position and its destination.

These parameters are crucial because the decision about whether the battery is sufficient or not depends on the distance the robot needs to travel. By calculating the Euclidean distance between the robot's current position and the goal position, and also between the battery location (where the robot should return if the battery is low) and the destination, a comprehensive assessment can be made. The total distance considered is the sum of the distance to the goal and the return distance to the battery location. This total distance is compared against an estimate of battery consumption per unit distance. Based on this comparison, the action determines whether the remaining battery percentage is adequate for the robot to complete its mission or if it needs to return to the battery location. The architecture of the check action is illustrated in Figure 4.5.

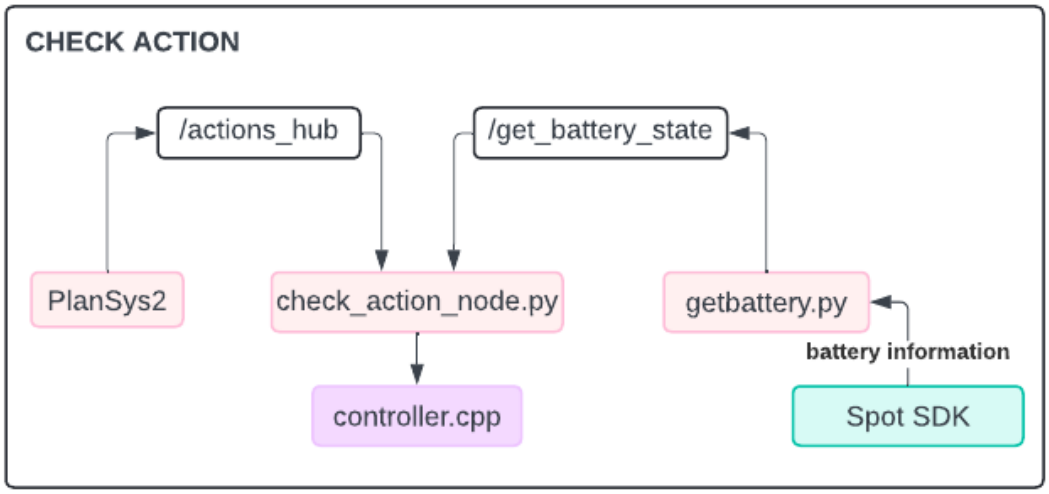


Figure 4.5 Structure of the check action architecture.

4.7 People detection for Search action

The PDDL action for searching for people is implemented in ROS2 with Python and is named search_action_node.py. The purpose of this action is to rotate the robot 360º while looking for people and to store all the information about any individuals detected during the search. For this action, the custom payload of the ZED2 camera is used.

First, the camera data needs to be sent to the Docker container in order to use the information while the camera is recording. As explained in Section 3.7, sockets have been used to establish this connection. On the ZED2 camera processor, a Python file called body_tracking_nogui.py has been created. This file establishes a TCP socket client

connection using IP address 10.0.0.1 and port 65433. Using the StereoLabs library for the ZED2 camera, the script starts the camera and enables body tracking. It is important to notice that only when a person is detected the body tracking of the camera returns information. For this reason, and to avoid initializing the camera twice, the detection of people has been implemented through body tracking rather than proper object detection. For each detected person, a message containing the ID, confidence level, action state, keypoints, keypoint confidence scores, and the positions of each keypoint is sent via the TCP socket connection to ensure the entire message is transmitted correctly.

On the other hand, the TCP socket server connection is established within the Docker container through a file called server_both.py, written in ROS2 with Python. This file receives the data sent by the client running on the camera's processor. The received information is split into two parts. The first part, which includes the position, confidence, and ID of the detected person, is published on a topic called `/position`. This topic publishes a custom message of type Detection that encapsulates this data. The second part, which contains details such as confidence, action state, key points, key point confidences, and the positions of all key points, is published on a separate topic called `/evaluation`.

At this point, the search_action_node.py file can access the camera's information through the available topics. Specifically, it subscribes to the `/position` topic to retrieve the confidence level of the detected person, their ID, and their position. A confidence threshold is set—high enough to ensure that only real people are detected, but not so high as to miss too many valid detections. This balance helps reduce false negatives, as it's preferable to detect a non-person object rather than miss an actual person. When the confidence exceeds the threshold and the detected ID is different from the previously reported one, it considers a new person found. It then reports the person's ID, position, and the location where they were detected to the PDDL controller. To obtain the current search location, the node subscribes to the `/actions_hub` topic, allowing it to access the parameters of the action and determine the location where the robot is performing the search.

To achieve the complete behavior of the search action, the robot needs to rotate. To facilitate this, the search_action_node.py file introduces a new topic called `/rotation`. This topic publishes angle values ranging from 0 to 360 degrees, with increments of 18 degrees. To implement the rotation, the gotopoint.py file used in the move action (as described in Section 4.5) is adapted. This file is modified to include a new subscriber to the `/rotation` topic. When a new angle value is received by the callback function, it is converted from degrees to radians and sent to the Spot SDK, which was initialized earlier in this file. This setup allows the search_action_node.py to effectively communicate with the SDK through

the gotopoint.py file. The architecture of the search action, including the interaction between these components, can be observed in Figure 4.6.

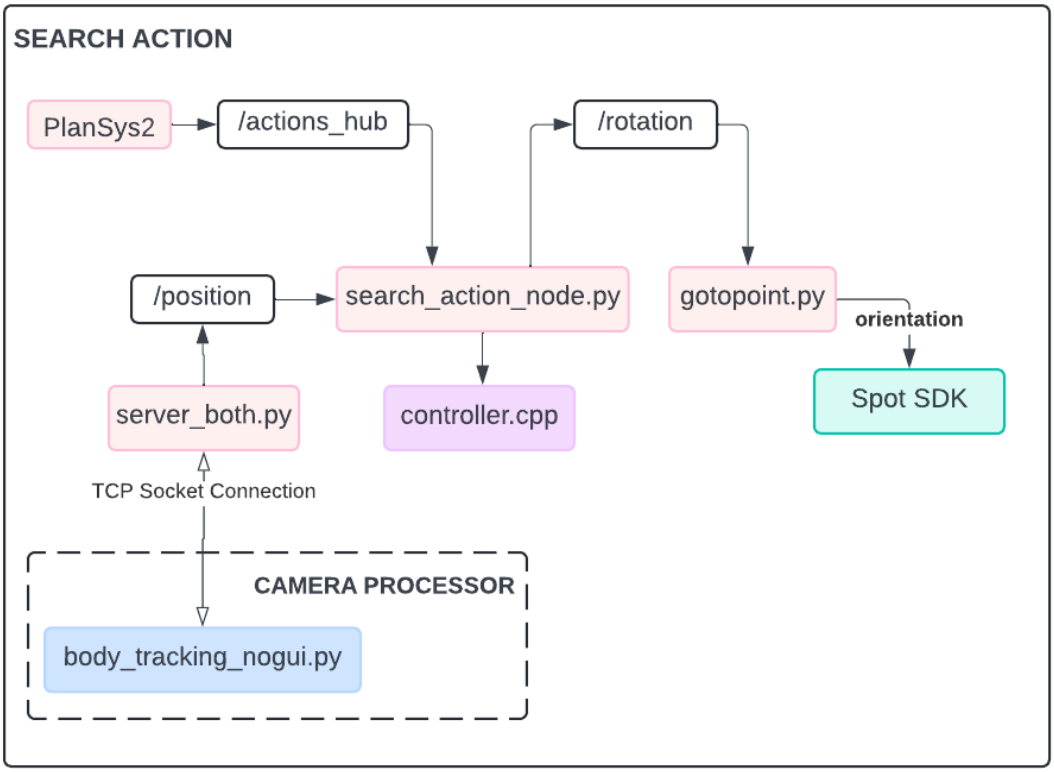


Figure 4.6 Structure of the search action architecture.

4.8 Pose Estimation for Evaluate action

The evaluation action consists of assessing the state of the person found in order to report their condition to rescuers. During this action, the pose of the person is evaluated. The action is implemented in a file called evaluation_action_node.py, written in Python for ROS2. This file subscribes to the `/evaluation` topic, which contains information about the key points of the detected person, as explained in Section 4.7. It also subscribes to the `/actions_hub` topic, provided by PlanSys2, to obtain the parameters of the action—namely, the person, the location, and the robot. Additionally, it subscribes to the `/reported_info` topic, where the controller (Section 4.4) publishes information about the people found.

The first step is to determine if there is a new person to evaluate. The `/actions_hub` topic provides the person ID parameter, which assigns a unique identifier to each person

found. This identifier is constructed with the letter "p" (indicating a person) followed by a number that starts at 0 and increments by one for each new person found.

On the other hand, by subscribing to the `/reported_info` topic, it is possible to track how many people have been reported so far. By comparing the people listed in the `/reported_info` topic (those already reported) with the new person ID, it is possible to determine if the person has already been evaluated (if the person's ID is lower than the total number of reported people) or if they still need to be evaluated.

To determine the pose of the person, it is necessary to orient the robot in that direction, as a full rotation is performed during the search action. The angle where the person was located is stored in the `/reported_info` topic, and the robot rotates to that angle using the `/rotation` topic. As explained in the previous section, the gotopoint.py file, which communicates with the Spot SDK, subscribes to this topic and sends the angle command to the robot. The architecture of the evaluation action is shown in Figure 4.7.

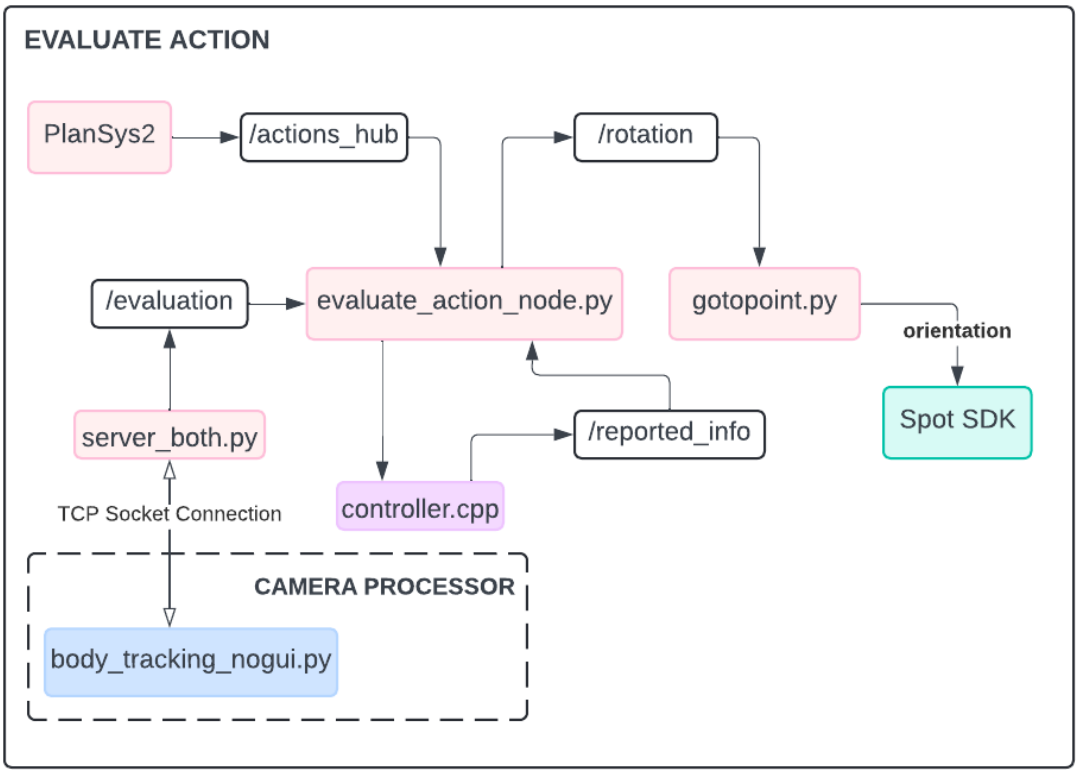


Figure 4.7 Structure of the evaluate action architecture.

Once the robot centers on the person, it uses key point information to estimate the person's pose. The ZED2 camera SDK provides a set of 18 key points (Figure 4.8) that follow a

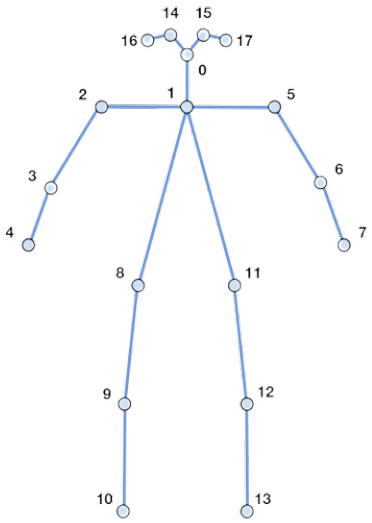


Figure 4.8 18 keypoints detected by ZED2 camera.

KEYPOINT INDEX	KEYPOINT NAME	KEYPOINT INDEX	KEYPOINT NAME
0	NOSE	9	RIGHT_KNEE
1	NECK	10	RIGHT_ANKLE
2	RIGHT_SHOULDER	11	LEFT_HIP
3	RIGHT_ELBOW	12	LEFT_KNEE
4	RIGHT_WRIST	13	LEFT_ANKLE
5	LEFT_SHOULDER	14	RIGHT_EYE
6	LEFT_ELBOW	15	LEFT_EYE
7	LEFT_WRIST	16	RIGHT_EAR
8	RIGHT_HIP	17	LEFT_EAR

Figure 4.9 Indexes corresponding to each part of the body.

skeleton representation. Each key point is indexed by a number and represents a different part of the human body (Figure 4.9). Since the camera's data includes the position of each key point, it is possible to estimate the person's pose based on the arrangement of these points.

First, a general evaluation is performed by analyzing the shoulder, knee, hip, and ankle. After confirming that all key points are detected, the distances between the shoulder and hip, hip and knee, and knee and ankle are computed along both the x (horizontal) and y (vertical) axes. If the distance between the hip and shoulder along the x-axis is greater than that along the y-axis, the person is considered to be lying down.

On the other hand, if the distances between the hip and shoulder, hip and knee, and knee and ankle along the y-axis exceed a certain threshold, the person is considered to be standing, as shown in Figure 4.10. Similarly, with a smaller threshold, the person is considered to be sitting. These thresholds are defined in the ZED2 camera SDK. This evaluation is performed twice to ensure that the key points of the hip, shoulder, knee, and ankle on the right side meet these conditions, as well as those on the left side.

A second evaluation is performed by calculating the angle between three points (Betta et al. (2023)): the shoulder, hip, and knee on both sides of the body. If the angle exceeds a certain threshold, this indicates that the person is either standing or lying down. However, if the angle is smaller than the threshold, the person is considered to be sitting.

A third evaluation is performed based on the distance between the nose and the ankles. The distance from the nose to both ankles is computed along the x and y axes. If the distance

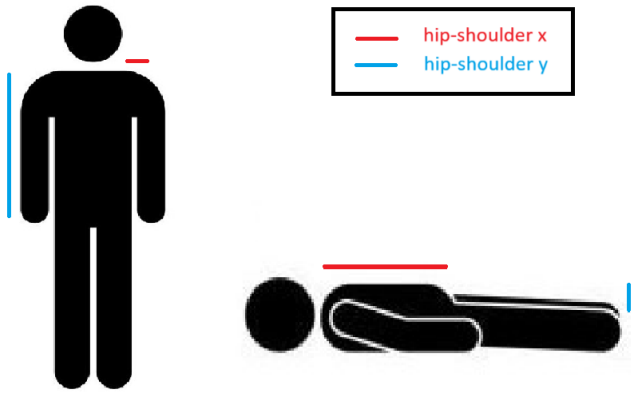


Figure 4.10 Distances between shoulder and hip over x and y axes.

on the x-axis is smaller than that on the y-axis, it indicates that the person is either sitting or standing. Depending on this distance and a defined threshold, the person will be classified as either sitting or standing. Conversely, if the distance between the nose and the ankles is greater along the x-axis, the person will be considered to be lying down, using the same idea as in Figure 4.10.

The results of these three evaluations, which totals to six due to the evaluations on both sides, are stored in an array. Any evaluations that could not be calculated are disregarded, and a voting procedure is used to determine a final result. To enhance consistency, this procedure is repeated more than twenty times. By obtaining over twenty different results and applying the voting procedure repeatedly, the final result is decided. If all results are inconclusive, the person's state is defined as unknown.

4.9 Dialog action

The dialog action has been developed in ROS2 using Python, in a file called dialog_action_node.py. This action has been implemented as a simulated action since speakers have not been included as part of the robot's payload. However, it is considered an important action to take into account due to its significance in detecting a person's consciousness. As future developments of this work, the implementation of this module is necessary.

The purpose of the action is to evaluate a person's consciousness by asking three questions and assessing their answers. The first question is "What is your name?" which tests self-awareness. If the person cannot recall their name, it might indicate serious cognitive impairment, confusion, or disorientation. The second question is "What day of the week is

it today?" which checks orientation to time. Knowing the current day requires awareness of time and short-term memory. Failure to answer could suggest memory issues or mental confusion. The third question is "Do you know where you are?" This evaluates orientation to place, assessing whether the person understands their environment. A conscious and cognitively intact person should be able to recognize their location.

If, after answering these three questions, the person is categorized as conscious, a fourth question will be asked: "Are you able to move?" If they respond affirmatively, the action will inform the PDDL controller that the person is conscious. Otherwise, the person will be categorized as either confused or unconscious. They will be classified as unconscious if there is no response and as confused if their answers do not make sense in the context of the questions asked.

Only if the person is conscious, the controller will replan to guide the person to the exit, ensuring they receive human assistance as soon as possible outside the disaster environment. This replan will involve changing the goal to one where the robot's sole objective is to reach the exit. Once the robot arrives at the exit, it will verify whether the original plan (exploring all locations) was completed. If not, the robot will restore the previous goal and replan to continue the exploration. If the person is unconscious or confused, the state of the person and their location will be reported to the controller, and the robot will continue with the exploration, leaving this task for the rescuers.

It is important to note that even though the action is simulated, the questions and computations to detect consciousness have been developed using fixed answers. To create a fully functional action, the robot should be equipped with speakers and a microphone. This would allow the robot to ask questions through the speakers and receive answers via the microphone to assess consciousness. Even though it is very simple, the architecture of the action can be found in the Figure 4.11.

4.10 Report action

The report action has been developed in ROS2 with Python, contained in a single file called report_action_node.py. This file subscribes to the `/reported_info` topic and writes all the information provided by the topic into a text file. The text file is saved in a fixed location within the Docker container, specifically at `/home/linuxbrew/ros2_ws/src/plansys/plansys2_simple_example_py/plansys2_simple_example_py/StatusReport.txt`. Although the architecture of this action is straightforward, it is illustrated in Figure 4.12.

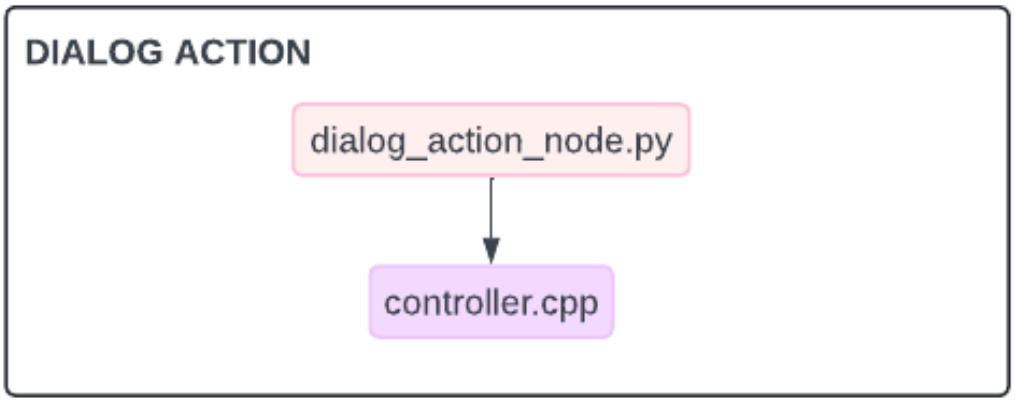


Figure 4.11 Structure of the dialog action architecture.

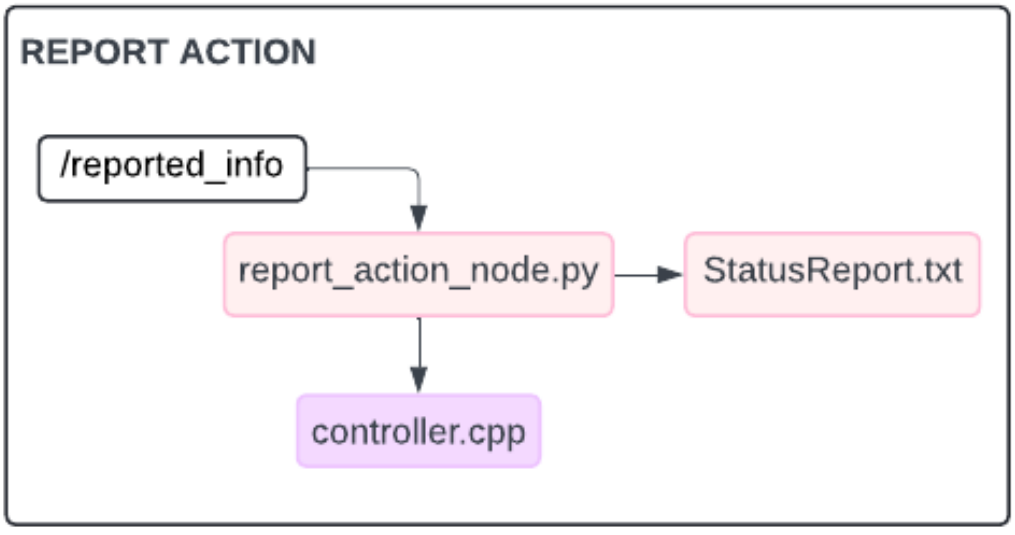


Figure 4.12 Structure of the report action architecture.

4.11 Portability to ROS2

As previously mentioned, a key objective of this project is to develop a system that fully integrates with ROS2, specifically ROS2 Humble, due to its long-term support, performance improvements, and the reduction in bugs compared to earlier versions of ROS2. Additionally, while ROS2 Humble is not the latest release, it offers more stable package releases and greater support compared to newer versions, making it a more reliable choice for development.

However, this presents a challenge given the system specifications of the Spot robot. The Spot CORE operates on a Linux-based OS, specifically Ubuntu 18.04. Due to the outdated nature of this OS (the latest being Ubuntu 24.04), Ubuntu 18.04 does not support the desired version of ROS2. According to the ROS2 documentation (ROS2 (2024b)), ROS2 Humble is only supported from Ubuntu 20.04 onwards.

To achieve portability to ROS2, several steps were necessary. First, the Spot CORE had to be equipped with a tool capable of supporting Ubuntu 20.04, the required version for running ROS2. Docker was selected to address this need by running an Ubuntu 20.04 image, which created a container, allowing the system to support the desired ROS2 environment within the existing infrastructure. Once an empty Ubuntu 20.04 container was accessible from the Spot CORE, ROS2 had to be installed, along with all the necessary dependencies for the project. This included packages such as Nav2, RTAB-Map or Plansys2, among others, to ensure the system had the required tools for successful operation.

Another important challenge faced, once ROS2 was running on the Spot CORE container, was accessing a visual interface for container-based tools, such as RViz, to visualize the map and understand the system's behavior. To achieve this, the Docker image had to be run with the appropriate flags to create the container with the desired behavior. These flags were:

- `-i (interactive)`: Keeps STDIN open even if not attached, allowing interaction with the container.
- `-t (tty)`: Allocates a pseudo-TTY, providing a terminal interface useful for running interactive applications.
- `-v /tmp/.X11-unix:/tmp/.X11-unix`: This flag mounts a volume from the host to the container, sharing the X11 socket directory. This allows the container to communicate with the host's X11 server, enabling graphical applications like RViz to be rendered on the host display.

- `-e DISPLAY=unix$DISPLAY`: Sets the DISPLAY environment variable inside the container, which instructs graphical applications where to send their output, effectively allowing them to be rendered on the host's display.
- `-e GDK_SCALE and -e GDK_DPI_SCALE`: These options control the scaling of graphical elements for GTK applications, which can be essential for ensuring proper display on high-DPI screens.
- `-net=host`: This flag enables the container to share the host's network stack directly, simplifying networking and improving performance by removing the need for explicit port mapping. It also ensures seamless access to host services.
- `-privileged`: Grants the container extended privileges, allowing it to perform actions typically restricted, such as accessing devices or modifying kernel parameters. This flag is useful when the container needs elevated permissions to function properly.

By using these flags, the container was able to run graphical applications like RViz, while retaining the necessary interactive and networking features required for the project.

However, this approach did not resolve all issues, particularly with using VSCode for coding. Although the VSCode GUI was accessible, problems arose when saving and compiling the code. Specifically, the last version saved in the log folder during the `colcon build` process did not match the actual latest version of the written code. This issue took time to identify and made it impossible to reliably use VSCode or other visual coding tools. As a result, all development had to be done using nano, a more basic text editor with limited features compared to VSCode. This made the development process more challenging, as nano offers fewer tools for tasks like code navigation, syntax highlighting, and debugging.

A key step in the project was adapting the Spot SDK from ROS1 to ROS2, as the provided examples were designed for ROS1 (Github (2024)). This required significant changes due to differences in communication systems, node structure, and APIs between ROS1 and ROS2. ROS1 uses a global message bus for communication, while ROS2 relies on DDS. The node structure also changed, with ROS2 introducing explicit node lifecycle management, requiring refactoring of node initialization and execution. Additionally, services, actions, and parameters had to be updated to ROS2's API, which uses new syntax and frameworks. Timing mechanisms in ROS1, which controls sleep rate for loops, also differ in ROS2, requiring adjustments to how loops were handled.

Finally, as mentioned previously, all necessary topics had to be created inside the Docker container to obtain information from the robot. Since standard topics available outside the

container were not accessible from within, the data had to be retrieved using the Spot SDK inside the container. This data was then published to new ROS2 topics within the container, ensuring the required information could be accessed internally when needed.

Despite presenting a significant challenge during the project's development, this approach offers many positive features. The system is built on the latest version of ROS, which is preferred over ROS1, even though ROS1 has more documentation. ROS1 is reaching its end of life, meaning that using it would have resulted in limited long-term support, despite the availability of more tools and information.

The requirement to use Docker adds another advantage: portability. This allows any user to access the entire project and install it on the Spot CORE, regardless of the OS version, as long as Docker is installed. Not only does Docker provide access, but it also simplifies deployment—installing the Docker image sets up all the necessary nodes for the project. By following the detailed steps outlined in the Appendix A, users can easily execute the project without any issues.

Chapter 5

Experiments

Summary

This chapter presents several experiments conducted, all utilizing the Spot CORE. Various types of data will be presented, including execution times, replanning durations, and the different scenarios designed to stress the system and demonstrate the reliability of the project.

5.1 Experiments with simulated actions

To stress the system and analyze the ability to replan and execute in large environments, 30 experiments were conducted. These experiments were tested on the Spot CORE, with the difference that the actions used were simulated. Instead of relying on real information obtained from the robot, random feedback was provided to the PlanSys2 planner to compute the next steps. This approach allowed testing of various scenarios where a large number of people could be found, different locations could be included, and the system could be stressed to assess whether performance varies across scenarios compared to the normal behavior when sensor data from the robot is used. The hypothetical scenario involving the displacement of locations established for these experiments is shown in Figure 5.1. All experiments were conducted with the objective of exploring every location, and the planner chosen to compute the plan is POPF.

The search action was implemented to detect a random number of people during the robot's search. The evaluation was configured to randomly assign a state to each person: either sitting, standing, or lying down. On the contrary, for the dialog action, although it was simulated, there was a bag of possible answers to the different questions —some

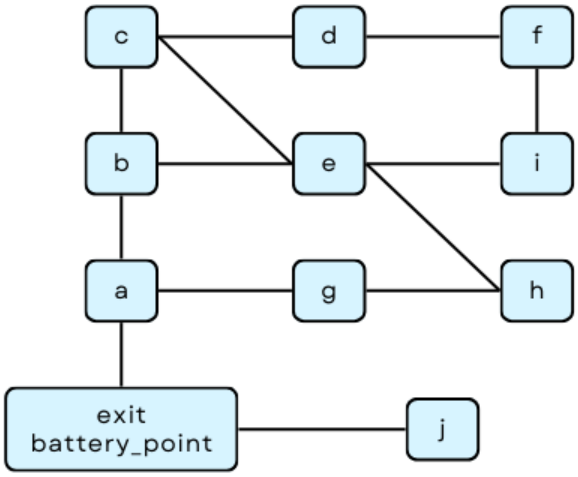


Figure 5.1 Scenario displacement for the experiments with fake actions.

correct and some incorrect— that were chosen randomly to determine the consciousness state. Additionally, the check battery action was set to return a random battery percentage during each check, simulating diverse conditions. This configuration allowed the system to be tested in a variety of environments and scenarios, ensuring its functionality under different and challenging circumstances.

Data collection is presented in Table 5.1, summarizing key information from each experiment conducted. The first value represents the number of replans executed by the controller to reach the goal based on the given scenario. The second value shows the average replan time, measured in microseconds, across all experiments. The third value indicates the number of locations used in the experiment, while the fourth value represents the total time the system took to complete the search and report all results, measured in seconds. The fifth column indicates the total number of people detected during the robot's search.

As a general observation, the average replan time remains under 300 milliseconds, regardless of the number of replans or locations involved in the problem. This is significant because it indicates that the size of the problem or the frequency of replanning does not cause a delay in the replanning process. Additionally, in experiments designed to stress the system, the robot consistently completed the necessary replans and successfully finished the task, regardless of how many locations or people were detected. Naturally, the total execution time depends on the number of locations to search and the number of people found, as more locations require longer searches, and detecting more people leads to additional replans and actions. In Figure 5.2 the relation between the execution time and the number of replans

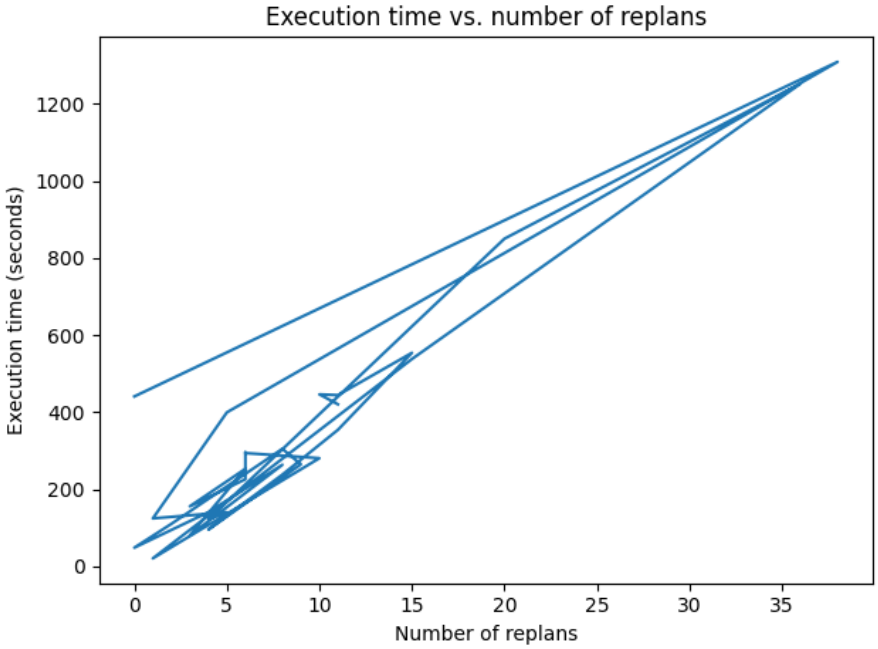


Figure 5.2 Time of execution vs number of replans.

can be observed. Although for a big number of replans is visible that the time of execution increases, other variables as the number of people found or the numbers of rooms to explore also depend on the total time.

	N replans	**Avg. replan time (ms)**	**N locations**	**Total time (s)**	**People found**
Exp1	0	0	10	440.737	0
Exp2	38	291.68	3	1308.45	35
Exp3	5	260.89	4	399.401	4
Exp4	1	230.146	5	124.403	0
Exp5	5	270	5	140.515	3
Exp6	4	266	5	121.725	3
Exp7	20	260.42	5	849.848	17
Exp8	36	291.72	3	1251.88	33
Exp9	15	238.78	4	537.745	12
Exp10	1	227.532	4	20.8021	0
Exp11	10	262.79	4	280.574	8
Exp12	6	237.211	4	293.868	5
Exp13	6	236.06	4	295.798	5
Exp14	6	240.55	3	226.049	5
Exp15	3	218.94	3	155.813	2
Exp16	6	232.776	3	253.045	4
Exp17	3	234.498	3	80.4416	2
Exp18	8	262.824	3	263.567	5
Exp19	4	254.814	3	140.389	2
Exp20	0	0	3	48.5511	0
Exp21	2	232.25	3	110.239	2
Exp22	8	229.68	3	304.832	6
Exp23	9	260.061	3	264.74	7
Exp24	4	241.735	5	95.5068	2
Exp25	8	265.811	5	238.009	6
Exp26	11	274.486	5	354.12	8
Exp27	15	267.921	5	553.838	12
Exp28	11	251.739	5	444.193	9
Exp29	10	249.555	5	446.146	7
Exp30	11	260.628	5	420.42	10

Table 5.1 Experiments information

5.2 Experiments with real actions

The experiments involving real-world actions were conducted on the second floor of the E building, located at Via dell'Opera Pia 13, University of Genoa.

These experiments took place in various rooms, utilizing the corridor and two separate rooms to create a realistic environment for the robot to operate in. For the experimental trials, the navigation strategy was slightly modified, with goal positions being commanded directly through the SDK instead of using the previously developed local planner. This adjustment was made due to challenges encountered when creating the map with RTAB-Map. Despite tuning various parameters to improve real-time map updates as the robot moved, navigating narrow spaces like the corridor proved problematic. The map failed to update quickly enough, and the walls caused issues in map creation, preventing the global planner from generating a feasible path to the final goal.

Although the global planner, and subsequently the local one, were not used in these experiments due to these mapping difficulties, it is worth noting that the navigation system, including the local planner and RTAB-Map, was tested successfully in a larger, more open room. In that environment, it was able to plan a path, move between points, and update the map dynamically. However, for these trials, the focus was on creating a realistic scenario, and the corridor and two different rooms were prioritized for that purpose. As a drawback of this solution the navigation was not always reliable and it sometimes failed due to the inability of the robot to generate a global path.

The experiments were conducted using the Spot robot, equipped with the Spot CORE, the LiDAR, and the ZED2 camera. To achieve the desired behavior, various adjustments were made during the code development, such as fine-tuning thresholds for people detection and pose estimation. Multiple individual tests were performed to ensure the system's reliability. For these experiments, an initial setup involved defining different waypoints within the environment, which were subsequently used to initialize the PDDL problem. These waypoints can be observed in the Figure 5.3.

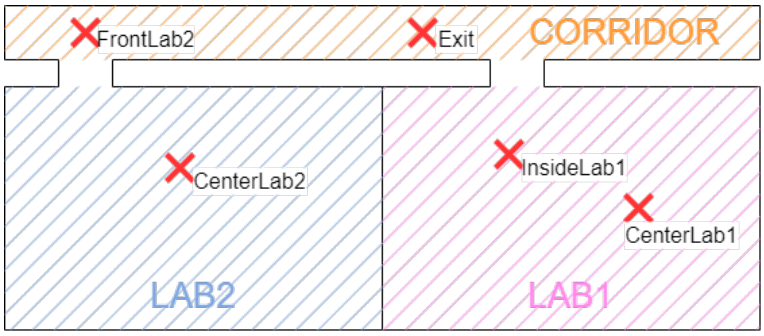


Figure 5.3 Real scenario on the second floor of the E building.

The PDDL problem definition for all the experiments conducted is as follows:

```
(set instance spot robot)
(set instance exit location)
(set instance insidelab1 location)
(set instance centerlab1 location)
(set instance frontlab2 location)
(set instance centerlab2 location)
(set instance battery_location location)

(set instance stand state)
(set instance sit state)
(set instance lay state)
(set instance unkown state)

(set instance conscious consciousness)
(set instance unconscious consciousness)
(set instance confused consciousness)

(set predicate robot_at spot exit)
(set predicate connected exit frontlab2)
(set predicate connected frontlab2 exit)
(set predicate connected centerlab2 frontlab2)
(set predicate connected frontlab2 centerlab2)
(set predicate connected exit insidelab1)
(set predicate connected insidelab1 exit)
(set predicate connected centerlab1 insidelab1)
(set predicate connected insidelab1 centerlab1)
(set predicate connected exit battery_location)
(set predicate connected battery_location exit)
(set predicate is_free spot)
(set predicate battery_unchecked spot)
```

It is important to note that the connection between rooms must be established in both directions to provide the planner with all the necessary information to compute a reachable plan.

Although the goal is to perform the search only in the center of the rooms, one of the advantages of PDDL is that different waypoints can be added to establish a connection path, which may facilitate navigation or create customized points for the robot to search. It is important to note that this setup is entirely customizable based on the specific requirements of the situation, allowing for the addition of various points that the robot can be instructed to search. For these experiments, in line with the problem's objectives, the robot will search only in the center of the rooms to search for people. According to the domain definition explained in Section 4.2, at each waypoint, the robot will check its battery status based on the Euclidean distance it has to travel. It will then decide whether to continue with the plan or return to recharge. The `battery_location` point is considered to be the same as the exit waypoint.

For all the experiments, as the plan is created, the PDDL computes an executable plan. However, this does not necessarily mean it is the best plan, as the quality may vary depending on the planner used. In these experiments, the planner employed is POPF.

5.2.1 First Experiment

The first experiment was conducted in a single room, where the robot starts in the corridor at the exit waypoint. Its goal is to search in Lab 1, specifically at its center, defined as:

```
(:goal (and (searched spot centerlab1))
```

The robot has sufficient battery to complete the entire search, and there is one person in the room who is sitting and conscious. The initial plan computed by the planner is as follows:

```
Action: (check spot exit insidelab1)
Action: (move spot exit insidelab1)
Action: (check spot insidelab1 centerlab1)
Action: (move spot insidelab1 centerlab1)
Action: (search spot centerlab1)
```

When the robot finds the person, the problem knowledge is updated by adding the instance of the new person, along with the predicate corresponding to the new knowledge:

```
(set instance p0 person)
set predicate (person_detected p0 centerlab1)
```

At this point, since one person was detected, a replanning occurs with the new goal as follows, taking a time of 488.176 milliseconds for replanning:

```
(:goal (and (searched spot centerlab1)
            (person_evaluated p0)
            (person_reported p0)
            (dialog_finished p0))
)
```

At this point, the original goal is satisfied since the robot has already completed the search in the center of Lab 1. However, the plan did not finish due to the replanning; the new goal is now to complete the search in Lab 1 but also to evaluate the person, establish a dialogue, and report their state. Thus, the plan computed according to the new goal is as follows:

```
Action: (evaluate centerlab1 p0 spot)
Action: (dialog centerlab1 p0 spot)
Action: (report centerlab1 p0 spot)
```

After completing the assessment, the robot updates the problem knowledge with the information obtained:

```
set predicate (person_state p0 sit)
set predicate (person_dialog p0 conscious)
set predicate (person_evaluated p0)
set predicate (dialog_finished p0)
set predicate (person_reported p0)
```

Since the person was conscious, meaning they are able to go by themselves to the exit, the robot replans again, taking 578.089 milliseconds, in order to guide the person to the exit. The new goal is:

```
(:goal (and (robot_at spot exit)))
```

The plan of actions computed according to the goal is:

```
Action: (check spot centerlab1 insidelab1)
Action: (move spot centerlab1 insidelab1)
Action: (check spot insidelab1 exit)
Action: (move spot insidelab1 exit)
```

After this, the robot has searched all the desired locations, reported all the individuals, and guided the conscious people to the exit. Additionally, it creates a text file containing the search information, along with the topic `/reported_info` (Figure 5.4), which retains all the relevant details. The text file looks like this:

```
person location state consciousness
p0, centerlab1, sit, conscious
```

```
linuxbrew@isabel-Valhalla:~$ ros2 topic echo /reported_info
people_information:
- person: p0
  state: sit
  consciousness: conscious
  location: centerlab1
---
```

Figure 5.4 Output of the /report_info topic.

The total duration of this experiment was 104.6 seconds. This time includes the duration spent navigating between waypoints, conducting the search in Lab 1, and identifying one conscious individual who was subsequently guided to the exit.

The experiment can be observed by scanning the QR code in Figure 5.5.

Figure 5.5 First experiment.

5.2.2 Second Experiment

The second experiment was conducted in two different rooms, using the entire map shown in Figure 5.3. The robot started in the corridor at the exit waypoint, with the goal of searching both Lab 1 and Lab 2, specifically targeting their centers, defining the problem goal as:

```
(:goal (and (searched spot centerlab1)
            (searched spot centerlab2))
)
```

The robot had enough battery to complete the entire search. However, despite the presence of a person in each room, no one was detected due to a combination of poor lighting and a high threshold set for people detection. The threshold was calibrated earlier in the morning, under better lighting conditions and with more people present in the laboratory, which led to

inaccurate detection settings for the later experiment in dimmer conditions. The initial plan computed by the planner is as follows:

```
Action: (check spot exit frontlab2)
Action: (move spot exit frontlab2)
Action: (check spot frontlab2 centerlab2)
Action: (move spot frontlab2 centerlab2)
Action: (search spot centerlab2)
Action: (check spot centerlab2 frontlab2)
Action: (move spot centerlab2 frontlab2)
Action: (check spot frontlab2 exit)
Action: (move spot frontlab2 exit)
Action: (check spot exit insidelab1)
Action: (move spot exit insidelab1)
Action: (check spot insidelab1 centerlab1)
Action: (move spot insidelab1 centerlab1)
Action: (search spot centerlab1)
```

After the robot moves to the center of Lab 2 and completes its search, it determines that no one has been found, allowing the plan to proceed without the need for replanning. The robot then follows the waypoints outlined in the plan until it reaches the center of Lab 1. Once again, due to the lighting conditions, no person is detected, and the plan concludes as both searches have been completed, in a total time of 153.7 seconds.

The experiment can be observed by scanning the QR code in Figure 5.6.

Figure 5.6 Second experiment.

5.2.3 Third experiment

The third experiment was conducted in two different rooms. The robot started in the corridor at the exit waypoint, with the goal of searching both Lab 1 and Lab 2, targeting their centers, defined as:

```
(:goal (and (searched spot centerlab1)
            (searched spot centerlab2))
)
```

The robot had sufficient battery to complete the entire search. This time, based on the lighting conditions encountered in the previous experiment, the threshold for people detection was slightly lowered to improve accuracy under the adjusted lighting. The robot successfully detected two individuals, one in each room, both of whom were sitting and unconscious. The initial plan computed by the planner is as follows:

```
Action: (check spot exit frontlab2)
Action: (move spot exit frontlab2)
Action: (check spot frontlab2 centerlab2)
Action: (move spot frontlab2 centerlab2)
Action: (search spot centerlab2)
Action: (check spot centerlab2 frontlab2)
Action: (move spot centerlab2 frontlab2)
Action: (check spot frontlab2 exit)
Action: (move spot frontlab2 exit)
Action: (check spot exit insidelab1)
Action: (move spot exit insidelab1)
Action: (check spot insidelab1 centerlab1)
Action: (move spot insidelab1 centerlab1)
Action: (search spot centerlab1)
```

When the robot finds a person in the first room it visits, which is Lab 2, the problem knowledge is updated by adding an instance of the newly detected person, along with the predicate corresponding to the current knowledge:

```
(set instance p0 person)
set predicate (person_detected p0 centerlab2)
```

At this point, since a person was detected, a replanning process is triggered, taking 767.055 milliseconds to complete with the updated goal as follows:

```
(:goal (and (searched spot centerlab2)
            (person_evaluated p0)
            (person_reported p0)
            (dialog_finished p0))
)
```

Upon completing the search in the center of Lab 2, the robot triggered a replanning process. The updated goal now involves evaluating the detected individual, initiating a dialogue, and reporting their condition. Consequently, the plan computed based on this new goal is as follows:

```
Action: (evaluate centerlab2 p0 spot)
Action: (dialog centerlab2 p0 spot)
Action: (report centerlab2 p0 spot)
Action: (check spot centerlab2 frontlab2)
Action: (move spot centerlab2 frontlab2)
Action: (check spot frontlab2 exit)
Action: (move spot frontlab2 exit)
Action: (check spot exit insidelab1)
Action: (move spot exit insidelab1)
Action: (check spot insidelab1 centerlab1)
Action: (move spot insidelab1 centerlab1)
Action: (search spot centerlab1)
```

After completing the assessment, the robot updates the problem knowledge with the newly obtained information, incorporating the person's condition and any relevant details:

```
set predicate (person_state p0 sit)
set predicate (person_dialog p0 unconscious)
set predicate (person_evaluated p0)
set predicate (dialog_finished p0)
set predicate (person_reported p0)
```

Since the person was unconscious, the robot continues with the plan, as the individual cannot move on their own, and reports their status. The robot then navigates through the waypoints to reach the center of Lab 1, where it detects another person. It updates the problem knowledge by adding the new person instance, along with the corresponding predicate based on the current knowledge:

```
(set instance p1 person)
set predicate (person_detected p1 centerlab1)
```

Upon detecting a person, a replanning process is triggered, taking 532.429 milliseconds to complete, with the new goal defined as follows:

```
(:goal (and (searched spot centerlab2)
            (person_evaluated p0)
            (person_reported p0)
            (dialog_finished p0)
            (searched spot centerlab1)
            (person_evaluated p1)
            (person_reported p1)
            (dialog_finished p1))
)
```

At this point, although the searches in the centers of both Lab 2 and Lab 1 have been completed and the first person state has been reported, the new goal has not yet been satisfied. There remains a need to evaluate the newly found person, initiate a dialogue, and report their condition. Therefore, the plan computed according to this updated goal is as follows:

```
Action: (evaluate centerlab1 p1 spot)
Action: (dialog centerlab1 p1 spot)
Action: (report centerlab1 p1 spot)
```

After completing the assessment, the robot updates the problem knowledge with the newly acquired information, incorporating details about the individual's condition and any relevant observations:

```
set predicate (person_state p1 sit)
set predicate (person_dialog p1 unconscious)
set predicate (person_evaluated p1)
set predicate (dialog_finished p1)
set predicate (person_reported p1)
```

Since the person was unconscious, indicating that they are unable to move on their own, the robot reports their condition both through the topic (Figure 5.7) and in the text file as follows:

```
person location state consciousness
p0, centerlab2, sit, unconscious
p1, centerlab1, sit, unconscious
```

```
linuxbrew@isabel-Valhalla:~$ ros2 topic echo /reported_info
people_information:
- person: p0
  state: sit
  consciousness: unconscious
  location: centerlab2
- person: p1
  state: sit
  consciousness: unconscious
  location: centerlab1
---
```

Figure 5.7 Output of the /report_info topic.

After that, all predicates in the goal are satisfied, and the experiment concludes with a total duration of 241.2 seconds.

The experiment can be observed by scanning the QR code in Figure 5.8.

Figure 5.8 Third experiment.

5.3 Final considerations

This section will provide an overall comment about the experiments done, both simulated and real ones.

Regarding to the use of PDDL for the computation of the plan, it showed in all the experiments a reasonable action seqeunce to reach the final goal. The update of the PDDL problem state done by the controller (Section 4.4) while the information was being obtained was also successfully completed during the experiments execution. In addition, when there was a need for are planning, the new plan computed was also coherent with the current knoweledge that the robot had.

In all the experiments, the use of PDDL for plan computation always calculated a reasonable sequence of actions to achieve the final goal. The controller (Section 4.4) successfully updated the PDDL problem state as the information was being collected during the execution of the experiments. Furthermore, when replanning was necessary, the newly computed plan was coherent with the robot's current knowledge.

Concerning the replanning times, it is observable that in real experiments, the time taken is slightly higher than in the simulated ones. This difference is because of the computational effort required by the robot, as it was simultaneously processing information from the camera and LiDAR while using the motors for stability and movement. However, all replanning instances remained below 800 milliseconds, which is an acceptable threshold to maintain the quick responsiveness necessary for the robot. Figure 5.9 shows the relation between total execution time and number of replans for both simulated and real experiments. As can be observed, the time does not only depend on the number of replans, but also on the people and the number of rooms to explore.

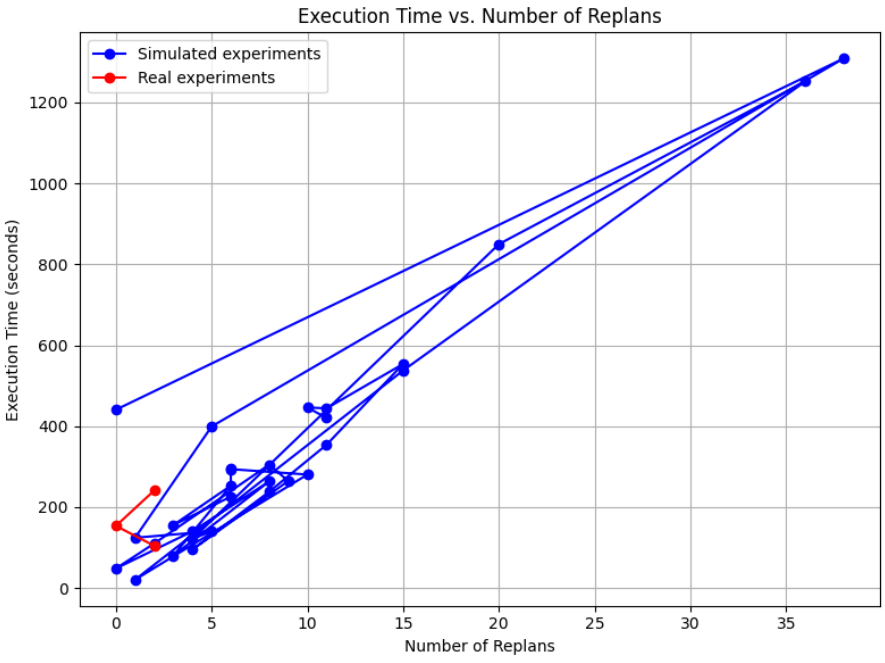


Figure 5.9 Time of execution vs number of replans.

In relation to the detection of people, as explained in Experiment 5.2.2, lighting was a decisive factor for accurate detection. Light conditions vary when the robot is positioned against the light, such as when facing windows, and also depend on the time of day. Therefore, the threshold must be adjusted according to these factors to minimize the number of false

positives while still effectively detecting the actual individuals present. However, once the person was detected, the assessment of the pose was consistently accurate throughout all the experiments, demonstrating a high level of precision in pose computation.

The reporting of information during the people assessment was consistently successful, providing details in both the text file and the topic. This included information about the person, their location, level of consciousness, and pose.

For the consciousness detection, although it was simulated as explained in the Section 4.9, the evaluation in the simulated experiments was generated through random responses that a person could give. These responses were drawn from a set of possible answers regarding their name, the day of the week, and yes or no, depending on whether the person knew where they were or if they were able to move. Based on these responses, the consciousness level was successfully tagged as conscious, unconscious, or affected.

Regarding the times, the search is completed in less than 15 seconds, as the robot always performs a 360º turn while collecting information about all the people detected. The assessment of the individuals takes around 40 seconds, assuming the dialog is realistic. Additionally, since the assessment of all people found in the same room is conducted sequentially, the robot only needs to guide them once, optimizing the search process.

Chapter 6

Conclusions

As mentioned at the beginning, catastrophes can cause significant damage, resulting in victims or injuries, which need prompt action from rescuers. This is a challenging task that must be carried out as quickly as possible; however, sometimes rescuers cannot respond in time or may even become victims themselves. In these situations, time and information is crucial to save as many lives as possible.

Consequently, in this project, the robot Spot has been employed to develop a system that uses PDDL to create a plan for acting in disaster situations. The system aims to locate individuals and respond according to their state, either by reporting their condition or by assisting them in reaching the exit if they are able to move.

The system has been developed using the latest version of ROS, which enables the use of the most recent tools for performing searches and ensures long-term support. Additionally, by utilizing Docker, the project offers quick portability, allowing the entire system to be installed on the Spot robot with just a few commands.

After the robot reaches the point where the search must be conducted, it begins the search, detecting individuals and reporting information through the topic and a text file, while also updating its knowledge to create a new plan that better aligns with the current understanding of the environment. The robot also estimates the pose and consciousness level of each person, which proved to be accurate during the execution of the experiments. If a person is detected as conscious, they are guided to the exit; if not, their status is reported to the rescuers, providing them with essential information to take appropriate action. If, at any point during the search, the robot detects that it has low battery or cannot reach the next point due to distance and battery constraints, it will return to recharge.

In general, the system consistently computed a plan based on the needs of the goal and performed well in pose and consciousness estimation. Additionally, the time required

for replanning was very short, enabling rapid exploration. However, there are several future improvements that could enhance the system. One potential improvement is the implementation of speakers and microphones in the robot to make the dialog action, used for evaluating consciousness, entirely real rather than simulated.

Improving the management of lighting conditions for people detection is another area that could be improved. For example, automatically adjusting the threshold based on the search area (such as if it contains many windows or is in a darker location) and the time of day to possibly improve the detection of people. Furthermore, the parameters for map creation using RTAB-Map should be fine tuned to prevent issues in narrow passages, as experienced during the experiments. These adjustments would facilitate navigation alongside map creation, providing rescuers with valuable information about potential unknown obstacles that may arise after a catastrophe occurs.

References

Abdelsalam, A., Mansour, M., Porras, J., and Happonen, A. (2024). Depth accuracy analysis of the zed 2i stereo camera in an indoor environment. *Robotics and Autonomous Systems*, 179:104753.

Al-Dhief, F., Sabri, N., Abdul Latiff, N. M., Nik Abd Malik, N. N., Abd Ghani, M. K., Mohammed, M., Al-Haddad, R., Dawood, Y., Ghani, M., and Ibrahim Obaid, O. (2018). Performance comparison between tcp and udp protocols in different simulation scenarios. *International Journal of Engineering Technology*, 7:172–176.

Betta, Z., Paneri, S., Gaudino, A., Benini, A., Recchiuto, C. T., and Sgorbissa, A. (2023). Multi-floor danger and responsiveness assessment with autonomous legged robots in catastrophic scenarios. In *IEEE RoMan 2023*, pages 1063–1070.

BostonDynamics (2022). Learn how our robotic solutions simplify industrial inspections, continuous data collection, and worker safety across a range of industries. https://www.bostondynamics.con/solutions.

BostonDynamics (2024a). Payloads from boston dynamics.

BostonDynamics (2024b). Spot from boston dynamics.

BostonDynamics (2024c). Spot sdk.

Bouman, A., Ginting, M. F., Alatur, N., Palieri, M., Fan, D. D., Touma, T., Pailevanian, T., Kim, S.-K., Otsu, K., Burdick, J., and akbar Agha-mohammadi, A. (2020). Autonomous spot: Long-range autonomous exploration of extreme environments with legged locomotion.

Chen, Q. and Pan, Y.-J. (2024). An optimal task planning and agent-aware allocation algorithm in collaborative tasks combining with pddl and popf.

Chung, W. and Iagnemma, K. (2016). *Wheeled Robots*, pages 575–594. Springer International Publishing, Cham.

Coles, A. J., Coles, A., Fox, M., and Long, D. (2014). COLIN: planning with continuous linear numeric change. *CoRR*, abs/1401.5857.

Colledanchise, M. and Ögren, P. (2017). Behavior trees in robotics and AI: an introduction. *CoRR*, abs/1709.00084.

Cruz Ulloa, C., del Cerro, J., and Barrientos, A. (2023). Mixed-reality for quadruped-robotic guidance in SAR tasks. *Journal of Computational Design and Engineering*, 10(4):1479–1489.

Dadheech, P. (2022). Real-time survivor detection system in sar missions using robots. *Drones*, 6:1–20.

Dal Lago, U., Pistore, M., and Traverso, P. (2002). Planning with a language for extended goals. pages 447–454.

Docker (2024). Docker.

Durrant-Whyte, H. and Bailey, T. (2006). Simultaneous localization and mapping: part i. *IEEE Robotics Automation Magazine*, 13(2):99–110.

FF (2024). Ff homepage.

Garrett, C. R., Chitnis, R., Holladay, R., Kim, B., Silver, T., Kaelbling, L. P., and Lozano-Pérez, T. (2021). Integrated task and motion planning. *Annual Review of Control, Robotics, and Autonomous Systems*, 4(Volume 4, 2021):265–293.

Geffner, H. (2000). *Functional Strips: A More Flexible Language for Planning and Problem Solving*, pages 187–209. Springer US, Boston, MA.

Georgievski, I. and Aiello, M. (2015). Htn planning: Overview, comparison, and beyond. *Artificial Intelligence*, 222:124–156.

Gerevini, A. and Serina, I. (2002). Lpg: A planner based on local search for planning graphs with action costs. page 13 – 22. Cited by: 163.

Ghallab, M., Knoblock, C., Wilkins, D., Barrett, A., Christianson, D., Friedman, M., Kwok, C., Golden, K., Penberthy, S., Smith, D., Sun, Y., and Weld, D. (1998). Pddl - the planning domain definition language.

Github, S. S. (2024). Github sdk.

Ju, C., Luo, Q., and Yan, X. (2020). Path planning using an improved a-star algorithm. In *2020 11th International Conference on Prognostics and System Health Management (PHM-2020 Jinan)*, pages 23–26.

Labbé, M. and Michaud, F. (2018). Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation. *Journal of Field Robotics*, 36(2):416–446.

Li, N., Cao, J., and Huang, Y. (2023). Fabrication and testing of the rescue quadruped robot for post-disaster search and rescue operations. In *2023 IEEE 3rd International Conference on Electronic Technology, Communication and Information (ICETCI)*, pages 723–729.

Lyu, M., Zhao, Y., Huang, C., and Huang, H. (2023). Unmanned aerial vehicles for search and rescue: A survey. *Remote Sensing*, 15(13).

Marques, C., Cristóvão, J., Alvito, P., Lima, P., Frazão, J., Ribeiro, I., and Ventura, R. (2007). A search and rescue robot with tele-operated tether docking system. *Industrial Robot: An International Journal*, 34(4):332–338.

Martin, F., Clavero, J. G., Matellan, V., and Rodriguez, F. J. (2021). Plansys2: A planning system framework for ros2. In *IEEE International Conference on Intelligent Robots and Systems*, pages 9742–9749. Institute of Electrical and Electronics Engineers Inc.

Metric-FF (2024). Metric-ff.

Mishra, B., Garg, D., Narang, P., and Mishra, V. (2020). Drone-surveillance for search and rescue in natural disaster. *Computer Communications*, 156:1–10.

of Labor Statistics, U. B. (2017). Fatal work injuries for fire and rescue workers.

Optic (2024). Optic.

Python (2024). Socket module for python.

Queralta, J. P., Taipalmaa, J., Can Pullinen, B., Sarker, V. K., Nguyen Gia, T., Tenhunen, H., Gabbouj, M., Raitoharju, J., and Westerlund, T. (2020). Collaborative multi-robot search and rescue: Planning, coordination, perception, and active vision. *IEEE Access*, 8:191617–191643.

Ramezani, M., Brandao, M., Casseau, B., Havoutis, I., and Fallon, M. (2020). Legged robots for autonomous inspection and monitoring of offshore assets. Offshore Technology Conference.

ROS (2021). Robot operating system.

ROS2 (2024a). Robot operating system 2.

ROS2 (2024b). Ros2 documentation.

StereoLabs (2024). Zed2 from stereolabs.

Ulloa, C. C., Dominguez, D., Barrientos, A., and del Cerro, J. (2023). Design and mixed-reality teleoperation of a quadruped-manipulator robot for sar tasks. In Cascalho, J. M., Tokhi, M. O., Silva, M. F., Mendes, A., Goher, K., and Funk, M., editors, *Robotics in Natural Settings*, pages 181–194, Cham. Springer International Publishing.

Wang, H., Yu, Y., and Yuan, Q. (2011). Application of dijkstra algorithm in robot path-planning. In *2011 Second International Conference on Mechanic Automation and Control Engineering*, pages 1067–1069.

Wang, Y., Guo, Z., and Xu, J. (2022). Underwater search and rescue robot based on convolutional neural network. In *2022 IEEE 4th International Conference on Civil Aviation Safety and Information Technology (ICCASIT)*, pages 786–790.

WISQARS (2022). Unintentional injury deaths in the u.s. for ages 1-44 from 1981-2022.

Yeom, S. (2024). Thermal image tracking for search and rescue missions with a drone. *Drones*, 8(2).

Appendix A

How to install Docker and setup the project

Linux Ubuntu

1. Add Docker's official GPG key with the commands below

```
sudo apt-get update
```

```
sudo apt-get install ca-certificates curl
```

```
sudo install -m 0755 -d /etc/apt/keyrings
```

```
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o
    /etc/apt/keyrings/docker.asc
```

```
sudo chmod a+r /etc/apt/keyrings/docker.asc
```

2. Add the repository to Apt sources with the command

```
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/
    keyrings/docker.asc] https://download.docker.com/linux/ubuntu
     \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

```
sudo apt-get update
```

3. Download the image of the project

```
docker pull isacg5/final_solution
```

4. Allow display GUI applications with Docker

```
xhost +
```

5. Start for the first time the container

```
sudo docker run -it --name=thesis_container -v /tmp/.X11-unix:/
    tmp/.X11-unix -e DISPLAY=unix$DISPLAY -e GDK_SCALE -e
    GDK_DPI_SCALE --net=host --privileged isacg5/final_solution
```

6. Exit and start normally the container

```
exit
```

```
docker start thesis_container
```

```
docker attach thesis_container
```

7. If more than one terminal windows are needed for the container, write

```
docker exec -it thesis_container /bin/bash
```

Run the code

First, be sure to execute `xhost +` in the terminal outside the Docker container.

Additionally, in order to connect with the ZED2 camera, the server is activated with the following command inside the Docker container, but be aware that the client must be running from the camera side:

- `cd /ros2_ws/src/zed/scripts`
- `python3 server_both.py`

Then, in different terminals inside the Docker container execute in the following order:

1. `cd /ros2_ws/src/plansys/plansys2_simple_example_py/plansys2_simple_example_py`

2. `python3 gotopoint.py`
3. `ros2 launch isabel_interface all.launch.py`
4. `ros2 launch plansys2_simple_example_py plansys2_launch.py`
5. `ros2 run plansys2_simple_example_py controller_node`

The code can also be found in GitHub: https://github.com/isacg5/thesis_project.