FINAL YEAR PROJECT
IMPERIAL COLLEGE LONDON

Evaluating the navigation capabilities of Spot and ROS Integration

Department:	Department of Aeronautics
Course:	MEng Aeronautical Engineering
Name:	Ming Jie See
CID:	01486590
Internal Supervisor:	Professor Sergei Chernyshenko
External Supervisor:	Professor Volker Krueger
Date:	5 June 2023

A thesis submitted in partial fulfilment of the requirements for the degree of Master of Engineering in Aeronautical Engineering from Imperial College London.

Acknowledgements

I would like to thank Professor Volker Krueger, Professor Mathias Haage from Lund University and Professor Sergei Chernyshenkov from Imperial College London for their guidance and supervision of my project. I am also very grateful to Mr Ola Nilsson and Mr Johan Lindberg at Cognibotics AB, who helped me resolve technical issues working with the Spot robot. Additionally, Dr Michal Staniaszek from the Oxford Robotics Institute was very helpful in integrating my work into the open-source ecosystem. Finally, and most importantly, I would like to thank my family for giving me the opportunity to study abroad and for constantly supporting me during my time in Imperial College London and Lund University.

Abstract

This report discusses the navigation capabilities of Spot and the integration of these capabilities in a ROS package, allowing users to interface Spot with existing ROS packages to extend its functionality. Previous efforts have allowed Spot to be moved through a ROS interface. However, they do not enable advanced Spot Autonomy functionality such as automated calibration, graph navigation and arm picking of objects, which highlight the unique capability of Spot as the first commercially available quadruped robot. This report describes the software design behind the ROS package to make programming for Spot more accessible to ROS developers, while evaluating the navigation capabilities of Spot. The native navigation stack of the Spot robot was evaluated to be competitive with the ROS navigation stack and RTAB-Map, performing similarly in a real-world task of navigating to a button and pressing it with the arm. The ROS package produced in this report facilitates future application of the Spot robot by making it more accessible to ROS developers, while maintaining full functionality of the manufacturer's SDK.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

In June 2020, Boston Dynamics released the Spot robot for commercial sale [1]. This robot was groundbreaking, being the first commercially available quadruped robot that could traverse rough terrain autonomously. Unlike other robotics platforms, it is designed to work in uncontrolled environments, and brings new capabilities such as climbing stairs, stepping over obstacles, and opening doors with its 6-degree of freedom arm. Industry took an interest in the robot, aiming to use it for documenting construction progress, monitoring hazardous areas such as power plants, and providing general situational awareness in remote environments.

As an example, the construction industry spends roughly 4% of contract value in correcting construction faults alone [2]; a large proportion compared to the typical profit margin for a construction company of 4% [3]. Additionally, the UK Atomic Energy Agency purchased Spot for use in hazardous radiation environments [4]. This shows the commercial potential of Spot in robotics automation.

In order to support the adoption of this new generation of quadruped robotics platform, an easy to use software package should be made available. Boston Dynamics provides an open-source Python Application Programming Interface (API) to control the robot within its Spot Software Development Kit (SDK). However, most robotics software that has been developed in academia make use of the Robot Operating System (ROS). This makes it difficult for out-of-the-box interoperability with the robot, requiring custom solutions for every application.

1.2 Objectives

The complexity of writing custom software packages for each use case of the Spot robot makes it difficult for companies to integrate robots into their work processes. This project aims to create a ROS software package that encapsulates the functionalities of the Spot SDK, allowing robotics engineers to make use of existing ROS packages to add functionality to the robot. The main objectives are as follows:

1. Design a ROS-based wrapper around the Spot SDK, mapping commonly used robotics functions such as setting the movement velocity, pose of the robot and odometry to ROS standard message topics. The software package should be open source friendly and utilise unit testing for Continuous Integration, facilitating collaboration.

2. Integration testing of the ROS wrapper with standard ROS programs to ensure its functionality on hardware. Benchmark its performance against the manufacturer's Spot SDK.

3. Create a simple demonstration to showcase the ease of use of the ROS wrapper; integrating with existing ROS packages such as Simultaneous Localisation and Mapping (SLAM) packages.

1.3 Outline

The report is structured into 7 sections. Chapter 1 provides background on the Spot robot and discusses the motivation behind the project. Chapter 2 highlights past work that has been done in the field, as well as theoretical background helpful to robotics software development on the Spot robot. Chapter 3 discusses the software development approach to writing the ROS package for Spot, and how it maps to available hardware functionality. Chapter 4 showcases the integration testing of software and hardware on Spot, benchmarking it against the manufacturer's Spot SDK. Chapter 5 discusses potential applications for the ROS package on Spot, delving deeper into a specific application of Spot for indoor SLAM. Chapter 6 showcases the application on hardware, analysing the results produced and showcasing its ease of use as opposed to a custom-tailored solution with the manufacturer's Spot SDK. Additionally, the navigation capabilities of the robot with the new ROS package is evaluated. Finally, the conclusions of the project and possible future work are discussed in Chapter 7.

2 Literature Review

2.1 Spot Robot

Spot is Boston Dynamic's first commercially available robot. It is a quadruped robot with its own sensor package for local navigation and obstacle avoidance. The robot is designed to encapsulate low level functions such as motor control, leg kinematics, collision avoidance, movement and odometry. This allows users to focus on the high level applications without worrying about hardware and control details such as the inertial measurement unit for odometry, or PID gains set on the motors for accurate movement. However, it is still important to understand the theory underlying the robot's capabilities in order to understand its strengths and limitations.

The base Spot robot has the following capabilities [5]:

Table 2.1: Spot robot capabilities

Capability	Specification
Payload	Carry up to 14kg, 150W of DC power available per payload
Sensors	5 stereo depth cameras with 360° field of view, 4m range overall
Battery	90 minute runtime per battery
Mobility	Stairs and complex terrain traversal
Movement	Max speed of 1.6m/s, max climb angle of 30°, max step height of 300mm
Stair stepping	60cm min width, 45° max pitch, 22cm max step height

There are numerous additional capabilities enabled through software, which this project aims to make more accessible to users by creating a plug and play ROS package for integration with existing codebases. The Spot robot that was used in this report is seen in Figure 2.1, and the anatomy of the robot is shown in Figure 2.2.

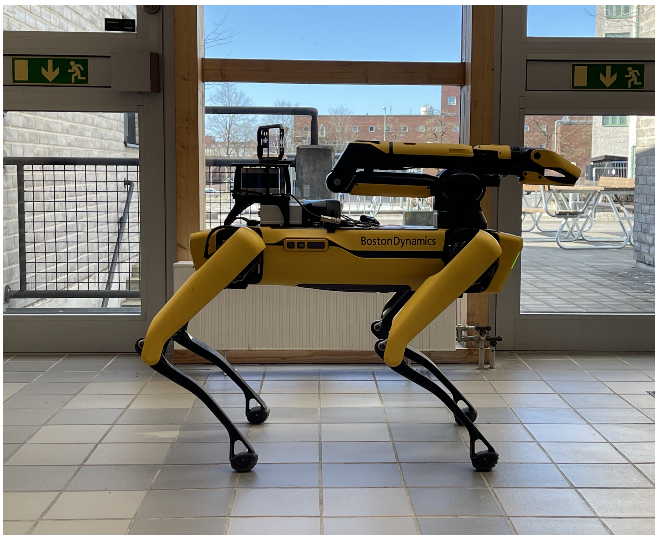


Figure 2.1: Spot robot used in this project

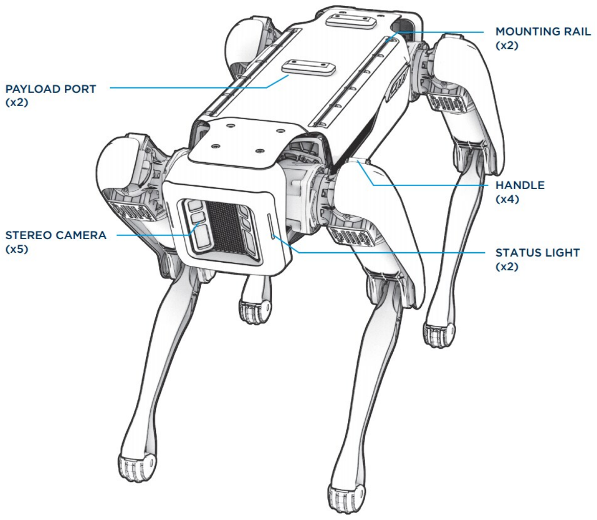


Figure 2.2: Spot anatomy [6]

2.1.1 Payloads

The robot is able to carry a variety of payloads, both manufacturer recommended and custom payloads. As seen in Figure 2.1, the robot used in this project is equipped with the Spot Arm [7]. The arm allows it to interact with its environment in a multitude of ways, including picking up an object, grasping a door handle and interacting with interfaces not specifically designed for robots, such as buttons and switches.

Spot has an API meant to abstract the kinematics of the arm, allowing users to dictate the movement of the arm using joint angles, or Cartesian frame transformations. This abstracts the motor controls from the pose of the arm, thus shifting the user's focus to higher level applications. However, the arm does not have the same collision avoidance features as the body, due to the lack of depth cameras installed directly in the arm. Thus, limits have to be set in software to avoid crashing the arm into obstacles such as overhangs or nearby objects. Additionally, there are pre-defined poses such as "Stow", "Ready" and "Carry" that provide convenient reference poses to go to before or after performing an action. The angular limits of each degree of freedom of the Spot arm are shown in Figure 2.3.

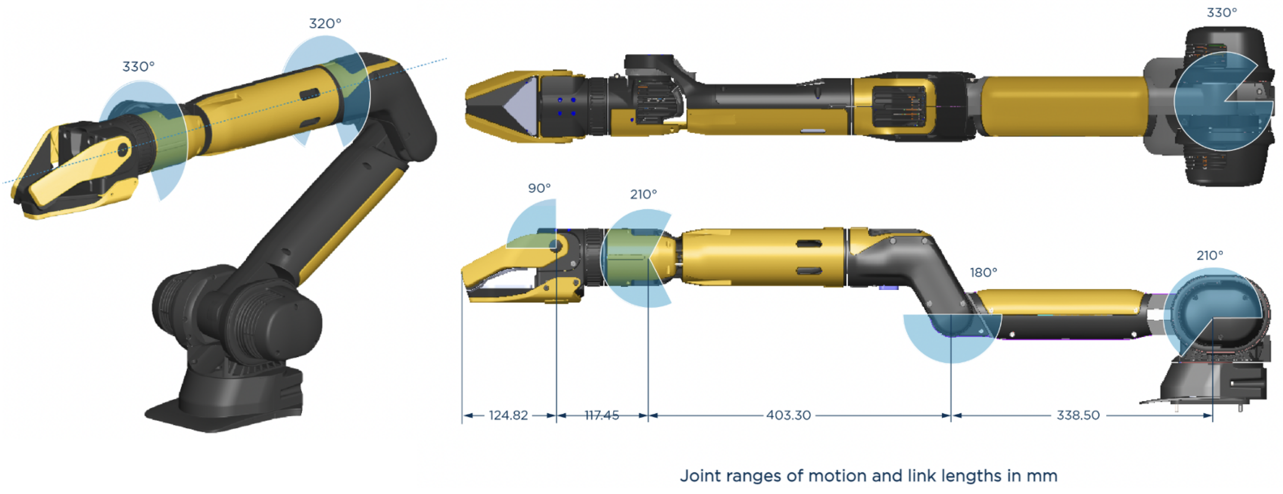


Figure 2.3: Spot arm degrees of freedom [7]

The Spot arm has the following capabilities:

Table 2.2: Spot arm capabilities

Capability	**Specification**
Accessories	50W DC power available, gigabit ethernet, camera sync
Sensors	4k RGB camera, Time of Flight sensor
Mobility	6 degrees of freedom
Capacity	11kg lifting, 5kg continuous, 25kg drag capacity on carpet
Gripper	130N peak clamp force, 175mm max aperture

The robot is also equipped with the Enhanced Autonomy Package Version 1 (EAP) [8], the latter of which comes with a LIDAR sensor that provides enhanced sensing capabilities. This enhances the navigation capabilities of the robot. The stereo cameras on the base Spot robot have a very limited range of 4m; sufficient for collision avoidance but not enough range for localisation and mapping purposes. Additionally, the Spot EAP includes the Spot Core, which is a small computer that allows users to run custom code that interfaces with the Spot robot, without having to alter the firmware running on the Spot robot for custom applications. Any intelligence that is added by the user has to be deployed on the Spot Core.

The Spot EAP has the following capabilities:

Table 2.3: Spot EAP capabilities

Capability	**Specification**
Lidar	Velodyne VLP-16, 100m$\pm$3cm, 5-20 Hz rotation rate
Field of view	$\pm 15^\circ$ vertical, 2° resolution, 360° horizontal, $0.1 - 0.4^\circ$ resolution
Spot Core	i5 Intel 8th Gen CPU, 16GB RAM, 512GB SSD, Ubuntu 18.04 LTS

The Velodyne VLP-16 LIDAR sensor has a much further range and higher resolution than the depth cameras on the Spot robot for environmental sensing. The LIDAR functions by rapidly spinning photon transmitters and photon detectors to measure the distance to points in its surroundings, generating 3D point cloud data relative to the reference frame of the sensor [9]. The point cloud data can be combined with other sensor data such as depth camera data or inertial measurements to generate a map of the surrounding area, using an algorithm such as Simultaneous Localization and Mapping (SLAM).

Figure 2.4: Velodyne VLP-16 Puck LIDAR [10]

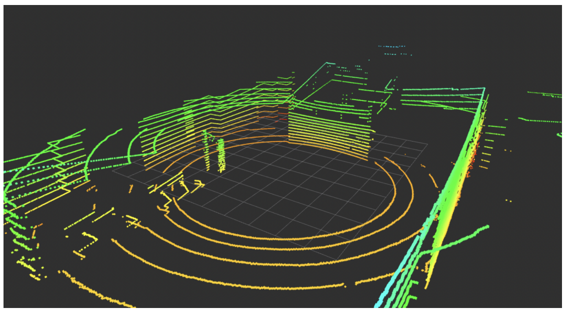


Figure 2.5: VLP-16 point cloud output [11]

Of interest is the Spot Core computer configuration. In order to interface with the Spot robot, the robot has a static IP configuration to forward the relevant TCP/UDP ports from the robot to its payloads. The Spot Core connects to these ports to send commands to the robot, which requires careful networking configuration when deploying applications to the Spot Core.

2.1.2 Reference Frames

In order to perceive the environment and move around it, the robot has to translate between reference frames. For example, to make use of sensor data such as the point cloud generated by the LIDAR, it has to transform between the sensor frame to the body frame, allowing it to know the relative position of the body to the environment. A (x, y, z) vector is used to denote the translation between frames, and a (w, x, y, z) quaternion is used to denote the rotation between frames. As opposed to the traditional roll, pitch and yaw (ψ, θ, ϕ) representation of rotation, quaternions do not suffer from gimbal lock [12]. Thus, a (x, y, z) position vector and a (w, x, y, z) quaternion is used to represent transformations between reference frames.

In the case of the Spot robot, it manages the transformations between reference frames with a `FrameTreeSnapshot`. The robot stores the transformation between each frame, such as the `vision`, `odom` and `body` frame, as a tree with the `body` frame as the root of the tree. This reduces networking overhead. For n frames, the robot only has to send $O(n)$ edges instead of $O(n^2)$ edges in the case of a fully connected graph. An example of a `FrameTreeSnapshot` can be seen in Figure 2.6. In order to find the transformation between the `fiducial` and `odom` frame, it is necessary to traverse the `FrameTreeSnapshot` and apply transformations of each edge in the path between the two nodes. Traversing the graph can be done with either a depth-first search or breadth-first search [13]. In the Spot SDK, this transformation math has been abstracted by Python magic methods [14] that allow the transformation objects to be directly multiplied to get the resulting transformation. The root of the tree can be identified as the only frame with an identity transformation to itself, which is usually the `body` frame.

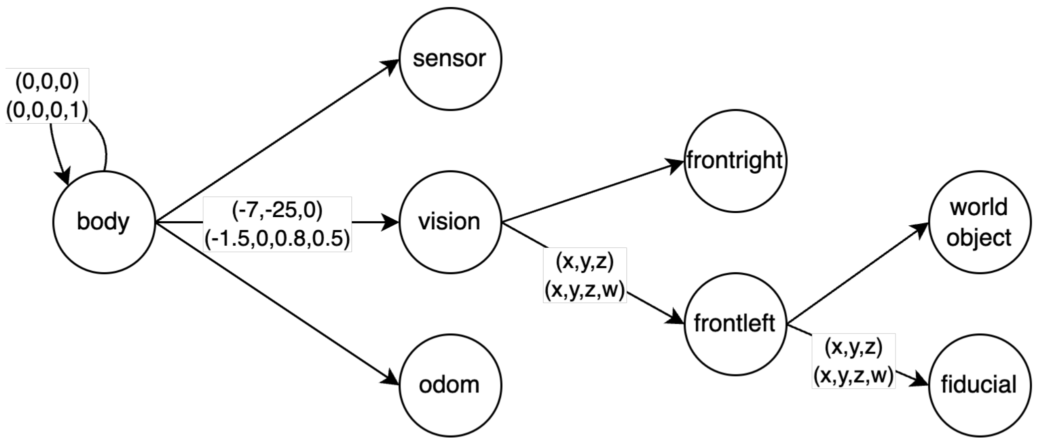


Figure 2.6: FrameTreeSnapshot example

During operation of the robot, the joints in motion cause the transformations to vary over time. In response, the Spot robot dynamic updates the `FrameTreeSnapshot` over time. It is important to apply the `FrameTreeSnapshot` at the correct timestamp to get an accurate pose of the robot in the desired reference frame.

2.1.3 Quadruped Kinematics

Despite the high level interface for controlling the robot's movements, it is important to understand quadruped kinematics to be aware of its limitations. The Spot robot's dynamics can be described using forward kinematics and inverse kinematics.

Forward kinematics use the rotations of each joint to calculate the position of each joint relative to the robot's body frame. Inverse kinematics calculate the joint rotation required to place the end of the kinematic chain, in this case the robot's feet, at the desired position [15]. This allows the robot to calculate the required rotations of each servo motor to achieve the desired gait.

In the case of the Spot robot, the movement kinematics of the body are completely encapsulated by the Spot SDK. In regular movement, the only method to change the gait is through the SDK, such as entering Crawl or Walk modes. In the Choreography API, there is the option to tweak custom movements that alter the movement of the robot [16]. However, there is no method to manually set the rotations of each individual actuator. This is likely to prevent the user from commanding the robot into entering an unstable state that may damage the robot.

2.1.4 Client-Server Architecture

The Spot SDK operates on a client-server architecture, where the Spot robot firmware functions as a server, with commands being sent to it by clients. Communication is handled with gRPC [17], a performant Remote Procedure Call framework that allows for simple client-server communication. The services available on the Spot robot are divided into three groups: Core, Robot and Autonomy, as seen in Figure 2.7. Core describes the lowest level of the software stack, dealing with authentication, identification and payload registration. Robot describes the basic commands to interact with the robot and get the robot state. Autonomy describes the autonomous actions that can be performed based on pre-recorded or pre-programmed intelligence; the highest level of the software stack. Further functionality can be made available with the use of payloads mentioned in Section 2.1.1, using the same software architecture.

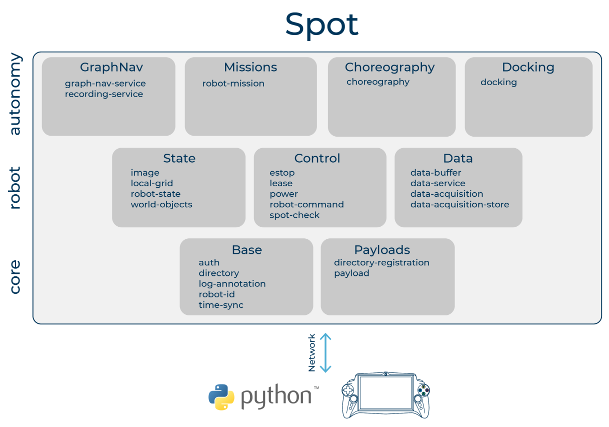


Figure 2.7: High level view of Spot SDK [18]

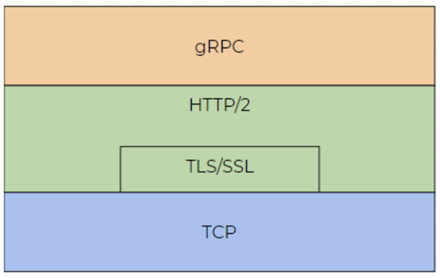


Figure 2.8: Spot network protocol stack [19]

This Client-Server architecture allows for encapsulation of lower level functions of the robot such as defining the control system for actuator movement, while exposing a high-level API for users to take control of the robot. gRPC calls can be made in parallel with HTTP/2 on the same network connection, and the Transmission Control Protocol (TCP) allows for reliable transport, ensuring that messages sent reach their intended destination. The gRPC server is part of the Spot robot firmware and cannot be directly modified by the user, thus all programmatic interactions with the robot have to be handled through client-server requests. This is the foundation of this project.

2.2 Robot Operating System

The Robot Operating System (ROS) is an open-source middleware software suite designed in 2007 by researchers at Stanford University and the Willow Garage [20] It was designed to facilitate code extraction and reuse from one application to another, across different robotics platforms. Generic functionality needed by different robots such as coordinate transforms or camera streaming pipelines could be written once and applied on any robot running ROS.

The fundamental concepts underlying ROS are nodes, messages, topics and services. Nodes are modular pieces of software that perform specific functionality. They communicate with each other by passing messages through common topics. In some cases, more complex communication is required. Services are used in a request-response model, where a node makes a request and the receiving node issues a response.

2.2.1 Publisher Subscriber Design Pattern

At the core of ROS is the publisher-subscriber design pattern. In order to break down a software system into nodes, there must be an asynchronous method of communication between nodes to share data. In the case of ROS, nodes can publish data to a topic through the ROS Master, which manages the naming and registration of other nodes and services, as well as tracking publishers and subscribers to topics [21].

Nodes that have the latest data will publish to their respective topics at regular intervals, while nodes that require data will subscribe to these topics and receive new data as soon as they become available. This results in a low level of coupling between software nodes [22], allowing software to be written in a more modular and reusable fashion.

2.2.2 ROS Ecosystem

There are currently two versions of ROS, ROS 1 and ROS 2. ROS 2 was released 8 years after ROS 1, picking up on lessons learned from ROS 1 while creating a more industry-friendly software package that incorporates safety, certification, security and real-time features [23]. This project utilises the last ROS 1 release, ROS Noetic, for simplicity of development. Existing work to implement ROS on the Spot robot has been done in ROS 1. Due to the limited scope of this project, the decision was made to maintain the ROS 1 dependency and focus on adding new functionality to the project that would make it more user friendly.

Since its inception, ROS has gained widespread popularity, with its packages being downloaded millions of times [24]. Many open-source packages are available online, providing functionality such as bridging between ROS message formats and OpenCV, Simultaneous Localisation and Mapping (SLAM), robot sensor data visualisation and even robot simulation. By using ROS middleware, this project aims to make integrating the Spot robot with existing functionality more accessible to users.

2.3 ROS for Spot

2.3.1 Existing Work

In the past, there have been two main projects working to get ROS running on the Spot robot, by Microsoft and Clearpath Robotics. In 2020, Microsoft released a ROS wrapper for Spot [25]. This software package allowed users to access data from Spot's sensors and cameras and issue basic movement and trajectory commands. Since its release, there has been no further development from Microsoft. In the same year, Clearpath Robotics collaborated with Boston Dynamics to integrate ROS with the Spot SDK [26]. In 2022, Clearpath Robotics stopped maintenance of the package. Since then, Michal Staniaszek from the Oxford Robotics Institute has taken over maintenance of the project. This package is now the main method of deploying ROS on Spot, even being used by researchers at Microsoft.

The current state of the package includes functionality for basic movement and trajectory commands, arm trajectory commands, velocity and movement safety limits, autonomous docking and self-righting. However, the original software design from Clearpath Robotics persists, using a monolithic software architecture that makes it difficult for collaboration. The source code is organised into two large files, `spot_ros.py` and `spot_wrapper.py`. The former contains ROS callback functions which interface to the Spot SDK functions in the latter. Functionality across the Core, Robot and Autonomy levels of the stack are coupled and make it difficult for open-source collaboration, as multiple editing the same file would cause merge conflicts and slow the development process.

2.3.2 Docker

Docker is a software platform that allows users to package their software with their own dependencies, containerising applications such that they do not interfere with other containers on the same computer [27]. Unlike full virtualization, Docker shares the operating system kernel with the host computer, making it more lightweight that a full virtual machine. Additionally, applications can be tested locally on a laptop first, before being deployed to the computer payload on the Spot robot.

Deploying custom applications on Spot is best done with Docker to avoid environment conflicts between applications [28]. On the Spot Core computer, Docker containers are managed using Portainer. This allows users to set up the the container remotely, ensuring that the correct network ports are forwarded such that the application can interaction with the Spot SDK on the robot. Once that has been setup, the robot will be able to run autonomously without manual input from an external computer.

2.4 Open Source Software

In order to extend the longevity and increase the usability of the project, it is important to consider making the project open-source friendly. Code modularity, community, project management and testing are key factors in ensuring the success of open-source software projects [29].

2.4.1 Code Modularity

High code modularity allows contributors to work on different parts of the program [30], without requiring a deep understanding of the entire software system; only an understanding of the interface to the core software is required. Successful open-source projects are written in a modular format and provide opportunities for innovation without adversely impacting the whole project [31]. This makes it easy to integrate new features as separate modules without introducing bugs to existing code. Many large companies have found success in writing modular code for their open source projects, such as Meta with React.js [32] and Google with Chromium [33], both of which are widely used in industry.

2.4.2 Unit Testing

Testing is crucial for quality control in any project. Open-source projects typically employ both automated testing and user system testing to catch bugs, before and after release respectively. Unit testing is one way to catch bugs before new features are released.

Unit tests are used to enforce a contract between code and expected behaviour, whereby the code in isolation is given a predetermined input and expected to reach a certain state at the end of its runtime. Tests can be applied at different levels, as described below.

Library unit tests enforce the behaviour of specific functions or classes. They test the behvaiour of an isolated piece of code with a set of predetermined test cases that cover core and edges cases, asserting that the output is as expected. In the context of robotics software, this typically includes control algorithms or driver-related helper functions.

ROS unit tests enforce the correctness of the program's response to ROS interactions such as publishers, subscribers, services and actions. This tests the ROS functionality of a single ROS node, using `rostest` as a test fixture to automatically set up a ROS master.

ROS integration tests test the functionality of several nodes working together, either in isolation from the system or as part of wider system integration testing. In the context of robotics software, integration tests involves testing the communication between different nodes, verifying that sensor data is being properly processed, or testing the behavior of a robot in a simulated environment.

It is important to ensure that edge cases are being tested so that the system does not exhibit unexpected behaviour when functioning within the defined scope. There are further levels of testing up the application stack, such as system integration testing to ensure that the entire system works together without error, and application testing to ensure that the system works as expected in a deployment environment.

2.4.3 Continuous Integration (CI)

In order to reduce repetitive work, automated tests can be set up using a continuous integration pipeline in the cloud. For example, Github offers the *Github Actions* service, allowing maintainers to set up customized actions to install and test new versions of code whenever they become available. This provides immediate feedback to contributors about their code correctness without having to wait for feedback from the maintainers, which can be slow when they reside in different timezones. Additionally, changes to the main branch of the project can only be made with a passing CI build, preventing contributors from unknowingly introducing bugs that later surface as mysterious failures [34].

2.5 Robotics Navigation

The problem of robotics navigation can be broken down into four main categories [35]:

1. Mapping-based navigation
2. Behavior-based navigation
3. Learning-based navigation
4. Communication-based navigation

Mapping-based navigation involves mapping, localization, and path planning. Behavior-based navigation involves obstacle avoidance, wall following, corridor following, and target seeking. Learning-based navigation focuses on creating a semantic understanding of the environment and the task to be performed. Communication-based navigation is most relevant to swarm robots that operate as a decentralized cluster of robots working together to achieve a common goal. This report focuses on mapping-based navigation, evaluating the ability of the Spot robot to map and localize itself in the physical world.

There are a few popular robotics mapping strategies:

1. Feature-based mapping: This strategy identifies salient features in the environment, such as corners or edges, and uses them to create a map [36].
2. Occupancy grid mapping: This strategy creates a grid-based map that not only represents whether a cell is occupied or unoccupied but also represents the probability that a cell is occupied [37].
3. Simultaneous Localization and Mapping (SLAM): This strategy involves creating a map of the environment while simultaneously localizing the robot within that environment [38]. A popular class of SLAM algorithm is Graph-SLAM [39], where the robot's trajectory and poses of world objects are captured as nodes in a graph, and edges represent the spatial constraint between nodes.

2.5.1 Simultaneous Localization and Mapping (SLAM)

SLAM is designed for operations in uncertain environments, where the map is unknown or dynamic, requiring updates during operation [38]. This report focuses on SLAM navigation as it is most similar to what is implemented on the Spot robot.

Spot utilises a variant of Graph-SLAM for its GraphNav [40] mapping and localization feature. Mapping occurs when the robot is manually operated and walked around its environment, placing nodes along its path. Spot uses its sensor and odometry data to generate a point cloud representation of its environment, and saves the walked path as a graph. It then uses environmental features saved in its map for localization as it traverses the graph. It is important to note that Spot executes mapping first during the recorded run, then localization for subsequent runs. The map is not dynamically updated during operation.

3 ROS Platform Development

This section discusses the design behind the ROS package that interfaces with the Spot SDK, highlighting key improvements made over previous work.

3.1 Software Architecture

The overall software architecture of this ROS software package is shown in Figure 3.1. It serves as a middleware interface between the vendor-specific Spot SDK and the popular ROS framework. The physical robot hosts all functionality, split between the Spot SDK running on the proprietary computer in the robot, and the user-specified programs running on the Spot Core computer. The Spot SDK is used to communicate with vendor-specific payloads such as the Spot Arm. The `SpotROS` wrapper is used to interface between the Spot SDK and the ROS Master node, relaying data and commands between the two. Additional functionality can be added through ROS, either as internal applications on the Spot Core computer, or on a laptop that is connected to the WiFi access point hosted by the robot. This allows for additional applications to be deployed on Docker and interfaced with the robot through ROS.

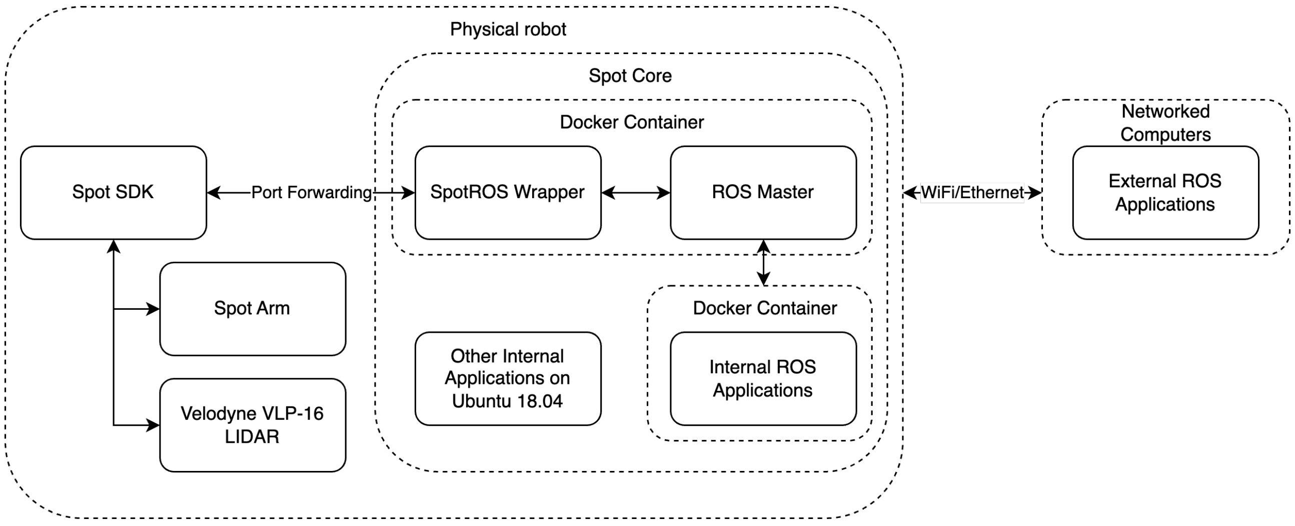


Figure 3.1: Software architecture of ROS on Spot

3.1.1 Previous Work

The previously designed ROS interface started by Clearpath Robotics [41] uses a monolithic software design. All functionality is collected in two super-classes (`SpotROS` and `SpotWrapper`), representing the Spot SDK interface and ROS interface respectively. This makes deployment of the software build system easy, as all required functionality is stored in the same file where it is being used. This type of design is advantageous in the early stages of development, where minimal planning and effort is needed to organise the code, reducing initial start-up effort.

The class Unified Modelling Language (UML) diagram can be seen in Figure 3.2. All functions are stored in two classes, and the level of code coupling between functions within each class is not clear. The control flow is not clear with this many functions, making testing hard as a bug in one function may be caused by an error elsewhere. In the long term as more features are added to the software package, the classes will only grow more complicated and become harder to test and maintain in the future.

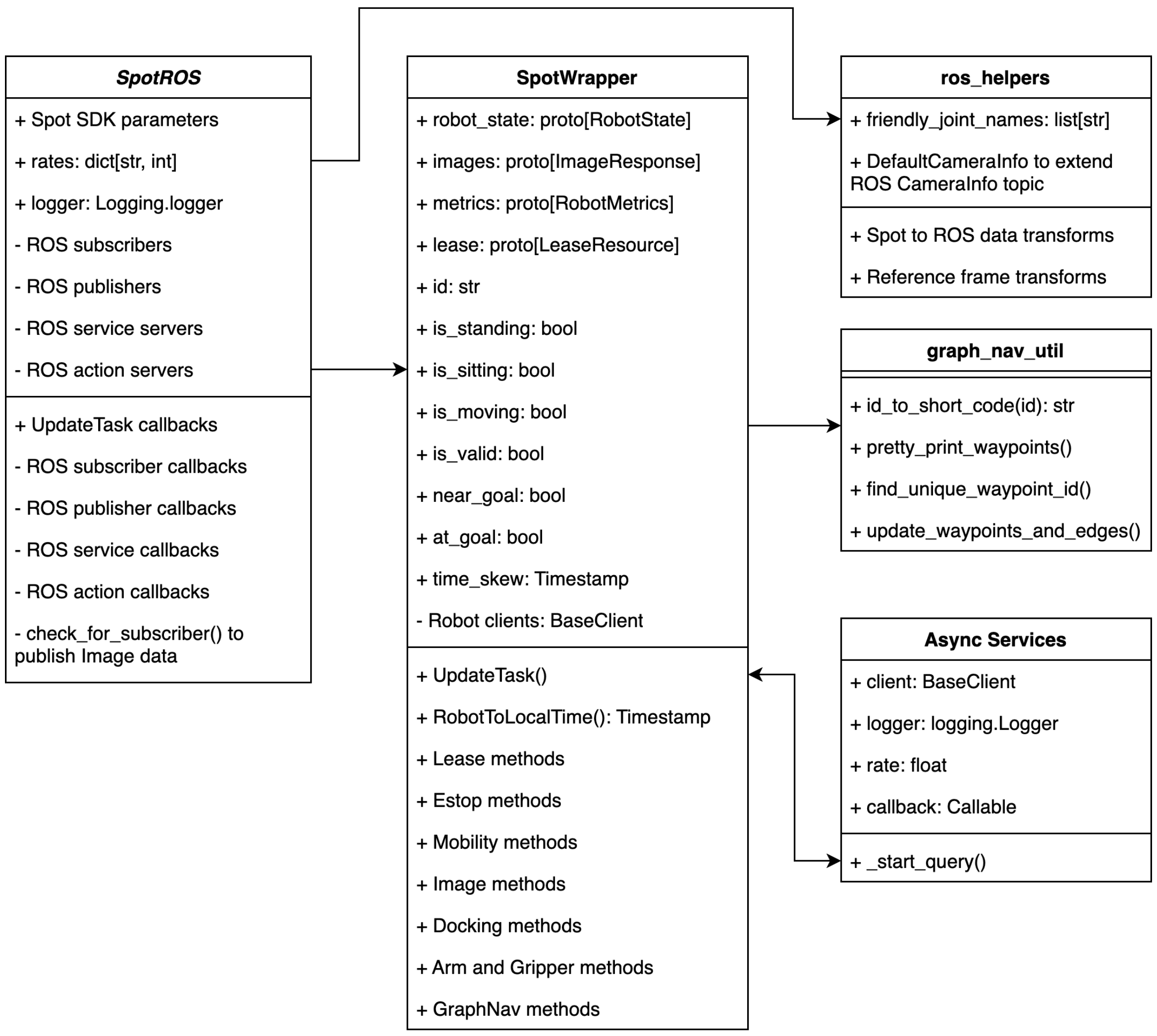


Figure 3.2: Class diagram of previous work of SpotROS

Furthermore, a monolithic design is difficult for open-source collaboration, where many contributors can be working on the same part of the project at the same time. In that case, there would be Git merge conflicts, as two contributors might be concurrently requesting different changes to the same file. This would need to be resolved manually, increasing development time and effort as well as making software testing more difficult.

3.1.2 Improvements

The updated design refactors the Spot SDK components in `SpotWrapper`. Core functionality of the robot including mobility methods such as powering on the motors, sitting, standing, and taking images from its onboard cameras are kept in the core `SpotWrapper` class. Other features with low coupling to the aforementioned core functionality are split into separate classes, that are instantiated in the main class. This makes it easier to add additional Spot SDK functionality to the `SpotWrapper` class with minimal interference to existing functionality. New features use standardised constructor arguments `robot, logger, robot_params, robot_clients` to interface with the Spot SDK clients in the `SpotWrapper` class, while providing their own set of methods to actuate the robot. This represents a separation of concerns between classes, where the `SpotROS` class handles ROS communication, the `SpotWrapper` class handles Spot SDK communication and each feature class handles its own functionality.

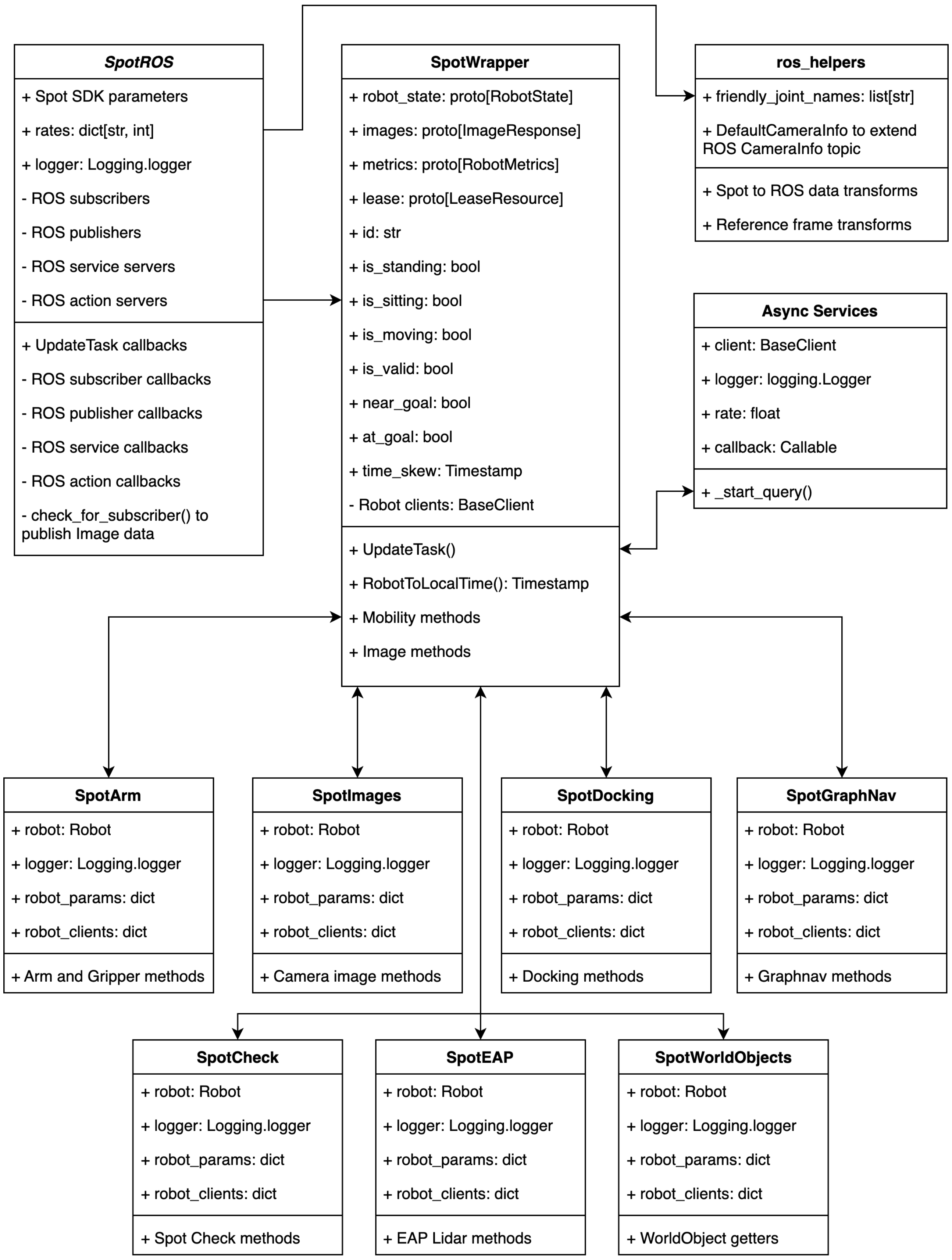


Figure 3.3: Class diagram of updated work of SpotROS

3.1.3 Detailed Software Design

Delving deeper into the details of the software package, the features can be classified into four different types of implementation.

ROS publishers are used to periodically publish data from the Spot SDK to ROS, without requiring an external trigger. Robot state and sensor data is polled at a constant rate using asynchronous RPC calls to the robot, then published to ROS. This is executed using `AsyncPeriodicQuery` tasks, where the main loop checks whether sufficient time has passed since the last call and triggers the callback to maintain a constant poll rate. The loop rate is specified to be much faster than the individual task rates, such that the main loop checks each task multiple times a cycle, maintaining a constant rate of polling. The main loop rate and task rates are configured with a `spot_ros.yaml` file as specified in the ROS `launch` file used to start the Spot ROS driver. For Spot, this involves sensor data from its cameras, LIDAR, robot state service, robot metrics service, lease status service and world object service.

ROS subscribers are used for simple features that do not require feedback when its callback is triggered. When new data is published to a ROS topic, the subscriber will pass that data to a callback function to execute a command. For Spot, this involves the `/spot/cmd_vel` topic to set the movement speed of the robot, and the `/spot/go_to_pose` topic to command a trajectory move to a point in the robot's `odom` or `vision` reference frame.

ROS services are used for simple features that execute in a short period of time and require passing feedback or results to the client calling the service. When a ROS service is called, the client execution is blocked until the a result status is sent from the server to the client, ensuring that the service has reached an end state before allowing the client to call additional services. For Spot, this involves services such as arm state changes with `/spot/arm_stow` and safety state changes with `/spot/power_on`.

Unlike services, ROS actions do not block the client and are asynchronous in nature. The client is able to pass a goal to an action and continue program execution, then retrieve the result when it is ready. ROS actions are typically used for longer tasks that involve movement of the robot, such as `/spot/navigate_to` and `/spot/trajectory` that allow it to move to specific locations in a GraphNav map or `odom` and `vision` reference frames respectively.

The full details of each ROS topic and service implemented in this ROS package are available in Appendix A. Together, they form a complete guide on what Spot SDK functionality is available through this ROS wrapper.

3.1.4 Application Sequence Diagrams

The sequence diagram for an example application is shown in Figure 3.4. The SpotROS node is started with a ROS launch file, entering the program by instantiating the `SpotROS` class and running its `main()` function. The `SpotWrapper` object is then instantiated, creating objects that interact with the Spot SDK and authenticating with the robot. Next, the ROS publishers, subscribers, action servers and services are created. The velocity limit is set as a parameter in `SpotWrapper`, limiting the speed of movement actions that may be triggered during operation. If the ROS package is configured, it will automatically claim the robot's lease, power on the motors and make the robot stand up. This allows further actions to be performed later. In the main loop of the program, it periodically calls `SpotROS` callbacks that retrieve the latest data from the Spot SDK and publish them to the relevant ROS topics.

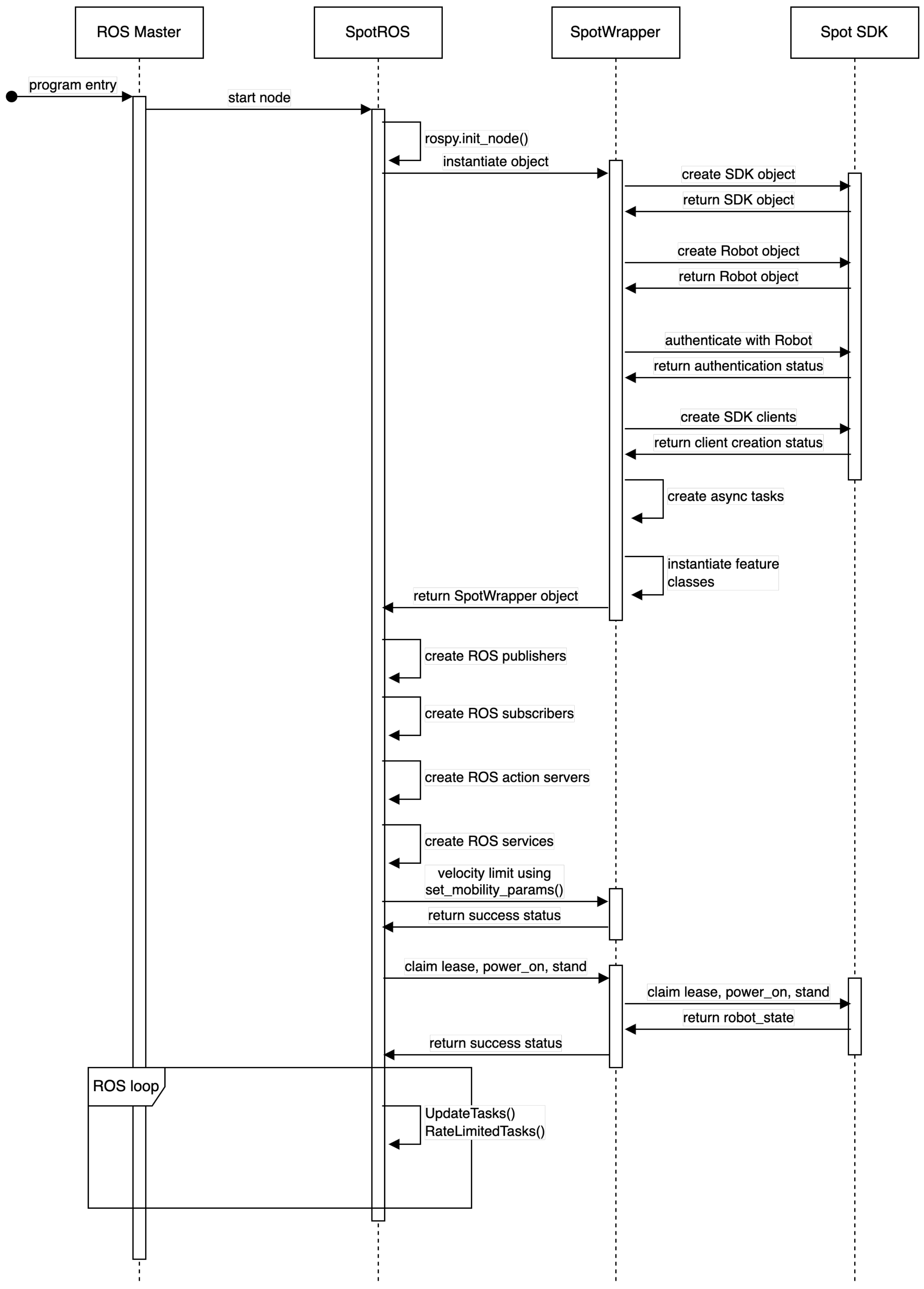


Figure 3.4: Sequence diagram for SpotROS initialisation

ROS subscribers, actions and services operate on an event-based model, triggering only when they get updated information or are called by another ROS node. A sequence diagram of a callback triggered by a ROS service is shown in Figure 3.5. When another ROS node calls the service, it gets the address of the service node from the ROS Master, then sends a request to it. The `SpotROS` object in the node then calls the relevant `SpotWrapper` method to send a request to the Spot SDK. The success status in the response is then relayed up the stack as a response to the original ROS request, closing the loop. The process is similar for ROS subscribers and actions, allowing ROS nodes to function seamlessly with the Spot SDK thanks to the abstraction provided by this ROS package.

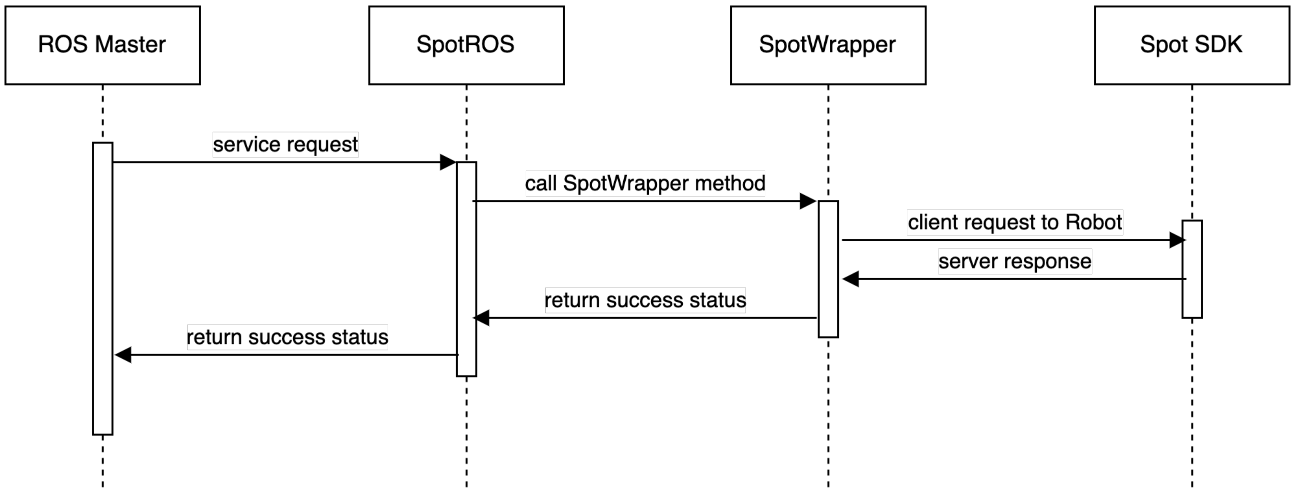


Figure 3.5: Sequence diagram for SpotROS service calls

3.2 Open Source Collaboration

As mentioned in Section 2.4, the key success factors for an open-source software project include code modularity, community, project management and testing. Efforts to improve code modularity have been explained in Section 3.1.2, while the testing aspects are explained below.

3.2.1 Unit Testing

To enforce coding standards and code correctness, several levels of testing have been set up in the ROS package. In this project, the main challenge with testing was isolating components from the Spot SDK. As seen in Figure 3.4, even the initialisation of the ROS package requires Spot SDK functionality, which is unavailable in a development environment. Boston Dynamics does not provide a digital twin or ROS Gazebo simulation of the Spot robot for automated testing. Thus, the Spot SDK interface has to be mocked for each test that interacts with the Spot SDK in order to remove the testing dependency on hardware [42]. The different types of unit tests are explored below.

For library-level unit tests, individual functions are tested for correctness with the corresponding inputs expected during actual operation of the robot. These tests are relatively straightforward, covering most coupling methods that translate data between the Spot SDK and ROS formats. ROS-level unit tests have added complexity as they require a ROS Master node to be started as they primarily test ROS communication and subsequent control flow within the program.

To isolate the ROS package from the Spot SDK during testing, a mock `SpotROS` class was created. Callback functions that would have used methods from the `SpotWrapper` class were stubbed to set and retrieve data within the class itself, allowing tests to test the ROS communication in isolation from Spot SDK communication in the `SpotWrapper` class. In more complex functions, the relevant `SpotWrapper` method was stubbed directly to allow testing of the logic of the `SpotROS` function. Examples of unit testing applied in this project can be found in Appendix B.

Lastly, system integration testing was explored. This involves running the robot with a specified command, and checking its behaviour such that it corresponds to the expected physical behaviour of the robot. However, automating this type of testing is not feasible without an accurate simulator of the Spot robot dynamics. The alternative is to dedicate an entire robot and space solely for automated integration testing, which is extremely expensive. Alternatively, implementing a mock gRPC server would have allowed testing of the client-server interactions with the Spot SDK but would not have served the purpose of ensuring that the system works together as a whole. Thus, automated system integration testing was not implemented in this software package.

3.2.2 Continuous Integration and Continuous Delivery (CI/CD)

For ease of collaboration with other maintainers on cloud-based Git repository services, an automated testing pipeline has been set up on Github. The CI pipeline can be seen in Figure 3.6. Every time a change is pushed to a branch in the cloud, the Github Actions runner will set up a Docker container with the required dependencies, download the updated code, build it with the ROS build manager `catkin_make`, run ROS testing with `rostest`, output the results, then gracefully shutdown the container.

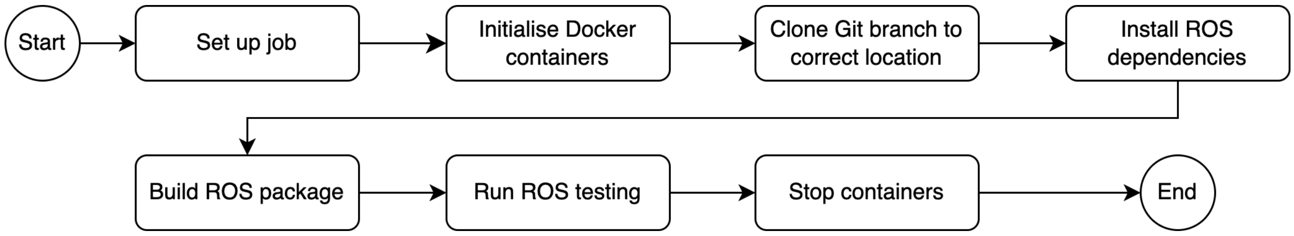


Figure 3.6: Github Actions CI pipeline

In addition, the new code can be prepared for deployment with an automated pipeline. The CD pipeline can be seen in Figure 3.7. With the same Github Actions runner, we will prepare a Docker image. The runner will use a Dockerfile to checkout the latest code branch within the Docker image, install the required dependencies, build the ROS package, and set up the package to auto-start when the Docker container is created with `ENTRYPOINT`. The Docker image is then uploaded to Docker Hub, where it can be downloaded as a compressed `tar` archive file to be uploaded to the Spot Core for deployment later. This streamlines the deployment process, allowing developers to test the latest code on hardware without going through the repetitive process of manual software delivery.

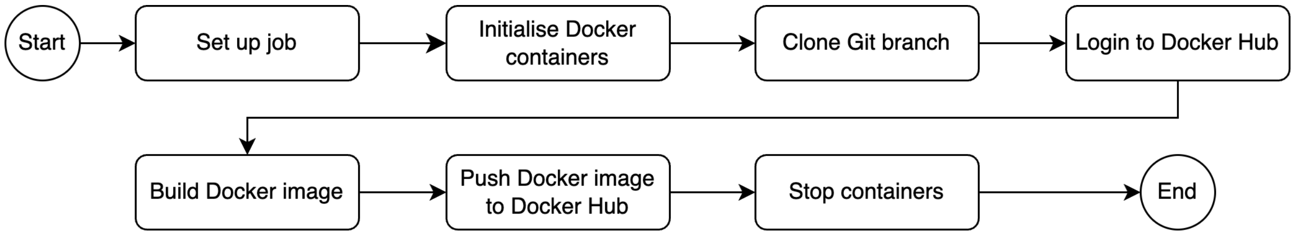


Figure 3.7: Github Actions CD pipeline

4 ROS Platform Testing

4.1 Deployment

Of particular interest to the ROS package is the networking setup that allows communication between the different software modules. For the purposes of testing, the ROS package is executed in a Docker container on a laptop connected to the Spot robot via its WiFi access point. This simplified networking as all ROS code is run on the same host, allowing it to automatically find the ROS Master node without setting up a static IP address for it. The Docker container packages the software environment, making it easier to transfer the ROS package and its dependencies to the robot later. Deployment on the Spot Core on board the robot will be explored in Section 6.

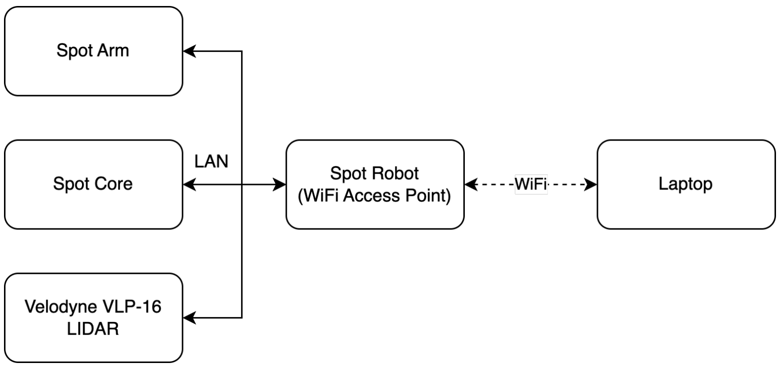


Figure 4.1: Hardware deployment of Spot with payloads

4.2 Integration Testing

To validate the correctness and stability of the software package created in Section 3, integration testing was performed with the actual Spot robot. Each time a new feature was added, the software was deployed as per Section 4.1. Then, the feature was called through a `rosservice` or `rostopic` call, corresponding to the type of ROS communication used by the feature. The general testing procedure is shown in Figure 4.2. All testing was conducted with a human operator nearby to ensure that a safety radius was maintained around the robot at all times. The results were validated against manufacturer-provided examples to ensure they maintained the original functionality.

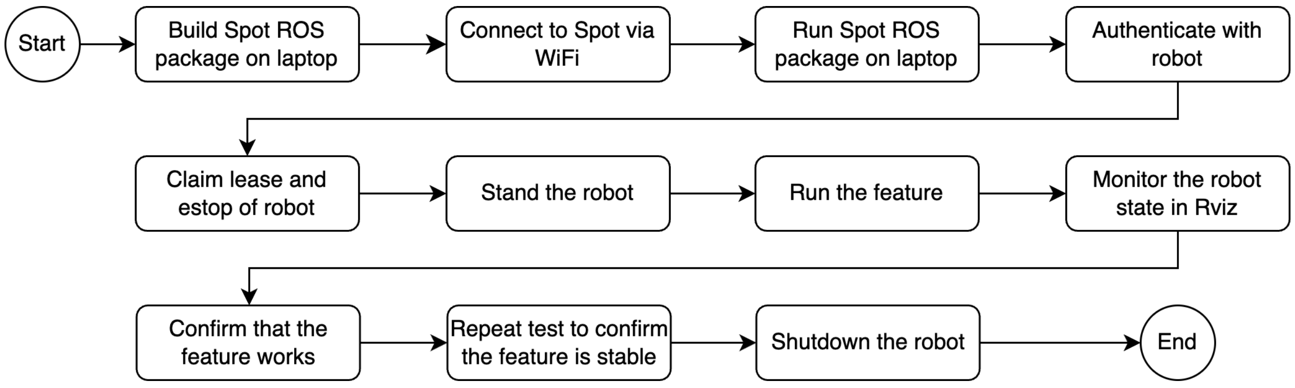

Start
Build Spot ROS package on laptop
Connect to Spot via WiFi
Run Spot ROS package on laptop
Authenticate with robot
Claim lease and estop of robot
Stand the robot
Run the feature
Monitor the robot state in Rviz
Confirm that the feature works
Repeat test to confirm the feature is stable
Shutdown the robot
End

Figure 4.2: Integration testing procedure on hardware

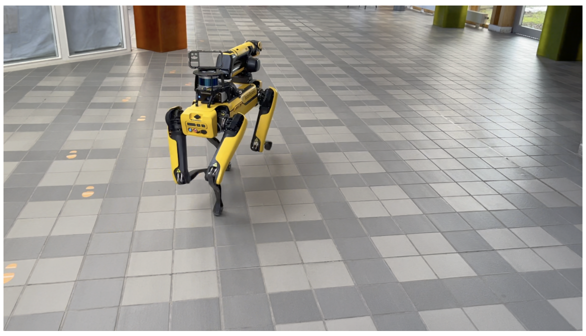
In the following sections, detailed testing results are showcased for each feature that was written for the ROS package.

4.3 Showcase: Velodyne LIDAR Data

The LIDAR data from the Spot EAP payload can be accessed in two different ways: through the Spot SDK or through the Ethernet interface of the LIDAR. Accessing the LIDAR directly would interfere with the existing interaction between the LIDAR and the Spot SDK in the `velodyne_service` function [8]. Thus, the decision was made to access the LIDAR data through the Spot SDK's `PointCloudClient`, providing filtered LIDAR data used by Spot's native odometry system.

During testing the robot was driven around with Autowalk, Spot's trajectory playback system, to walk in a recorded path around an indoor hall. This was done to ensure that the LIDAR data polling by the ROS package did not interfere with the autonomy capabilities of Spot in Autowalk, allowing it to maintain use of the LIDAR with the ROS package.

In Figure 4.3, the depth camera data on the Spot and the LIDAR data from the ROS package are shown in RViz, the ROS data visualisation tool. Depth camera data is coloured by proximity, where warmer colours indicate a lower height in the robot's body frame. LIDAR data is shown in white. The long range capabilities of the LIDAR is immediately apparent, having a significantly longer range than the depth cameras on the Spot robot. Features that are further away were more accurately mapped. For example, walls were mapped with points in a flat plane, whereas the depth camera data showed significant noise distorting the point cloud representation of the wall.


(a) Spot walking with EAP package

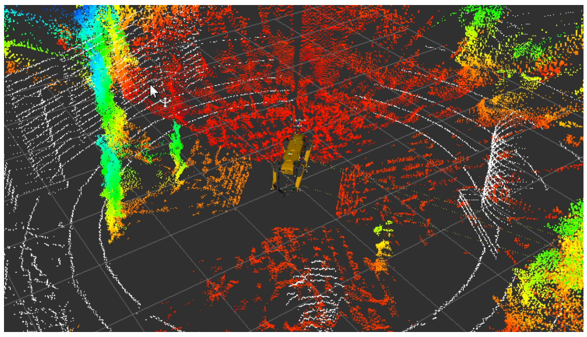

(b) Point cloud data seen in RViz

Figure 4.3: Spot robot with EAP package using LIDAR

Comparing the output data with that observed in existing literature utilising the same LIDAR [11], a similar level of fidelity was observed, indicating the correctness of the data output from hardware to the ROS package. Thus, it was remarked the the feature was implemented successfully.

4.4 Showcase: Graph Navigation

The Spot robot has a native localisation and mapping package, named GraphNav. This allows an operator to manually walk the robot around to record features in its environment, building a map and placing waypoints along its path. It then connects those waypoints with edges, performs anchor optimisation and loop closure to allow optimal navigation between waypoints in the future. Once the map has been saved, Spot can use it to localise itself with respect to environmental features, such as fiducial markers present along its route.

The GraphNav service was implemented on the Spot robot as a way to playback existing manually recorded maps and direct it to a desired waypoint in the map. A map recorded using the controller is shown in Figure 4.4. As the robot walks around the environment, it fuses data from its LIDAR and depth cameras to generate a feature map of its environment for localization. At regular intervals along its path, it places waypoints that form a navigation graph. When playing back the GraphNav graph, the robot can be told to traverse between any two waypoints, using the edges formed when recording the graph. The ROS interface allows use of the GraphNav playback functionality through ROS services and action servers, without blocking other functionality such as arm movement during the walk.

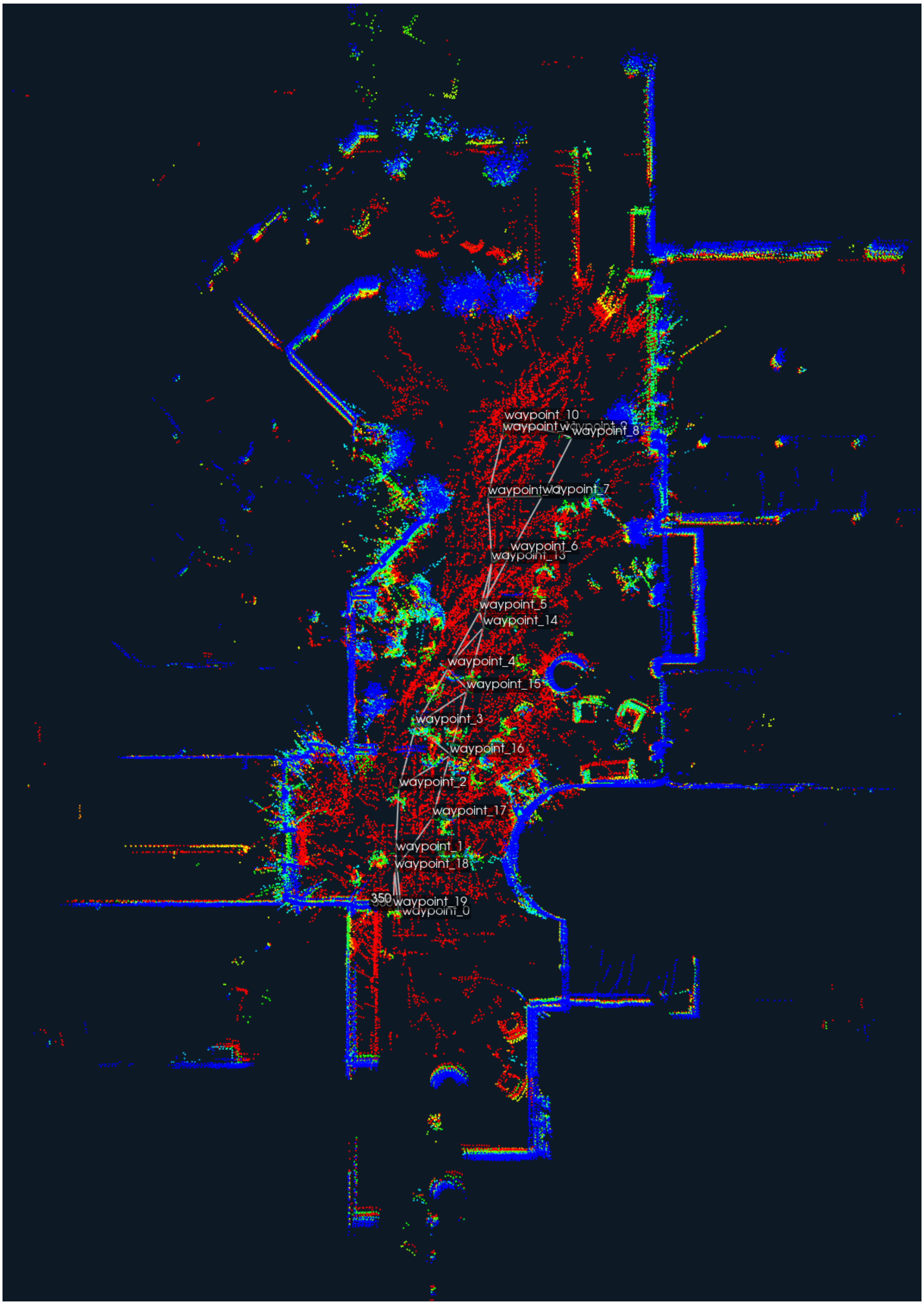
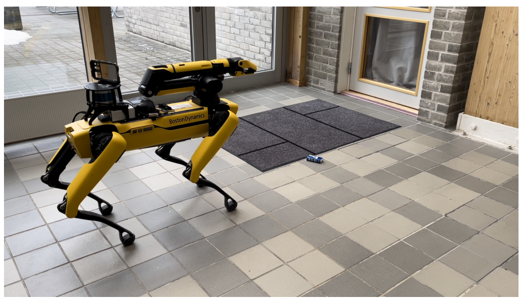


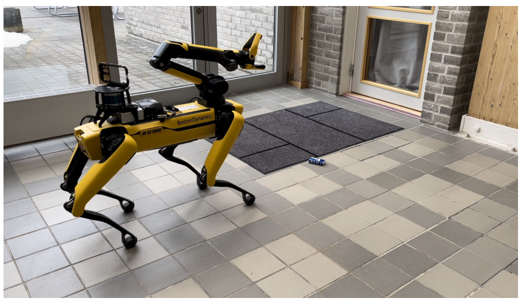
Figure 4.4: Map generated in GraphNav of Alfa 3, Ideon Science Park

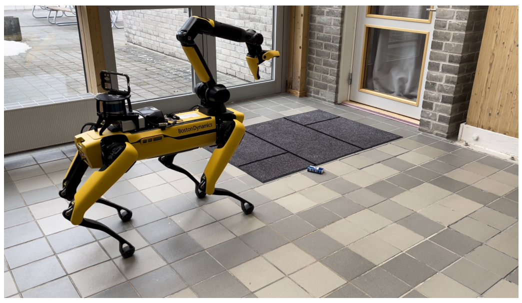
4.5 Showcase: Arm Picking

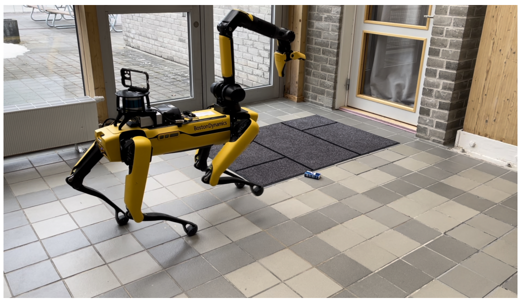
The Spot arm is able to use its gripper to pick up objects and carry them around. There are two methods to doing so: by specifying the reference frame of the target in the robot's body frame, or by manually selecting a point in a picture from one of its on-board cameras. The first method was implemented in the ROS package as the simpler method was deemed to be easier to adapt for use. Higher-level application code could identify the position of the robot separately, before calling this feature to initiate the actual picking of the target.

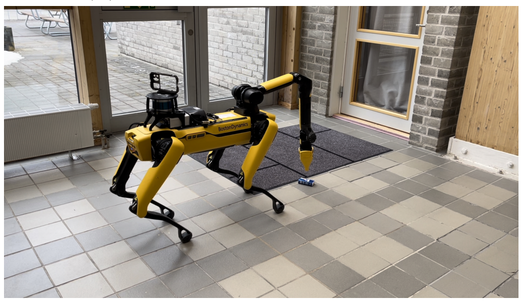
During testing, a packet of pocket tissues was used as the target object to be picked. A ROS service call was made with the target's reference frame (x, y, z) in the robot's body frame to initiate the picking process. The robot opens the gripper, moves the arm into the Carry position, moves towards the target, attempts to pick the object, then stays in the picked position. The ROS service then returns with the completion status of the pick. Subsequently, further arm actions can be issued to move the arm with the object.


(a) Starting from standing position


(b) Opening the gripper


(c) Arm in the Carry position


(d) Moving towards the target


(e) Picking the object

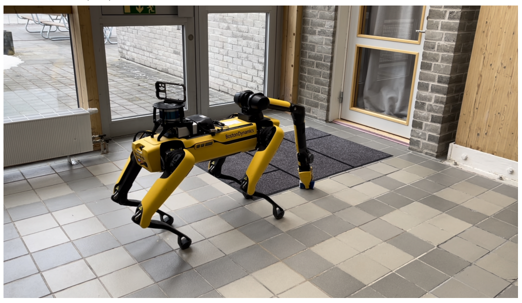
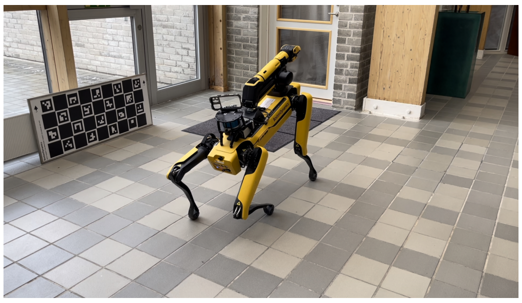

(f) Successful pick

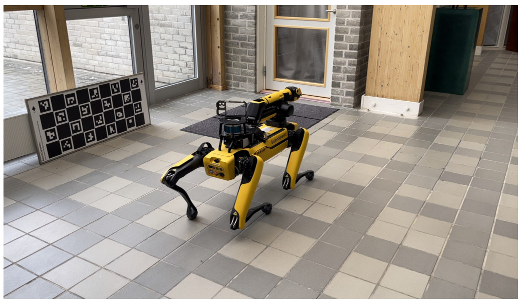
Figure 4.5: Spot robot picking an object

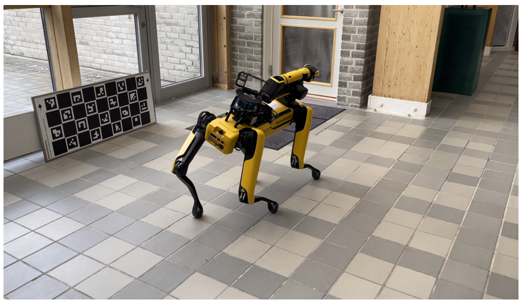
Throughout the process, it is important to note that the picking service requires the ownership lease of the robot to be claimed, meaning that the robot cannot perform other movement actions throughout the process. This blocks other actions such as trajectory movement, thus the action has to be cancelled or run to completion for the robot to execute other tasks. However, non-movement actions such as requesting camera data can still be performed asynchronously.

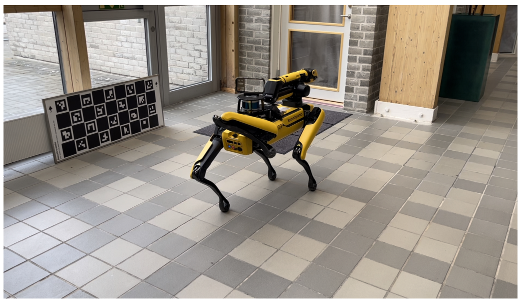
4.6 Showcase: Spot Check Diagnostics

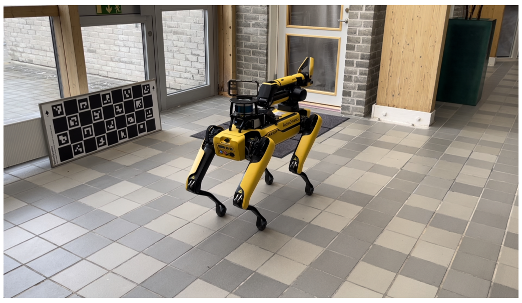
The Spot robot has a native calibration and diagnostics routine that should be regularly run to ensure that the joint servo motors are running within set parameters. This provides camera, load cell, kinematic joint, payload and hip range of motion calibration offsets and error states. Boston Dynamics recommends that this calibration be performed once every 30 days [43]. The ability to run this diagnostic through ROS automatically would be useful for an operator running a fleet of robots concurrently.

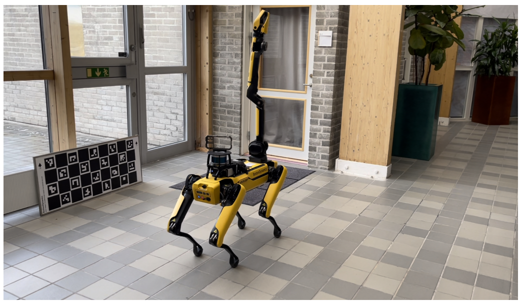

(a) Testing body pitch

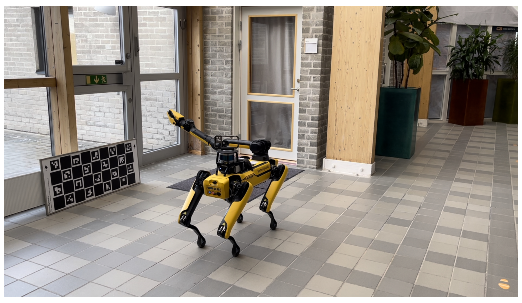

(b) Testing body height


(c) Testing body roll


(d) Testing hip range of motion


(e) Testing gripper open and close


(f) Testing arm base


(g) Testing arm elbow

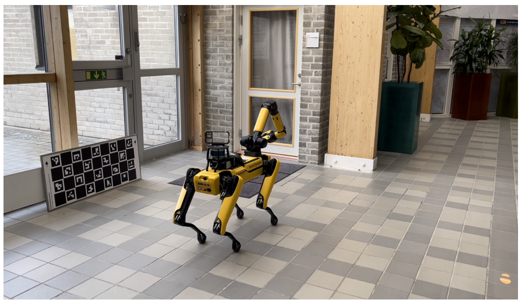

(h) Testing arm wrist

Figure 4.6: Spot check procedure

5 Platform Application Development

The application aims to evaluate the navigation capabilities of the Spot robot. The two benchmarks explored are path playback accuracy and mapping accuracy. The former is measured with AprilTag fiducials to triangulate the ground truth position of the robot, while the latter is compared against the map produced by an open-source LIDAR and Visual SLAM package, RTAB-Map [44]. Additionally, the application demonstrates the ease of integration of existing ROS packages with the SpotROS wrapper, showing how accessible the Spot robot platform is to existing ROS engineers and researchers.

5.1 Software Architecture

The software architecture of the application is shown in Figure 5.1. It builds on the Spot ROS wrapper architecture in Figure 3.1, adding a ROS node to test localization performance with fiducial tracking data and an RTAB-Map ROS node for mapping. For debugging purposes, the ROS master is made available via port forwarding, allowing an external laptop to connect to it and view ROS data through RViz. All processing is done on the Spot Core carried on the robot, to allow for fully autonomous functionality without an operator with a laptop nearby. The Docker container is built as part of the continuous delivery pipeline described in Section 3.2.2, and deployed to the Spot Core. This allows all dependencies to be shipped with the application software, thus the Spot robot does not need internet access during operation to download or update dependencies.

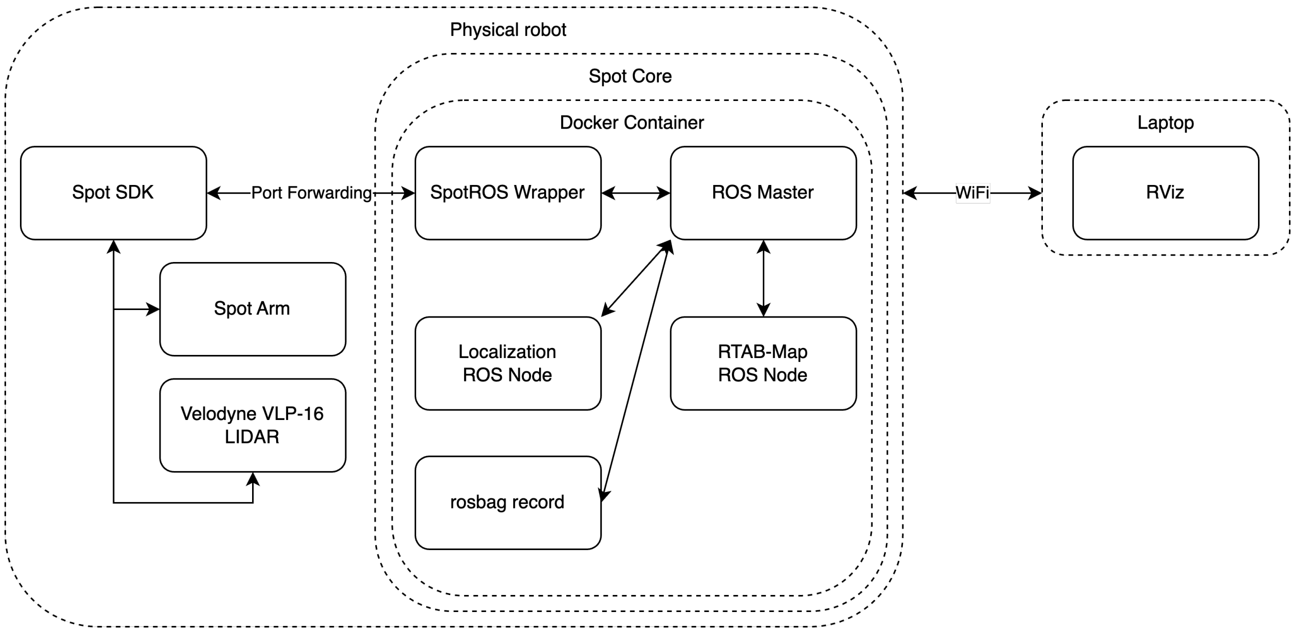


Figure 5.1: System architecture of the navigation application

5.2 Hardware Architecture

Additionally, edge processing of SLAM and recording all data produced by the SpotROS wrapper was found to be pushing the limit of the Spot Core computer. Recording all the data produced by the SpotROS node was estimated to produce 34GB of data over a 10 minute period, giving a rough data rate of 56 MB/s. However, the Spot Explorer robot used in this report has limited WiFi bandwidth, supporting only 2.4Ghz 802.11 b/g/n speeds [45]. This prevents real-time streaming of all data produced by Spot from the SpotROS package over WiFi to a laptop, hence data logging was performed directly in the Spot Core, which has a high-throughput Gigabit Ethernet connection to the robot.

5.3 Fiducial Localisation

The control flow of the localization ROS node used for the evaluation of the GraphNav localization capabilities of the Spot robot is shown in Figure 5.2. Fiducial locations are detected with the `WorldObjectService` present on the Spot robot, giving the pose of the fiducial in the `body` reference frame of the robot. This data is periodically published through `SpotROS` to the `/spot/world_objects` ROS topic at a rate of 5Hz, as defined in the SpotROS node.

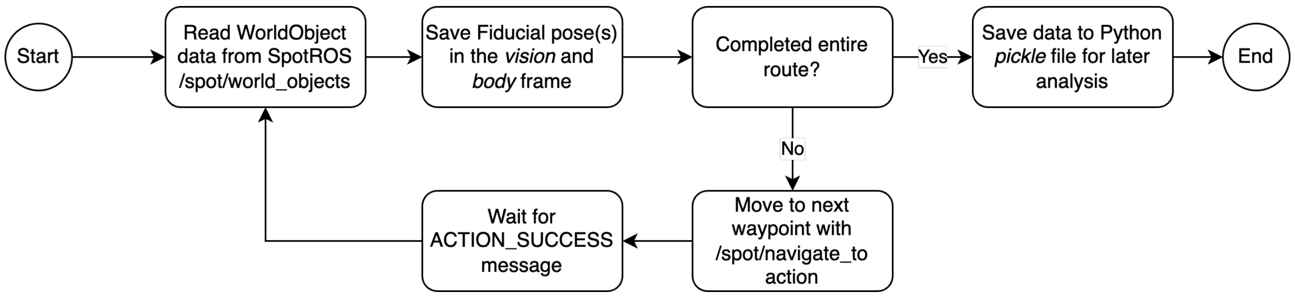

Start
Read WorldObject data from SpotROS /spot/world_objects
Save Fiducial pose(s) in the *vision* and *body* frame
Completed entire route?
Yes
Save data to Python *pickle* file for later analysis
End
No
Move to next waypoint with /spot/navigate_to action
Wait for ACTION_SUCCESS message

Figure 5.2: Fiducial localization ROS node control flow

During the test, the fiducial localization ROS node calls the `/spot/navigate_to` ROS action to make the robot traverse the GraphNav graph, from waypoint to waypoint, as seen in Figure 5.3. At each waypoint, the logging ROS node retrieves the WorldObject data with a ROS subscriber, parses it for fiducial detections, then saves it to a Python `pickle` file for offline data analysis. The robot is then made to walk the graph three times, giving a slightly different estimate of the fiducial pose at each visit. The variation in the pose of each fiducial at each waypoint is used to evaluate the revisit accuracy of Spot's GraphNav navigation service. The code used for the fiducial localization node is provided in Appendix C.

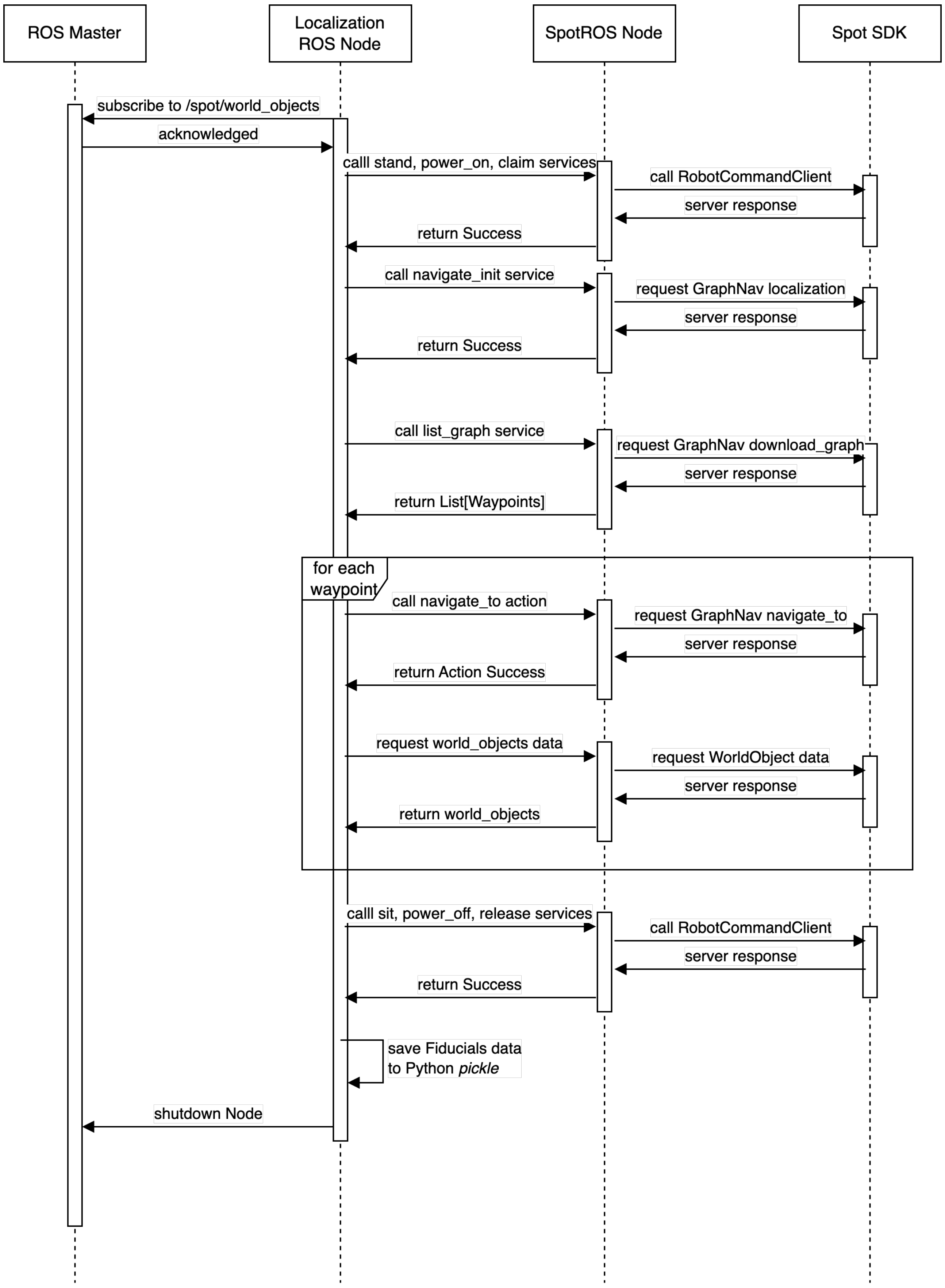


Figure 5.3: Fiducial localization ROS node sequence diagram

5.4 Sensor Fusion Pipeline for SLAM

RTAB-Map utilises odometry data, depth camera data and LIDAR point cloud data to perform environment mapping. The sensor data pipeline is defined in the `launch` file used to start the RTAB-Map node, allowing the user to define any number of camera and LIDAR data sources with their reference frame transformations relative to the body. The launch file used in this report is available in Appendix D.1. The sensor fusion pipeline is shown in Figure 5.4. For this report, all five stereo cameras on the Spot robot (frontleft, frontright, left, right, back) and the Spot EAP LIDAR were used. It is important to build the RTAB-Map ROS package from source with the `-DRTABMAP_SYNC_MULTI_RGBD=ON` CMake flag to enable multi-camera support.

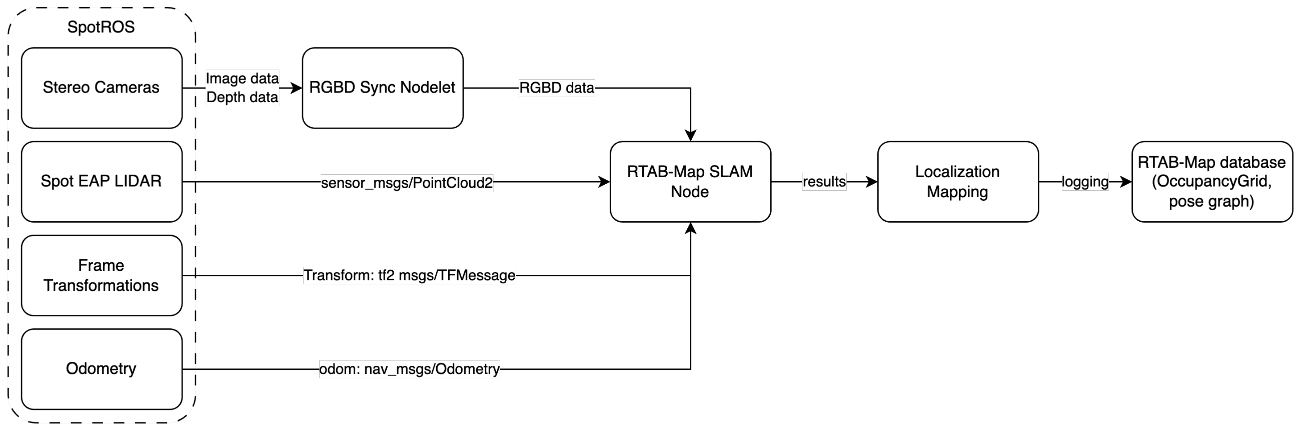


Figure 5.4: RTAB-Map SLAM sensor fusion pipeline

The static transforms between the sensor frames and the `body` reference frame were obtained from the `/tf_static` ROS topic in the SpotROS driver. The transforms for the Spot robot in this project are available in Appendix D.2. All sensor data was defined in the `body` reference frame to be consistent throughout the ROS package. The LIDAR data was provided in the `odom` reference frame by SpotROS and had to be transformed into the `body` frame using a transform from the `/tf` ROS topic. This transformation is executed automatically in the RTAB-Map node with the TF2 ROS package. For a fair comparison between the maps generated by GraphNav and RTAB-Map, only LIDAR and depth camera data was used for mapping, thus neither package had access to fiducial positions for localization and loop closure.

5.5 Path Planning and Navigation

The RTAB-Map package outputs its results into the following topics, as seen in Table D.1. The main output used in this report is the `cloud_map` for the point cloud reconstruction of the environment and the `grid_map` for the occupancy grid generated by walking around the environment. As compared to the graph output by Spot's native GraphNav service, the nodes and edges from RTAB-Map will be placed differently when walked on the same route, thus the graphs themselves will not be used for comparison. Instead, the point cloud map will be compared for the presence of artifacts such as distortion and noise, as seen in Section 6. The purpose of this test is to evaluate the reliability of each navigation stack for a real-world application; finding and navigating to a button in the map.

Using the map generated by the RTAB-Map package, the ROS Navigation `move_base` package [46] was used to complete the open-source navigation stack deployed on the Spot robot. The path planning ROS node takes in frame transformations on the robot, camera and LIDAR sensor data, odometry data and the occupancy grid map of the environment generated by the RTAB-Map SLAM ROS node.

Within the `move_base` navigation node, it uses costmaps and planners to plan a path from source to destination, in the context of the `OccupancyGrid` map provided to it, as seen in Figure 5.5. Costmaps represent the domain as cells with a cost value, depending on the current position of the robot and their occupancy status. The planner uses this information to plot a path, with the local planner dealing with short-term trajectory while the global planner focuses on the high-level objective of reaching the destination. If the planner fails to find a path, the default recovery behaviour defined is to clear out to a distance of `4 * ~/local_costmap/circumscribed_radius`. For this report, the planner chosen is the default `navfn` planner in the ROS navigation stack [47].

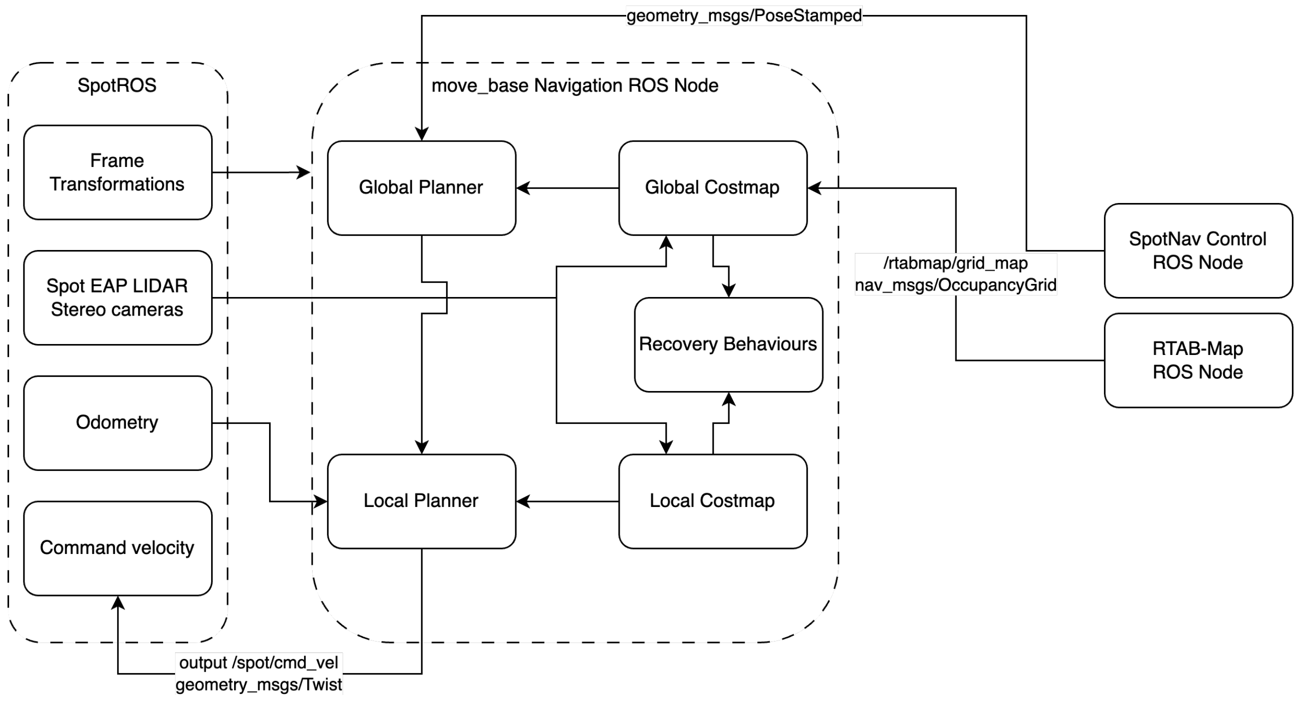


Figure 5.5: move_base path planning pipeline

When deploying RTAB-Map and the ROS navigation package, fiducial detections were fed to the RTAB-Map package for localization. The same was done for the GraphNav implementation, for a fair comparison between their reliability in this evaluation, in Section 6.4.3.

6 Platform Application Testing

6.1 Deployment

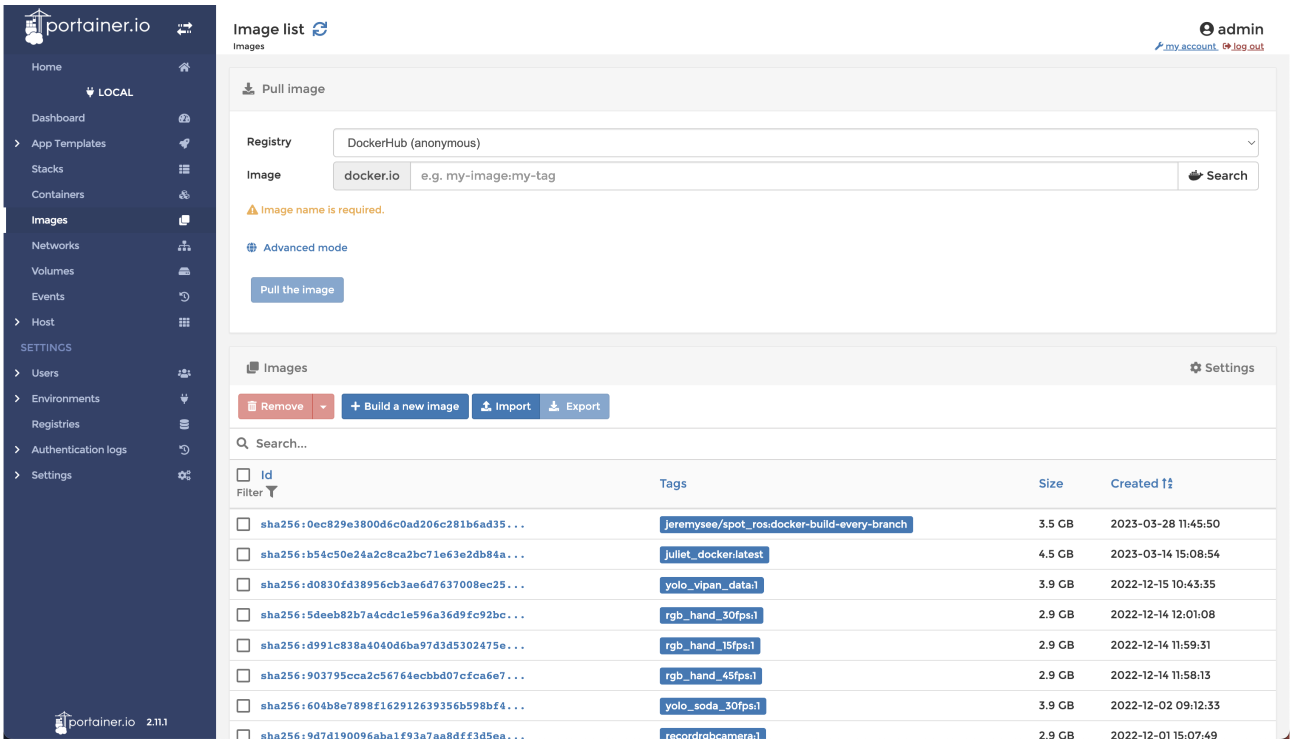
The code was deployed in Docker containers running on the Spot Core on-board computer. The Core hosts a Portainer GUI to allow for easy Docker container management, allowing the user to set up and start Docker containers from uploaded images, as seen in Figure 6.1. The Docker container was configured with the following settings:

1. `--host-ip 192.168.50.5 --guid <USERNAME> --secret <PASSWORD> 192.168.50.3` where `USERNAME` and `PASSWORD` are the Spot robot login credentials

2. Docker "Network" field to `host`

3. Environmental variables for the `SPOT_ARM` and `SPOT_PASSWORD` to set the Spot Arm RViz parameter and Spot robot login credentials for the SpotROS driver launch

4. Interactive mode to allow for terminal access once the container has been started



Registry: DockerHub (anonymous)

Image: docker.io e.g. my-image:my-tag Search

Image name is required.

Id	Tags	Size	Created
sha256:0ec829e3800d6c0ad206c281b6ad35...	jeremysee/spot_ros:docker-build-every-branch	3.5 GB	2023-03-28 11:45:50
sha256:b54c50e24a2c8ca2bc71e63e2db84a...	juliet_docker:latest	4.5 GB	2023-03-14 15:08:54
sha256:d0830fd38956cb3ae6d7637008ec25...	yolo_vipan_data:1	3.9 GB	2022-12-15 10:43:35
sha256:5deeb82b7a4cdc1e596a36d9fc92bc...	rgb_hand_30fps:1	2.9 GB	2022-12-14 12:01:08
sha256:d991c838a4040d6ba97d3d5302475e...	rgb_hand_15fps:1	2.9 GB	2022-12-14 11:59:31
sha256:903795cca2c56764ecbbd07cfca6e7...	rgb_hand_45fps:1	2.9 GB	2022-12-14 11:58:13
sha256:604b8e7898f162912639356b598bf4...	yolo_soda_30fps:1	3.9 GB	2022-12-02 09:12:33
sha256:9d7d190096aba1f93a7aa8dff3d5ea...	recordrgbcamera:1	2.9 GB	2022-12-01 15:07:49

Figure 6.1: Portainer GUI on the Spot Core

6.1.1 Networking Setup

In order to access the ROS Master Node hosted on the Spot ROS package, the Docker container, the Spot Core and the Spot robot require correct port forwarding configuration. The Spot robot acts as a WiFi access point and router, mapping external traffic in the 21000 – 22000 port range to be forwarded to the Spot Core. Applications deployed to the Spot Core on Docker use the `host` networking mode. In this mode, applications within the container share the same networking namespace as the host, making all ports accessible under the same IP address [48]. In this setup, any ROS node on the Spot robot will be able to access the nodes in this Docker container, as they share the same host network.

6.2 Waypoint Revisit Accuracy

The Spot robot was initially manually controlled to record the GraphNav map, which was automatically uploaded to the Spot robot after recording finished. The GraphNav map was then downloaded with the `/spot/download_graph` ROS Service for future analysis. Next, the robot was made to traverse the recorded GraphNav graph with the fiducial localization node, as described in Figure 5.3. The robot would walk from waypoint to waypoint, stopping at each waypoint to collect data. Data of nearby fiducials' poses in the `body` frame of the robot were collected at each waypoint, and compared between runs to evaluate the revisit accuracy of the GraphNav navigation service on the Spot robot. The test was ran three times each, for both indoors and outdoors environments at Alfa 3, Ideon Science Park, Lund, Sweden.

The Mean Variation of fiducial detections over all runs were calculated with Equation 1, taking the average of the difference between maximum and minimum position values in (x, y, z) of fiducial detections. This gives an indication of the spread of position of the fiducial relative to the robot's body, over all runs conducted.

$$\text{Mean Variation} = \frac{1}{n}\sum_{i=1}^{n}\Big(\max(X_i) - \min(X_i)\Big) \tag{1}$$

where n is the total number of detections, X_i is the i-th set of detections, and $\max(X_i)$ and $\min(X_i)$ are the maximum and minimum values in the i-th set of detections, respectively.

Additionally, the Mean Variance was calculated, combining the variance of each detection of a fiducial at a waypoint across runs, defined in Equation 2.

$$\text{Mean Variance} = \frac{1}{n}\sum_{i=1}^{n}\text{Var}(X_i) \tag{2}$$

where $\text{Var}(X_i)$ is the variance of the i-th set of detections, referring to the number of waypoints with a fiducial nearby. The variance of each set of detections is calculated using Equation 3 for variance across data from all runs.

$$\text{Var}(X) = \frac{1}{n-1}\sum_{m=1}^{n}(X_m - \bar{X})^2 \tag{3}$$

where n is the total number of observations in the dataset, referring to the number of runs, X_m is the detection from the m-th run, and $\bar{X}$ is the mean of the detections across runs.

Both parameters were used to evaluate the waypoint revisit accuracy of the Spot robot as it follows the GraphNav map, from waypoint to waypoint. A lower value of mean variation and mean variance would imply a lower spread in the revisit position, resulting in a higher waypoint revisit accuracy.

6.2.1 Indoor results

The indoor test was carried out with one initialization fiducial for the Spot robot to localize itself to the start point of the map. There were a total of 19 different AprilTag fiducials placed along the path of the robot near waypoints in its GraphNav map. The GraphNav map had a total of 101 automatically placed waypoints, resulting in a total of 107 detections of the placed fiducials as it traversed the path. An example of an AprilTag fiducial placed near a waypoint is shown in Figure 6.2. AprilTags are placed at a similar height to Spot's body, allowing it to detect it with its built-in cameras without requiring a specific pose to capture the fiducial within its field of view.

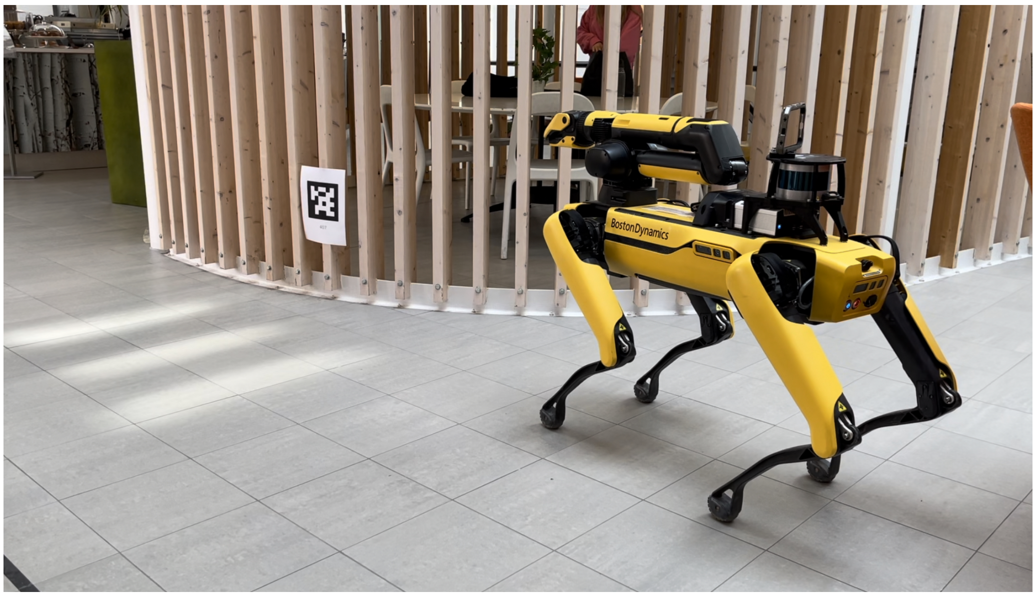


Figure 6.2: AprilTag fiducial near Spot during indoor localization test

Table 6.1: Indoor localization test result statistics

Fiducial Position	**Mean Variation (m)**	**Variance ($\mathbf{m^2}$)**
x	0.3343	0.0698
y	0.0623	0.0018
z	0.0178	0.00013

The histogram of fiducial localization variation for all detections across runs are shown in Figure 6.3. The plot of fiducial localization variation by waypoint in the path is shown in Figure 6.4. It can be seen that the variation in the y and z position is generally lower than that of x, as the latter two axes have occasional large spikes in variation. As the test was conducted in a public space with significant foot traffic, the robot had to sometimes deviate from its path to avoid people. It is likely that the large variation in position data points was due to this, as the number of data points varying by more than 0.5m is a small fraction of the total. Upon further inspection into the data, out of the three runs conducted, only data from Run 2 differed, where there was significant foot traffic and the robot had to manoeuvre around people in its path.

The variance of the dataset remains very small for both methods of calculation, indicating that the waypoint revisit accuracy based on fiducial data is precise. The robot was able to follow its path in a repeatable manner, with minimal deviation unless necessary.

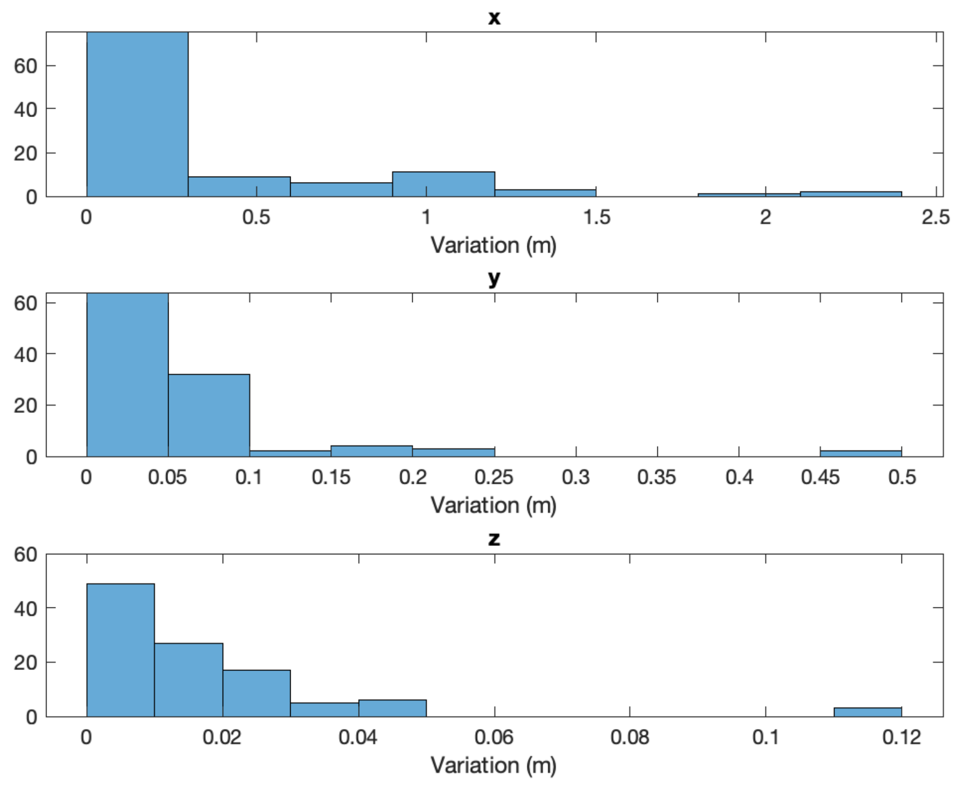


Figure 6.3: Variation in x, y, z across runs, indoors

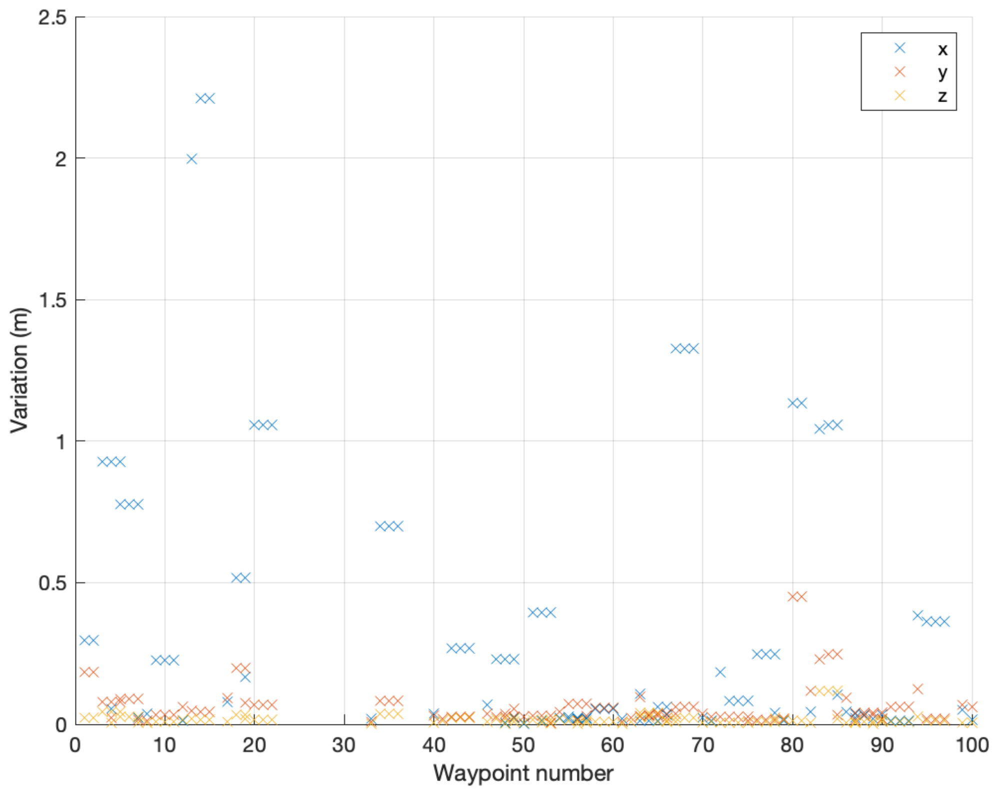


Figure 6.4: Variation in x, y, z across runs, by waypoint sequence in path, indoors

6.2.2 Outdoor results

For the outdoor test, there were a total of 10 different AprilTag fiducials placed along the path of the robot near waypoints in its GraphNav map. The GraphNav map had a total of 147 automatically placed waypoints, resulting in a total of 44 detections of the placed fiducials as it traversed the path. A lower density of waypoints was used due to the sparse environment leaving limited placements for fiducials that would be visible to the robot. As seen in Figure 6.5, when the Spot robot is in a park, there are no nearby positions to place a fiducial within 3 meters of the robot, the effective range of its cameras.

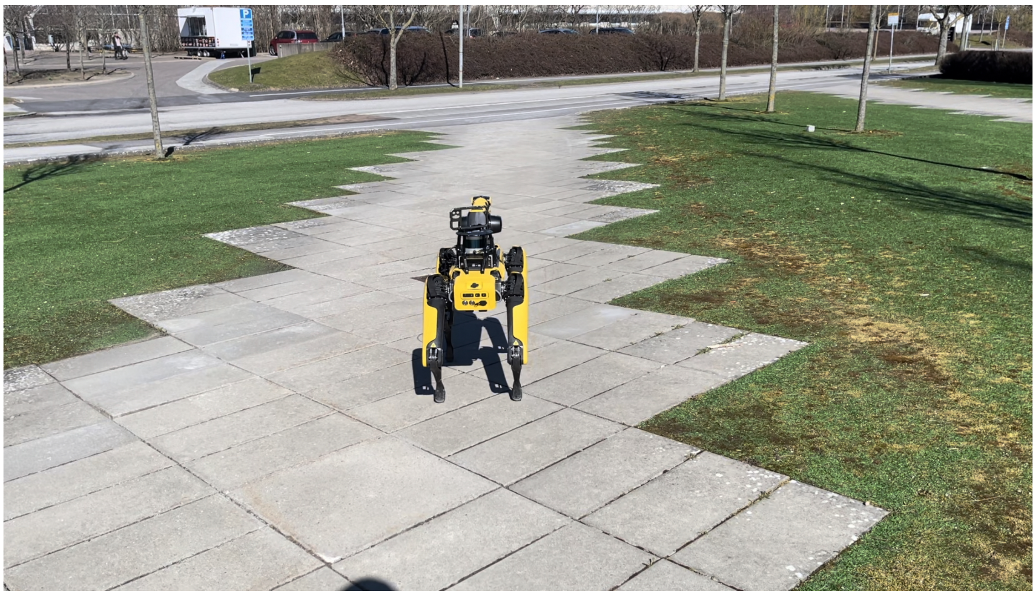


Figure 6.5: Spot robot during localization test in a park

The localization variation statistics of the outdoor run are shown in Table 6.2.

Table 6.2: Outdoor localization test result statistics

Fiducial Position	Mean Variation (m)	Variance (m^2)
x	0.2361	0.03240
y	0.0709	0.00151
z	0.0253	0.00015

The histogram of fiducial localization variation for all detections across all waypoints across runs are shown in Figure 6.6. The plot of fiducial localization variation by waypoint in the path is shown in Figure 6.7.

For both the indoor and outdoor datasets, the mean variation in y and z is very low as compared to the x position. In the reference frame of the Spot robot, this corresponds to objects in directly in front of the robot. This is likely due to the Spot arm obscuring the viewing angle of the LIDAR, thus Spot has to rely on its relatively less precise front depth cameras for localization, increasing the variation in its x-position during waypoint revisits. Comparing to the variance of the indoor dataset, it can be noted that the outdoor localization has a similar mean variation in position data. When performing the test outdoors, the density of foot traffic was much lower, thus Spot did not have to manoeuvre around obstacles in its path as often, allowing it to more accurately follow the recorded path. However, the environment was more feature sparse, thus feature-based localization would not be as effective. These two factors have opposite influences on the localization accuracy, thus there was no significant difference in variance between the indoors and outdoors results.

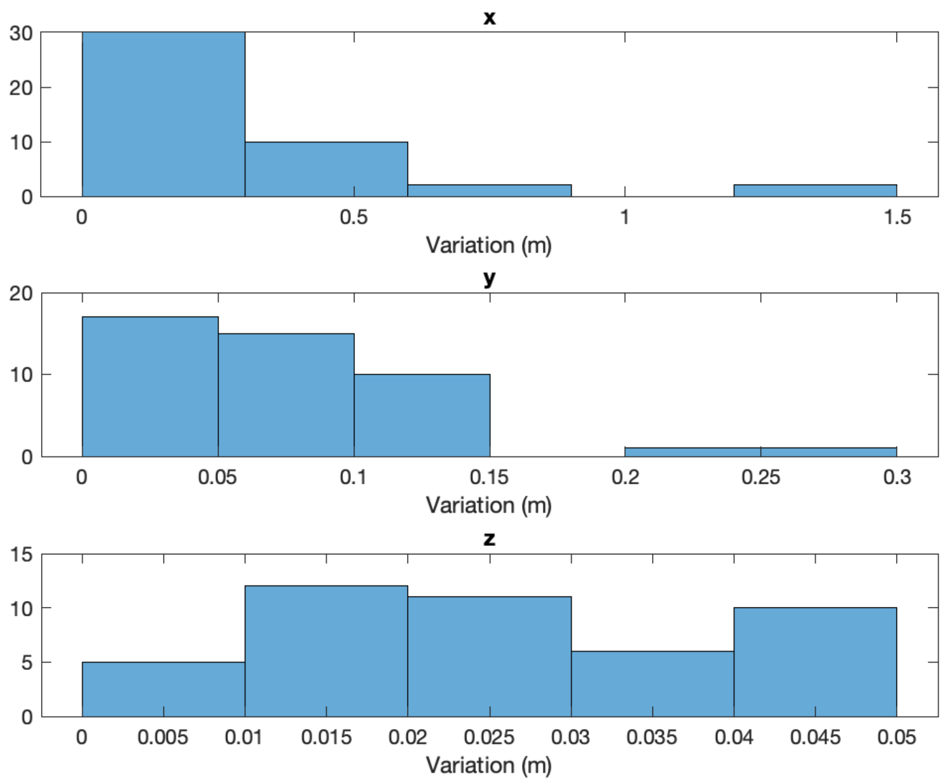


Figure 6.6: Variation in x, y, z across runs, outdoors

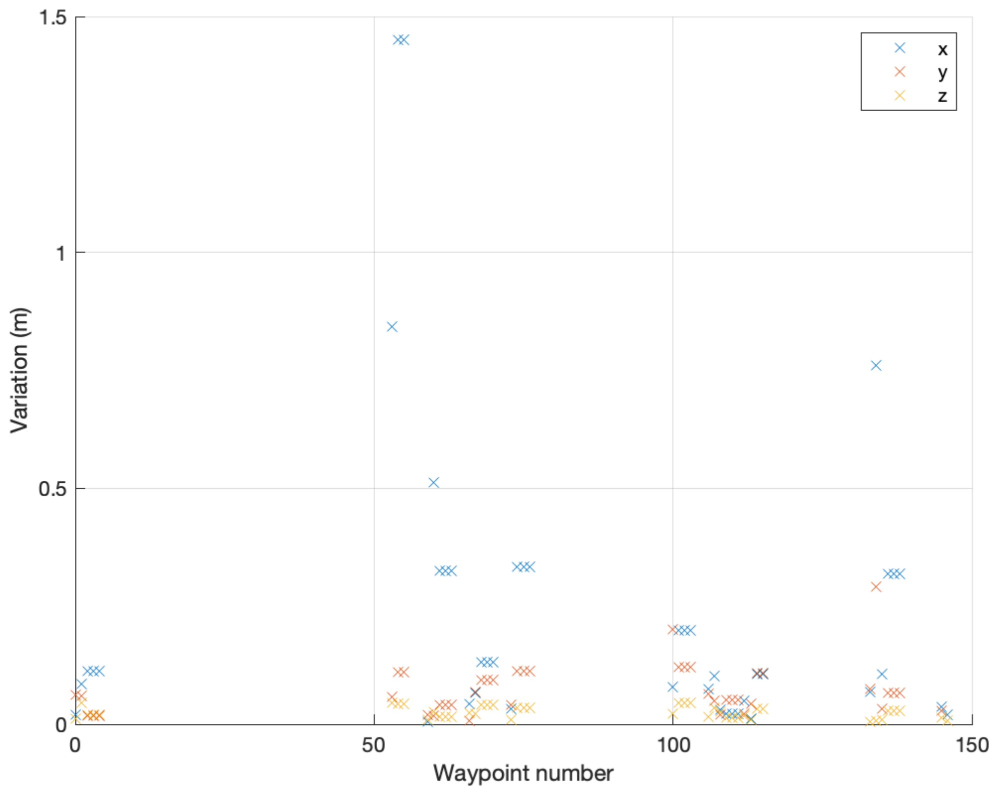


Figure 6.7: Variation in x, y, z across runs, by waypoint sequence in path, outdoors

6.3 Indoor Mapping

The point cloud reconstruction of the environment created by GraphNav is shown in Figure 6.8.

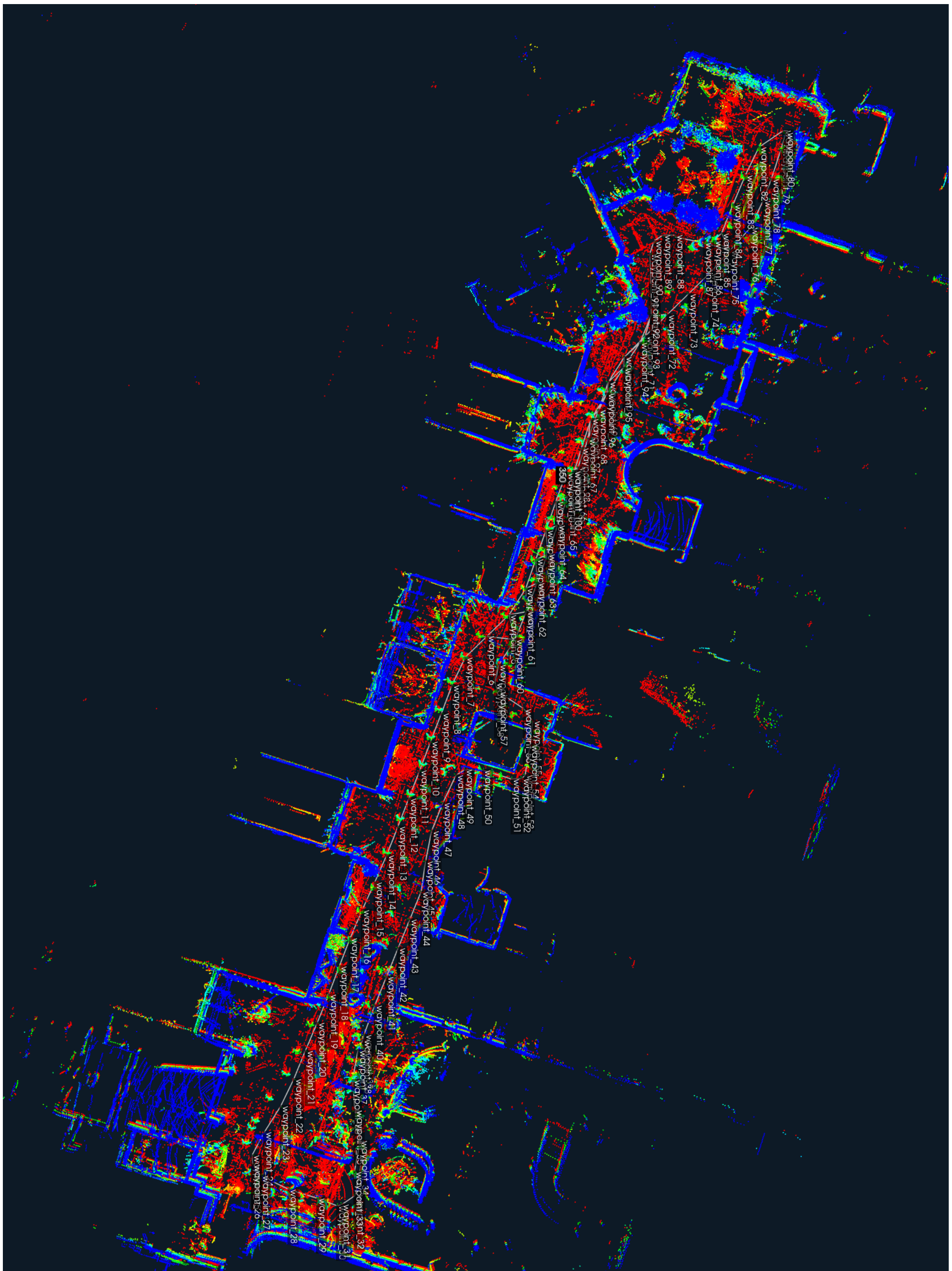


Figure 6.8: Point cloud reconstruction produced by GraphNav with generated waypoints

The 2D grid map created by the RTAB-Map SLAM package, walking the same route, is shown in Figure 6.9.

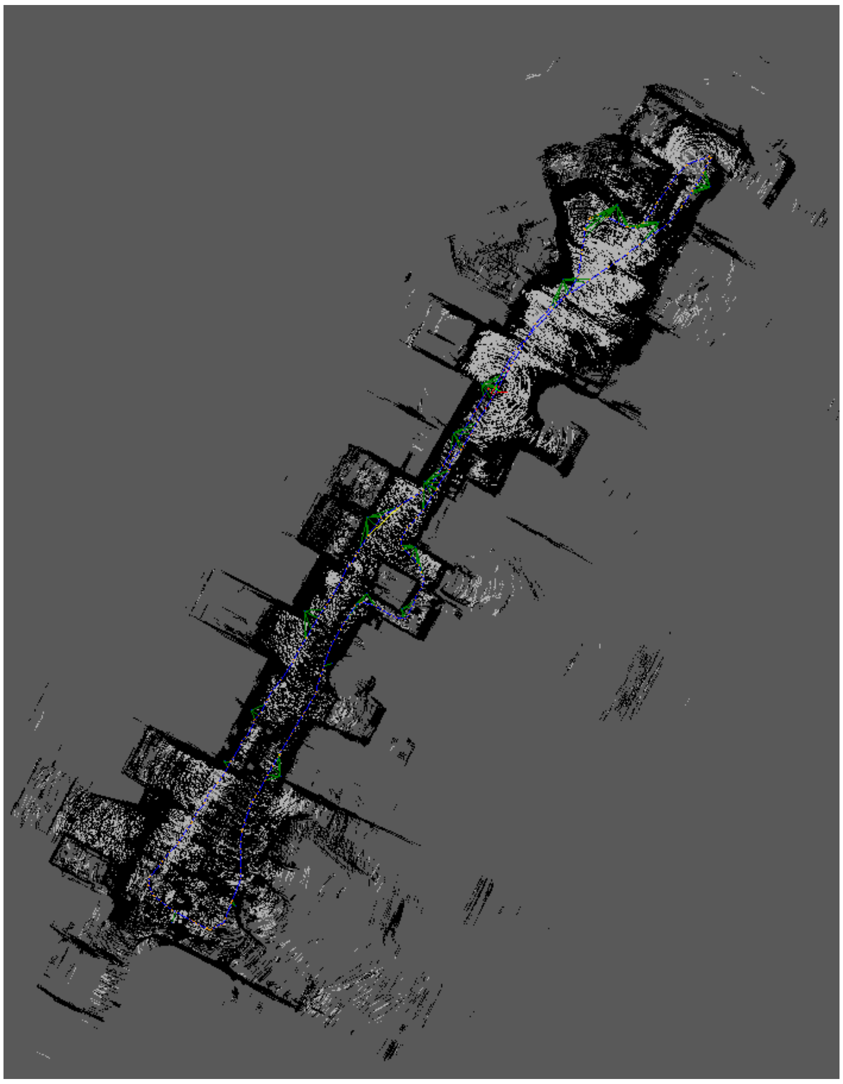


Figure 6.9: RTAB-Map indoor grid map with generated waypoints

Comparing the two maps generated by GraphNav and RTAB-Map in Figures 6.8 and 6.9 from the same input data, a similar level of detail and mapping range can be noted. The maps have been cleaned up using the Point Cloud Library in ROS and put side-by-side in Figure 6.10 to compare their features. It can be noted that the GraphNav map is significantly cleaner, as it only performs mapping once in the manually operated stage, and does not alter the map during operation.

By contrast, the RTAB-Map SLAM output performs mapping and localization simultaneously, adding onto the existing map even if it has visited the location before. This leads to the loop closure problem, where the robot is unable to recognise that it has visited the same physical location. This is particularly evident in the middle of the map produced by RTAB-Map in Figure 6.10, where the same walls have been mapped twice as separate obstacles. This occurs when the robot walks past once near the start of its run, and once more towards the end, and is unable to detect that it is at a previously visited location. This problem can be solved by tuning specific RTAB-Map parameters [49], which will vary with the type of environment the robot is in.

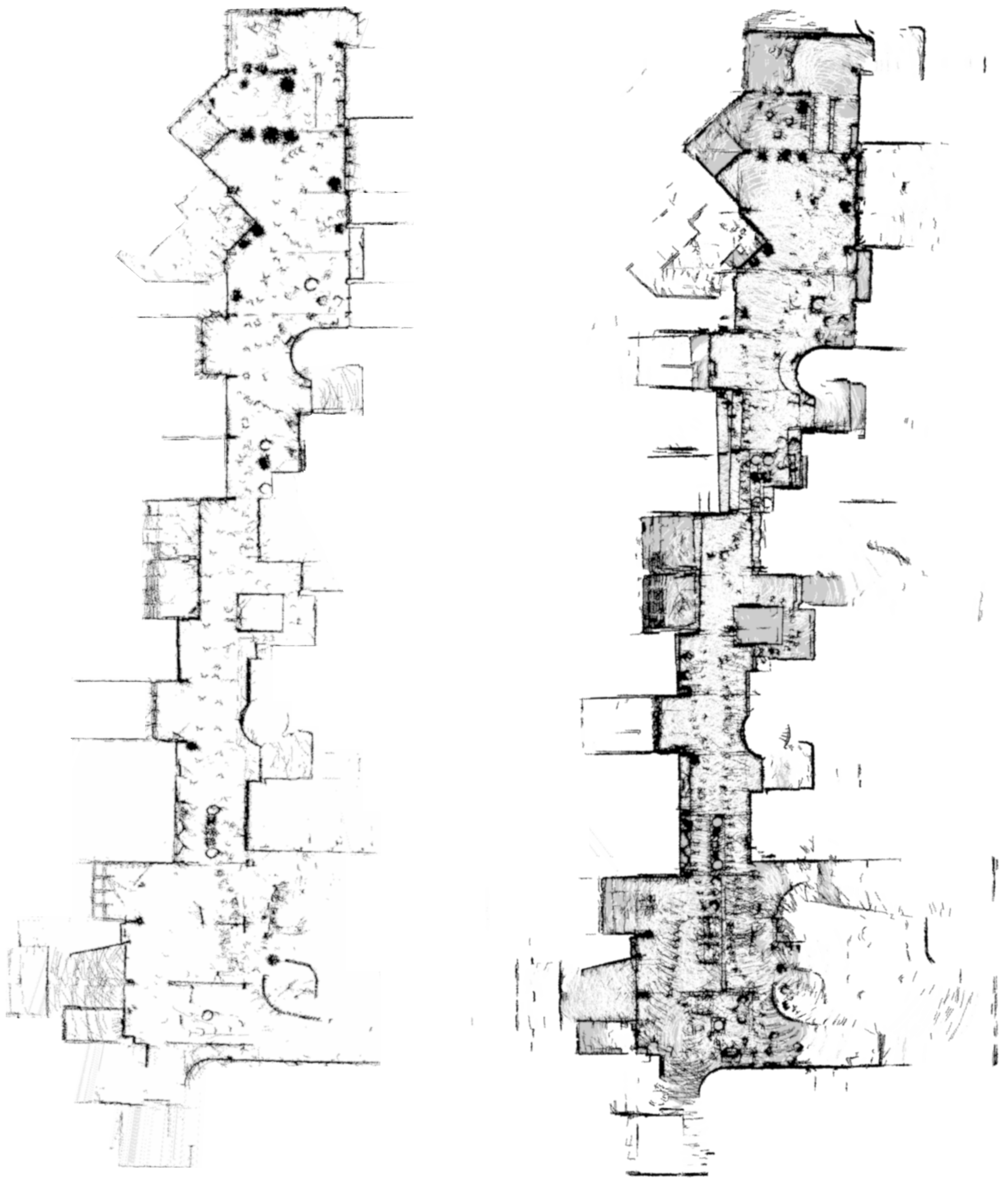


Figure 6.10: GraphNav (left) and RTAB-Map (right) mapping output

6.4 Indoor Path Planning Demonstration

In order to evaluate the performance of the entire navigation stack, it was decided to test an example task: navigating to a button and pressing it to open a door. The robot performed the following actions:

1. Map the area beforehand with manual control
2. Start at a known localization
3. Navigate to a manually defined goal location, near the button
4. Extend the arm to a set location. If the goal location has been accurately reached, the arm will press the button and open the door

This serves as a qualitative indication of the suitability of the each navigation stack in a practical robotics application.

6.4.1 GraphNav Navigation Service

The GraphNav navigation service was used through SpotROS. The robot was manually driven from the start to the goal location to generate the GraphNav map. The map was saved and used in the demonstration ROS application, seen in Appendix E. The robot then performed the tasks described previously, pressing the button to open the door, as seen in Figure 6.11. Over the six times the demonstration was run, Spot was successful in pressing the button six out of six times. In all cases, Spot was able to achieve the correct pose; facing the button directly for the arm to extend and press the button. However, there was some variation in the y-direction of the pose, resulting in one nearly missed attempt at pressing the button.

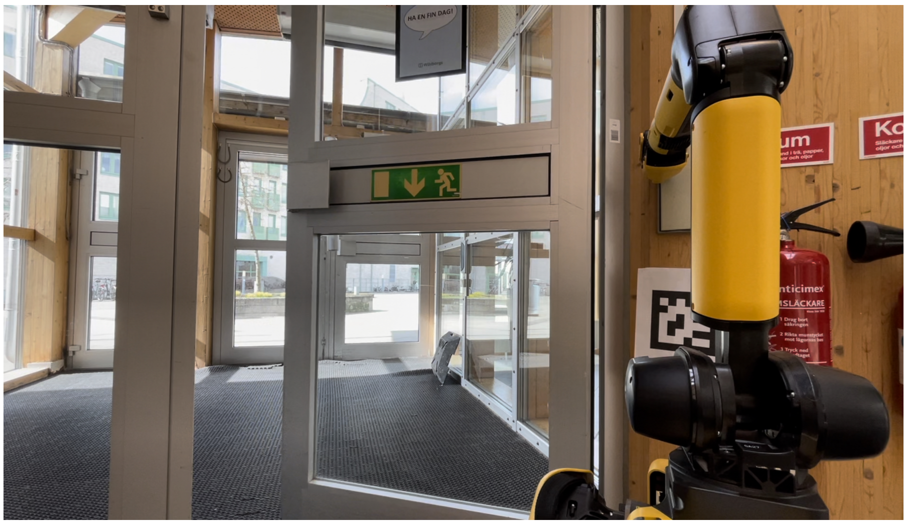
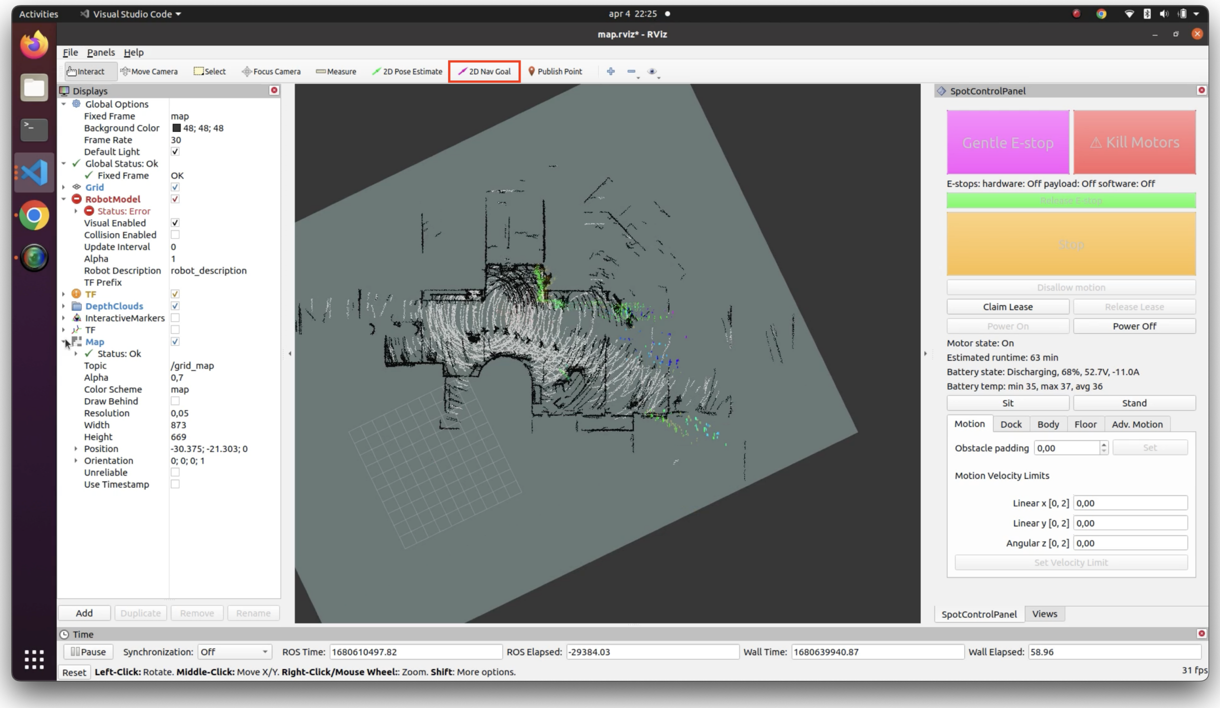


Figure 6.11: GraphNav demo: pressing a button

6.4.2 ROS Navigation Stack

To define the goal location, the `move_base` goal can be set using the `RViz` interface with the `2D Nav Goal`, as seen in Figure 6.12. Using the `2D Nav Goal` tool, the user can click on a point in the viewport to send a `move_base_simple/goal` to the `move_base` package. This will start the path planning and trigger the robot to move towards the point selected in the `map` frame. This was used to test the validity of the map produced by RTAB-Map and the navigation capabilities of Spot with the ROS navigation stack.



Global Options
Fixed Frame: map
Background Color: 48; 48; 48
Frame Rate: 30
Default Light: ✓
Global Status: Ok
Fixed Frame: OK
Grid: ✓
RobotModel: ✓
Status: Error
Visual Enabled: ✓
Collision Enabled: ☐
Update Interval: 0
Alpha: 1
Robot Description: robot_description
TF Prefix
TF: ✓
DepthClouds: ✓
InteractiveMarkers: ☐
TF: ☐
Map: ✓
Status: Ok
Topic: /grid_map
Alpha: 0,7
Color Scheme: map
Draw Behind: ☐
Resolution: 0,05
Width: 873
Height: 669
Position: -30.375; -21.303; 0
Orientation: 0; 0; 0; 1
Unreliable: ☐
Use Timestamp: ☐

E-stops: hardware: Off payload: Off software: Off
Motor state: On
Estimated runtime: 63 min
Battery state: Discharging, 68%, 52.7V, -11.0A
Battery temp: min 35, max 37, avg 36
Obstacle padding: 0,00
Motion Velocity Limits
Linear x [0, 2]: 0,00
Linear y [0, 2]: 0,00
Angular z [0, 2]: 0,00

Synchronization: Off ROS Time: 1680610497.82 ROS Elapsed: -29384.03 Wall Time: 1680639940.87 Wall Elapsed: 58.96

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:**: Zoom. **Shift**: More options.

Figure 6.12: RViz interface with SLAM mapping data

During testing of the ROS Navigation stack for the same task, the robot was able to consistently navigate to the goal location. However, the success rate of pressing the button heavily depended on the manually defined goal location in the `map` reference frame, as the SLAM package would update the map and it was not immediately apparent where to define the goal location based on the map in RViz.

6.4.3 Comparing GraphNav and ROS Navigation Stack

Both the GraphNav service and the ROS navigation stack were able to navigate the robot through the environment and position itself in front of the door opening button. With an appropriate button pose detection system and arm commands to move to that pose, the robot would be able to reliably press the button; as the navigation stack ensures that the robot will be within reach of the button at its destination.

However, while performing the navigation with GraphNav through the ROS package, it was noticed that the robot stutters in its walk, appearing to suddenly change its trajectory when deviating from the recorded path. When the same map was run through Autowalk on the controller, Spot's higher level navigation interface, the robot was able to traverse the path smoothly. It is possible that there are some parameters set by Autowalk that allow GraphNav to function smoothly, which could be investigated in the future development of the ROS package.

Compared to the original Spot SDK, the ROS package enables the use of complex functionality such as arm movement and autonomy features while GraphNav is active, which was previously not possible due to licensing limitations around the Autowalk feature of the Spot Explorer robot. This opens up the possibility of ROS developers mixing and matching features from the Spot robot and their own ROS packages to efficiently create a complete system to solve challenges specific to their domain. The application above with the RTAB-Map and ROS navigation stack showcases the interoperability of the Spot ROS package with open source ROS packages, lowering the barriers to entry of using Spot to existing ROS developers.

7 Conclusion and Future Work

7.1 Conclusion

The work carried out in this project has successfully created a ROS wrapper around the Spot SDK that is open-source friendly, lowering the barriers of entry to collaborators around the world through the use of open-source software best practices. The ROS package has been extensively tested to be working on the Spot robot with the Spot SDK v3.2.0, making available basic functionality such as image capture and movement, in addition to autonomy features such as the GraphNav service and automated diagnostic checks with Spot Check.

In order to keep up with updates in the Spot SDK, the ROS package has been set up for sustainable open-source development, with extensive automated unit testing, code modularity and clear documentation. In the future, if a breaking change were to be introduced in the Spot SDK, users could easily highlight them and submit their own pull requests to the package to fix them. They would not need to fully understand the inner workings of the package thanks to the modular software architecture; they would only need to know the interfaces required to introduce new features or that of existing ones. Moreover, the combination of explicit documentation, unit tests and commented code with clear names create several layers of documentation that lower barriers to entry for new users and contributors.

Lastly, the platform application successfully demonstrated the potential use cases for the Spot robot while validating its strong built-in navigation capabilities. Using the GraphNav service, an operator could select from a variety of pre-recorded maps to send out Spot robots, performing tasks such as mapping and data collection, or interacting with physical interfaces through the Spot arm. The main benefit of using the ROS wrapper would be the ease of integration of existing software already written on ROS for other platforms, saving the engineering work of re-writing software in the vendor-specific SDK. As compared to the base Spot Explorer robot and its license, the operator would be able to run advanced functions such as arm movement and picking up objects, which is not currently possible without an upgrade to the more expensive Spot Enterprise robot.

7.2 Future work

To capitalise on the work in this project, there are several opportunities for improvement:

1. Add new features to the ROS package as the Spot SDK expands to leverage its integrated autonomy features.
2. Develop a user-friendly graphical interface to interact with the Spot robot using the ROS package and RViz, to create a no-code development platform. This would be similar to what Universal Robots has done with Polyscope [50].
3. Adopt the ROS wrapper to lower barriers to deploying and testing state-of-the-art robotics research on the Spot robot platform.
4. Utilise the ROS wrapper in industrial applications with other open-source ROS packages, lowering the cost of industrial R&D in developing Spot for commercial applications.

In conclusion, this project has contributed to the field of robotics software development by creating a useful ROS package for the Spot robot, making it easier for robotics engineers to integrate the robot into their work processes. This project opens up the possibility for further research and development in this area, and we hope that this project serves as a useful resource for future work in this field, while continuing to be supported by the open-source community.

References

[1] Boston Dynamics. *Press Release: Commercial Sales Launch— Boston Dynamics*. 2020. URL: https://www.bostondynamics.com/press-release-spot-commercial-launch.

[2] Anthony Mills, Peter E. Love, and Peter Williams. "Defect Costs in Residential Construction". In: *Journal of Construction Engineering and Management* 135.1 (Jan. 2009), pp. 12–16. ISSN: 0733-9364. DOI: 10.1061/(ASCE)0733-9364(2009)135:1(12).

[3] Jimmy Bengtsson. *Veidekke Report Q2 2021*. 2021. URL: https://mb.cision.com/Public/17348/3399148/a2e0ac62d225f438.pdf.

[4] UK Atomic Energy Authority. *Annual Report and Accounts 2021/22*. Tech. rep. 2022. URL: https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/1092290/CPS21.463_v19.4_compressed.pdf.

[5] Boston Dynamics. *Spot Brochure*. 2022. URL: www.bostondynamics.com/products/spot.

[6] Boston Dynamics. *Spot Anatomy*. Oct. 2022. URL: https://support.bostondynamics.com/s/article/Spot-anatomy.

[7] Boston Dynamics. *Spot Arm Specifications and Key Concepts*. Oct. 2022. URL: https://support.bostondynamics.com/s/article/Spot-Arm-specifications-and-concepts.

[8] Boston Dynamics. *Enhanced Autonomy Payload Spot EAP*. 2022. URL: https://www.bostondynamics.com/sites/default/files/inline-files/spot-eap.pdf.

[9] Velodyne Lidar Inc. *High definition lidar system*. July 2007. URL: https://patents.google.com/patent/US7969558B2/en.

[10] Velodyne Lidar Inc. *Puck Lidar Sensor, High-Value Surround Lidar — Velodyne Lidar*. 2023. URL: https://velodynelidar.com/products/puck/.

[11] Jesús Morales et al. "Analysis of 3D Scan Measurement Distribution with Application to a Multi-Beam Lidar on a Rotating Platform". In: *Sensors 2018, Vol. 18, Page 395* 18.2 (Jan. 2018), p. 395. ISSN: 1424-8220. DOI: 10.3390/S18020395. URL: https://www.mdpi.com/1424-8220/18/2/395/htmhttps://www.mdpi.com/1424-8220/18/2/395.

[12] Valentin Koch. "Rotations in 3D Graphics and the Gimbal Lock Presentation Road Map". In: *IEEE Okanagan* (2016). URL: http://www.okanagan.ieee.ca/wp-content/uploads/2016/02/GimbalLockPresentationJan2016.pdf.

[13] Dexter C. Kozen. "Depth-First and Breadth-First Search". In: *The Design and Analysis of Algorithms*. New York, NY: Springer New York, 1992, pp. 19–24. DOI: 10.1007/978-1-4612-4400-4{_}4.

[14] Magnus Lie Hetland. "Magic Methods, Properties, and Iterators". In: *Beginning Python*. Berkeley, CA: Apress, 2017, pp. 163–193. DOI: 10.1007/978-1-4842-0028-5{_}9.

[15] Prathamesh Saraf, Abhishek Sarkar, and Arshad Javed. "Terrain Adaptive Gait Transitioning for a Quadruped Robot using Model Predictive Control". In: *2021 26th International Conference on Automation and Computing: System Intelligence through Automation and Computing, ICAC 2021* (2021). DOI: 10.23919/ICAC50006.2021.9594065.

[16] Boston Dynamics. *Boston Dynamics Choreographer Developer Guide*. 2023. URL: https://dev.bostondynamics.com/docs/concepts/choreography/choreographer.

[17] Xingwei Wang, Hong Zhao, and Jiakeng Zhu. "GRPC". In: *ACM SIGOPS Operating Systems Review* 27.3 (July 1993), pp. 75–86. ISSN: 0163-5980. DOI: 10.1145/155870.155881.

[18] Boston Dynamics. *Spot API Concepts*. 2023. URL: https://dev.bostondynamics.com/docs/concepts/readme.

[19] Boston Dynamics. *Networking*. 2023. URL: https://dev.bostondynamics.com/docs/concepts/networking.

[20] Morgan Quigley et al. *ROS An Open-Source Robot Operating System*. Tech. rep. Stanford University, 2009. URL: http://robotics.stanford.edu/~ang/papers/icraoss09-ROS.pdf.

[21] Yanqing Wu. *ROS Master*. Jan. 2018. URL: http://wiki.ros.org/Master.

[22] Srihitha Yerabaka and Ivan Marsic. *Analysis of the Trade Offs and Benefits of Using the Publisher Subscriber Design Pattern Technical Paper*. Tech. rep. 2011. URL: http://eceweb1.rutgers.edu/~marsic/books/SE/projects/AutoHome/2011-report_Srihitha.pdf.

[23] Vincenzo Diluoffo, William R Michalson, and Berk Sunar. "Robot Operating System 2: The need for a holistic security approach to robotic architectures". In: *International Journal of Advanced Robotic Systems* (2018). DOI: 10.1177/1729881418770011. URL: https://us.sagepub.com/en-us/nam/.

[24] ROS Metrics. *Top Downloaded ROS Packages*. 2022. URL: https://metrics.ros.org/packages_top.html.

[25] Microsoft. *ROS Wrapper for the Boston Dynamics Spot robot*. 2020. URL: https://github.com/microsoft/spot-ros-wrapper.

[26] Chris Bogdon. *Clearpath Robotics Releases ROS Package for Boston Dynamics' Spot Robot*. Sept. 2020. URL: https://clearpathrobotics.com/blog/2020/09/clearpath-robotics-releases-ros-package-for-boston-dynamics-spot-robot/.

[27] Charles Anderson. "Docker". In: *IEEE Software* 32.3 (May 2015), pp. 102–105. ISSN: 07407459. DOI: 10.1109/MS.2015.62.

[28] Boston Dynamics. *Running Custom Applications with Spot*. 2023. URL: https://dev.bostondynamics.com/docs/payload/docker_containers.

[29] Mark Aberdour. "Achieving quality in open-source software". In: *IEEE Software* 24.1 (Jan. 2007), pp. 58–64. ISSN: 07407459. DOI: 10.1109/MS.2007.2.

[30] Richard N. Langlois and Giampaolo Garzarelli. "Of Hackers and Hairdressers: Modularity and the Organizational Economics of Open-source Collaboration". In: *Industry and Innovation* 15.2 (Apr. 2008), pp. 125–143. ISSN: 1366-2716. DOI: 10.1080/13662710801954559.

[31] Carliss Y Baldwin and Kim B Clark. "The Architecture of Participation: Does Code Architecture Mitigate Free Riding in the Open Source Development Model?" In: *Management Science* 52.7 (2006), pp. 1116–1127. ISSN: 00251909, 15265501. URL: http://www.jstor.org/stable/20110584.

[32] Facebook. *React library for web and native user interfaces*. Mar. 2023. URL: https://github.com/facebook/react.

[33] Google. *Chromium The official GitHub mirror of the Chromium source*. Mar. 2023. URL: https://github.com/chromium/chromium.

[34] Mathias Meyer. "Continuous Integration and Its Tools". In: *IEEE Software* 31.3 (May 2014), pp. 14–16. ISSN: 0740-7459. DOI: 10.1109/MS.2014.58.

[35] Siti Nurmaini and Bambang Tutuko. "Intelligent Robotics Navigation System: Problems, Methods, and Algorithm". In: *International Journal of Electrical and Computer Engineering (IJECE)* 7.6 (Dec. 2017), p. 3711. ISSN: 2088-8708. DOI: 10.11591/ijece.v7i6.pp3711-3726.

[36] M. Betke and L. Gurvits. "Mobile robot localization using landmarks". In: *IEEE Transactions on Robotics and Automation* 13.2 (Apr. 1997), pp. 251–263. ISSN: 1042296X. DOI: 10.1109/70.563647.

[37] A. Elfes. "Using occupancy grids for mobile robot perception and navigation". In: *Computer* 22.6 (June 1989), pp. 46–57. ISSN: 0018-9162. DOI: 10.1109/2.30720.

[38] H. Durrant-Whyte and T. Bailey. "Simultaneous localization and mapping: part I". In: *IEEE Robotics & Automation Magazine* 13.2 (June 2006), pp. 99–110. ISSN: 1070-9932. DOI: 10.1109/MRA.2006.1638022.

[39] Matthew R. Walter, Ryan M. Eustice, and John J. Leonard. "Exactly Sparse Extended Information Filters for Feature-based SLAM". In: *The International Journal of Robotics Research* 26.4 (Apr. 2007), pp. 335–359. ISSN: 0278-3649. DOI: 10.1177/0278364906075026.

[40] Boston Dynamics. *GraphNav Technical Summary*. Oct. 2022. URL: https://bostondynamics.force.com/SupportCenter/s/article/GraphNav-Technical-Summary.

[41] Michal Staniaszek. *ROS driver for controlling Boston Dynamics Spot robot*. Mar. 2023. URL: https://github.com/heuristicus/spot_ros.

[42] Davide Spadini et al. "To Mock or Not to Mock? An Empirical Study on Mocking Practices". In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, May 2017, pp. 402–412. ISBN: 978-1-5386-1544-7. DOI: 10.1109/MSR.2017.61.

[43] Boston Dynamics. *Recalibration with SpotCheck*. Oct. 2022. URL: https://support.bostondynamics.com/s/article/Recalibration-with-SpotCheck#WhentorunSpotCheck.

[44] Mathieu Labbé and François Michaud. “RTAB-Map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation”. In: *Journal of Field Robotics* 36.2 (Mar. 2019), pp. 416–446. ISSN: 1556-4967. DOI: 10.1002/ROB.21831. URL: https://onlinelibrary.wiley.com/doi/full/10.1002/rob.21831https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21831https://onlinelibrary.wiley.com/doi/10.1002/rob.21831.

[45] Boston Dynamics. *Comparing Spot Explorer and Spot Enterprise*. Oct. 2022. URL: https://support.bostondynamics.com/s/article/Comparing-Spot-Explorer-and-Spot-Enterprise.

[46] Nicholas Fragale. *move_base ROS Wiki*. Sept. 2020. URL: http://wiki.ros.org/move_base.

[47] Jihoon Lee. *navfn ROS Wiki*. Oct. 2014. URL: http://wiki.ros.org/navfn.

[48] Docker. *Docker Host Networking*. 2023. URL: https://docs.docker.com/network/host/.

[49] Mathieu Labbé. *RTAB-Map ROS Advanced Parameter Tuning*. Apr. 2023. URL: http://wiki.ros.org/rtabmap_ros/Tutorials/Advanced%20Parameter%20Tuning.

[50] Universal Robots. *Program in your language with PolyScope*. 2023. URL: https://www.universal-robots.com/products/polyscope/.

[51] Mathieu Labbé. *rtabmap_ros ROS Wiki*. Dec. 2022. URL: http://wiki.ros.org/rtabmap_ros.

A SpotROS Topics, Services and Actions

Table A.1: ROS services available

Topic	Type	Description
/spot/claim	Trigger	Claim the lease
/spot/release	Trigger	Release the lease
/spot/self_right	Trigger	Flip onto the legs
/spot/sit	Trigger	Sit the robot
/spot/stand	Trigger	Stand the robot
/spot/power_on	Trigger	Enable motor power
/spot/power_off	Trigger	Disable motor power
/spot/estop/hard	Trigger	Trigger the estop immediately
/spot/estop/gentle	Trigger	Trigger the estop gracefully
/spot/estop/release	Trigger	Release the estop
/spot/allow_motion	SetBool	Enable or disable motion
/spot/stair_mode	SetBool	Enable or disable stair climbing mode
/spot/locomotion_mode	SetLocomotion	Set locomotion_mode
/spot/swing_height	SetSwingHeight	Set swing_height mode
/spot/velocity_limit	SetVelocity	Set velocity_limit as Twist message
/spot/clear_behavior_fault	ClearBehaviorFault	Clear faults by id
/spot/terrain_params	SetTerrainParams	Set ground_mu_hint and grated surfaces mode
/spot/obstacle_params	SetObstacleParams	Set obstacle avoidance parameters
/spot/posed_stand	PosedStand	Set body height and pose
/spot/list_graph	ListGraph	Get the list of waypoints in the active graph
/spot/download_graph	DownloadGraph	Download the list of waypoints in the active graph
/spot/navigate_init	NavigateInit	Localization with a fiducial in the provided map
/spot/graph_close_loops	GraphCloseLoops	Request loop closure in the active graph
/spot/optimize_graph_anchoring	Trigger	Request anchor optimization in the active graph
/spot/roll_over_right	Trigger	Roll body over to the right
/spot/roll_over_left	Trigger	Roll body over to the left
/spot/dock	Dock	Dock at dock_id
/spot/undock	Trigger	Undock the robot
/spot/docking_state	GetDockState	Get the dock_state of the robot
/spot/spot_check	SpotCheck	Run Spot Check self-calibration
/spot/arm_stow	Trigger	Arm stow position
/spot/arm_unstow	Trigger	Arm unstow position
/spot/gripper_open	Trigger	Gripper open position
/spot/gripper_close	Trigger	Gripper close position
/spot/arm_carry	Trigger	Arm carry position
/spot/gripper_angle_open	GripperAngleMove	Gripper open to specific angle
/spot/arm_joint_move	ArmJointMovement	Arm move to specific angle for each join
/spot/force_trajectory	ArmForceTrajectory	Arm move as a wrench
/spot/gripper_pose	HandPose	Gripper move to Pose in a specific frame
/spot/grasp_3d	Grasp3d	Execute a pick at a Position in a specific frame
/spot/arm_gaze	Trigger	Arm gaze towards the front, for demonstration
/spot/stop	Trigger	Stop the robot
/spot/locked_stop	Trigger	Stop the robot and disallow motion
/spot/spot_ros/tf2_frames	Trigger	Get tf2 transformation frames as a FrameGraph

Table A.2: ROS publishers available

Topic	Type	Description
/clock	rosgraph_msgs/Clock	ROS simulation time
/diagnostics	diagnostic_msgs/DiagnosticArray	ROS diagnostics
/joint_states	sensor_msgs/JointState	Spot joint pose states
/rosout	rosgraph_msgs/Log	Logging messages
/rosout_agg	rosgraph_msgs/Log	Logging messages
/spot/body_pose/status	actionlib_msgs/GoalStatusArray	Body pose from standing action server
/spot/camera/back/camera_info	sensor_msgs/CameraInfo	Spot monochrome fisheye camera info
/spot/camera/back/image	sensor_msgs/Image	Spot monochrome fisheye camera data
/spot/camera/frontleft/camera_info	sensor_msgs/CameraInfo	Spot monochrome fisheye camera info
/spot/camera/frontleft/image	sensor_msgs/Image	Spot monochrome fisheye camera data
/spot/camera/frontright/camera_info	sensor_msgs/CameraInfo	Spot monochrome fisheye camera info
/spot/camera/frontright/image	sensor_msgs/Image	Spot monochrome fisheye camera data
/spot/camera/hand_color/camera_info	sensor_msgs/CameraInfo	Spot colour hand camera info
/spot/camera/hand_color/image	sensor_msgs/Image	Spot colour hand camera data
/spot/camera/hand_mono/camera_info	sensor_msgs/CameraInfo	Spot monochrome hand camera info
/spot/camera/hand_mono/image	sensor_msgs/Image	Spot monochrome hand camera data
/spot/camera/hand/depth_in_color/camera_info	sensor_msgs/CameraInfo	Spot colour and depth hand camera data
/spot/camera/left/camera_info	sensor_msgs/CameraInfo	Spot monochrome fisheye camera info
/spot/camera/left/image	sensor_msgs/Image	Spot monochrome fisheye camera data
/spot/camera/right/camera_info	sensor_msgs/CameraInfo	Spot monochrome fisheye camera info
/spot/camera/right/image	sensor_msgs/Image	Spot monochrome fisheye camera data
/spot/depth/back/camera_info	sensor_msgs/CameraInfo	Spot depth camera info
/spot/depth/back/image	sensor_msgs/Image	Spot depth camera data
/spot/depth/frontleft/camera_info	sensor_msgs/CameraInfo	Spot depth camera info
/spot/depth/frontleft/image	sensor_msgs/Image	Spot depth camera data
/spot/depth/frontright/camera_info	sensor_msgs/CameraInfo	Spot depth camera info
/spot/depth/frontright/image	sensor_msgs/Image	Spot depth camera data
/spot/depth/hand/camera_info	sensor_msgs/CameraInfo	Spot depth hand camera info
/spot/depth/hand/depth_in_color	sensor_msgs/Image	Spot depth colour hand camera data
/spot/depth/hand/image	sensor_msgs/Image	Spot depth hand camera data
/spot/depth/left/camera_info	sensor_msgs/CameraInfo	Spot depth in visual camera info
/spot/depth/left/image	sensor_msgs/Image	Spot depth in visual camera data
/spot/depth/right/camera_info	sensor_msgs/CameraInfo	Spot depth in visual camera info
/spot/depth/right/image	sensor_msgs/Image	Spot depth in visual camera data
/spot/depth/back/depth_in_visual/camera_info	sensor_msgs/CameraInfo	Spot depth in visual camera info
/spot/depth/back/depth_in_visual	sensor_msgs/Image	Spot depth in visual camera data
/spot/depth/frontleft/depth_in_visual/camera_info	sensor_msgs/CameraInfo	Spot depth in visual camera info
/spot/depth/frontleft/depth_in_visual	sensor_msgs/Image	Spot depth in visual camera data
/spot/depth/frontright/depth_in_visual/camera_info	sensor_msgs/CameraInfo	Spot depth in visual camera info
/spot/depth/frontright/depth_in_visual/camera_info	sensor_msgs/Image	Spot depth in visual camera data
/spot/depth/left/depth_in_visual/camera_info	sensor_msgs/CameraInfo	Spot depth in visual camera info
/spot/depth/left/depth_in_visual	sensor_msgs/Image	Spot depth in visual camera data
/spot/depth/right/depth_in_visual/camera_info	sensor_msgs/CameraInfo	Spot depth in visual camera info
/spot/depth/right/depth_in_visual	sensor_msgs/Image	Spot depth in visual camera data
/spot/dock/status	actionlib_msgs/GoalStatusArray	Spot docking status
/spot/lidar/points	sensor_msgs/PointCloud2	Spot EAP Velodye VLP-16 LIDAR data
/spot/motion_or_idle_body_pose/status	actionlib_msgs/GoalStatusArray	Body pose with movement action server
/spot/navigate_to/status	actionlib_msgs/GoalStatusArray	GraphNav navigation action server
/spot/odometry	nav_msgs/Odometry	Spot odometry, ”vision” or ”odom”
/spot/odometry/twist	geometry_msgs/TwistWithCovarianceStamped	Legacy compatibility with /spot/cmd_vel
/spot/status/battery_states	spot_msgs/BatteryStateArray	Spot status: battery state
/spot/status/behavior_faults	spot_msgs/BehaviorFaultState	Spot status: behavior fault
/spot/status/estop	spot_msgs/EStopStateArray	Spot status: emergency stop (estop)
/spot/status/feedback	spot_msgs/Feedback	Spot status: standing, sitting, moving, version
/spot/status/feet	spot_msgs/FootStateArray	Spot status: feet state
/spot/status/leases	spot_msgs/LeaseArray	Spot status: active lease holder
/spot/status/metrics	spot_msgs/Metrics	Spot status: distance and power usage metrics
/spot/status/mobility_params	spot_msgs/MobilityParams	Spot status: velocity limit, other parameters
/spot/status/motion_allowed	std_msgs/Bool	Spot status: motion enabled status
/spot/status/power_state	spot_msgs/PowerState	Spot status: charging, shore power, battery power
/spot/status/system_faults	spot_msgs/SystemFaultState	Spot status: system fault
/spot/status/wifi	spot_msgs/WiFiState	Spot status: wifi ESSID, current mode
/spot/trajectory/status	actionlib_msgs/GoalStatusArray	Trajectory pose movement action server
/spot/world_objects	spot_msgs/WorldObjectArray	Spot world object detections
/tf	tf2_msgs/TFMessage	Tranformation frames from TF2 API
/tf_static	tf2_msgs/TFMessage	Tranformation frames from TF2 API
/twist_marker_server/update	visualization_msgs/InteractiveMarkerUpdate	Legacy compatibility with /spot/cmd_vel
/twist_marker_server/update_full	visualization_msgs/InteractiveMarkerInit	Legacy compatibility with /spot/cmd_vel

Table A.3: ROS actions available

Topic	Type	Description
/spot/navigate_to	NavigateToAction	Go to a GraphNav waypoint
/spot/navigate_route	NavigateRouteAction	Go to a series of GraphNav waypoints
/spot/trajectory	TrajectoryAction	Move robot to a Pose
/spot/motion_or_idle_body_pose	PoseBodyAction	Rotate body to a pose
/spot/body_pose	PoseBodyAction	Stand and move robot to a Pose
/spot/dock	DockAction	Dock or undock at a specific dock_id

B Unit Testing Implementation

B.1 Library Unit Tests

An example of testing the `ros_helpers.GetWifiFromState` helper function, which translates messages from the Spot format to ROS format. This function does not have any other code dependencies and does not require ROS, so it is appropriate to test it in isolation.

```python
#!/usr/bin/env python3
PKG = "ros_helpers"
NAME = "ros_helpers_test"
SUITE = "ros_helpers_test.TestSuiteROSHelpers"

import unittest

import spot_driver.ros_helpers as ros_helpers
from spot_driver.spot_wrapper import SpotWrapper
from google.protobuf import duration_pb2
from bosdyn.api import robot_state_pb2


class TestSpotWrapper(SpotWrapper):
    def __init__(self):
        pass

    @property
    def time_skew(self) -> duration_pb2.Duration:
        robot_time_skew = duration_pb2.Duration(seconds=0, nanos=0)
        return robot_time_skew


class TestGetWifiFromState(unittest.TestCase):
    def test_get_wifi_from_state(self):
        state = robot_state_pb2.RobotState()
        spot_wrapper = TestSpotWrapper()
        initial_wifi_state = robot_state_pb2.WiFiState(
            current_mode=robot_state_pb2.WiFiState.MODE_ACCESS_POINT, essid="test_essid"
        )
        state.comms_states.add(wifi_state=initial_wifi_state)

        wifi_state = ros_helpers.GetWifiFromState(state, spot_wrapper)
        self.assertEqual(
            wifi_state.current_mode, robot_state_pb2.WiFiState.MODE_ACCESS_POINT
        )
        self.assertEqual(wifi_state.essid, "test_essid")


if __name__ == "__main__":
    import rosunit

    rosunit.unitrun(PKG, NAME, TestGetWifiFromState)
```

B.2 ROS Unit Tests

An example of testing the `PointCloudCB` function used in SpotROS, which is triggered periodically to publish data from Spot to ROS. It requires ROS to test the full functionality of this function. First, we set up a mocked version of SpotROS running in another node to run the `PointCloudCB` function as if it were in a production environment, with fixed data.

```python
#!/usr/bin/env python3
import rospy
import typing

from bosdyn.api import point_cloud_pb2
from google.protobuf import duration_pb2, timestamp_pb2

from spot_driver.spot_ros import SpotROS
from spot_driver.spot_wrapper import SpotWrapper

# Stubbed SpotWrapper class for testing
class TestSpotWrapper(SpotWrapper):
    def __init__(self):
        self._point_cloud = [point_cloud_pb2.PointCloudResponse()]

    @property
    def time_skew(self) -> duration_pb2.Duration:
        robot_time_skew = duration_pb2.Duration(seconds=0, nanos=0)
        return robot_time_skew

    @property
    def point_clouds(self) -> typing.List[point_cloud_pb2.
        PointCloudResponse]:
        """Return latest _point_cloud data"""
        return self._point_cloud

    @point_clouds.setter
    def point_clouds(
        self, point_cloud: typing.List[point_cloud_pb2.PointCloudResponse
            ]
    ):
        """Set the _point_cloud data"""
        self._point_cloud = point_cloud


class MockSpotROS:
    def __init__(self):
        self.spot_ros = SpotROS()
        self.spot_ros.node_name = "mock_spot_ros"
        self.spot_ros.spot_wrapper = TestSpotWrapper()

    def set_point_cloud_data(self):
        # Create PointCloudResponse data
        point_cloud_data = point_cloud_pb2.PointCloudResponse()
        point_cloud_data.status = point_cloud_pb2.PointCloudResponse.
            STATUS_OK
        point_cloud = point_cloud_pb2.PointCloud(
            source=point_cloud_pb2.PointCloudSource(
                name="test_point_cloud",
                frame_name_sensor="eap",
                acquisition_time=timestamp_pb2.Timestamp(seconds=1, nanos=2),
                transforms_snapshot=None,
            ),
            num_points=3,
            encoding=point_cloud_pb2.PointCloud.ENCODING_XYZ_32F,
            encoding_parameters=None,
            data=b"\x00\x00\x80?\x00\x00\x00@\x00\x00\x80@\x00\x00\x80?\x00\x00\x00@\x00\x00\x80@\x00\x00\x80?\x00\x00\x00@\x00\x00\x80@",
        )
        point_cloud_data.point_cloud.CopyFrom(point_cloud)
        self.spot_ros.spot_wrapper.point_clouds = [point_cloud_data]

    def main(self):
        rospy.init_node(self.spot_ros.node_name, anonymous=True)
        # Initialize variables for transforms
        self.spot_ros.mode_parent_odom_tf = "vision"
        self.spot_ros.tf_name_kinematic_odom = "odom"
        self.spot_ros.tf_name_raw_kinematic = "odom"
        self.spot_ros.tf_name_vision_odom = "vision"
        self.spot_ros.tf_name_raw_vision = "vision"

        # Set up the ROS publishers, subscribers, services, and action servers
        self.spot_ros.initialize_publishers()
        self.spot_ros.initialize_subscribers()
        self.spot_ros.initialize_services()
        self.spot_ros.initialize_action_servers()

        # Manually set robot images data
        self.set_point_cloud_data()

        # Set running rate
        rate = rospy.Rate(50)

        while not rospy.is_shutdown():
            # Call publish callbacks
            for callback_name, callback in self.spot_ros.callbacks.items():
                callback(callback_name)

            rate.sleep()


if __name__ == "__main__":
    run_spot_ros = MockSpotROS()
    run_spot_ros.main()
```

Next, we set up a `unittest` to subscribe to the `/spot/lidar/points` topic and ensure that the fixed data is being passed correctly.

```python
#!/usr/bin/env python3
PKG = "spot_ros"
NAME = "spot_ros_test"
SUITE = "spot_ros_test.TestSuiteSpotROS"

import unittest
import rospy
import time

from sensor_msgs.msg import PointCloud2


class TestPointCloudCB(unittest.TestCase):
    def setUp(self):
        self.data = {}

    def point_cloud_cb(self, msg: PointCloud2):
        self.data["point_cloud"] = msg

    def check_point_cloud(self, point_cloud_msg: PointCloud2):
        # Check that the point cloud message is correctly populated
        self.assertEqual(point_cloud_msg.header.frame_id, "eap")
        self.assertEqual(point_cloud_msg.header.stamp.secs, 1)
        self.assertEqual(point_cloud_msg.header.stamp.nsecs, 2)
        self.assertEqual(point_cloud_msg.height, 1)
        self.assertEqual(point_cloud_msg.width, 3)
        self.assertEqual(point_cloud_msg.fields[0].name, "x")
        self.assertEqual(point_cloud_msg.fields[0].offset, 0)
        self.assertEqual(point_cloud_msg.fields[0].datatype, 7)
        self.assertEqual(point_cloud_msg.fields[0].count, 1)
        self.assertEqual(point_cloud_msg.fields[1].name, "y")
        self.assertEqual(point_cloud_msg.fields[1].offset, 4)
        self.assertEqual(point_cloud_msg.fields[1].datatype, 7)
        self.assertEqual(point_cloud_msg.fields[1].count, 1)
        self.assertEqual(point_cloud_msg.fields[2].name, "z")
        self.assertEqual(point_cloud_msg.fields[2].offset, 8)
        self.assertEqual(point_cloud_msg.fields[2].datatype, 7)
        self.assertEqual(point_cloud_msg.fields[2].count, 1)
        self.assertEqual(point_cloud_msg.is_bigendian, False)
        self.assertEqual(point_cloud_msg.point_step, 12)
        self.assertEqual(point_cloud_msg.row_step, 36)
        self.assertEqual(
            point_cloud_msg.data,
            b"\x00\x00\x80?\x00\x00\x00@\x00\x00\x80@\x00\x00\x80?\x00\
               x00\x00@\x00\x00\x80@\x00\x00\x80?\x00\x00\x00@\x00\x00\
               x80@",
        )
        self.assertEqual(point_cloud_msg.is_dense, True)

    def test_point_cloud_cb(self):
        self.point_cloud = rospy.Subscriber(
            "/spot/lidar/points", PointCloud2, self.point_cloud_cb
        )

        counter = 0
        while not rospy.is_shutdown() and counter < 10:
            time.sleep(1)
            counter += 1

        # Check that we got all the data
        self.assertTrue("point_cloud" in self.data, "Point cloud is empty
            ")

        # Check that the data is valid
        self.check_point_cloud(self.data["point_cloud"])


if __name__ == "__main__":
    print("Starting tests!")
    import rosunit

    rospy.init_node(NAME, anonymous=True)
    rosunit.unitrun(PKG, NAME, TestPointCloudCB)
```

C Fiducial Localization

C.1 Fiducial Python Object

```python
import typing
import numpy as np
from geometry_msgs.msg import PoseStamped


class Fiducial:
    """
    Fiducial class that stores the fiducial information in standard ROS
        geometry_msgs/Pose format
    """

    def __init__(
        self,
        tag_id: int,
        dim_x: float,
        dim_y: float,
        fiducial_pose: PoseStamped,
        filtered_fiducial_pose: PoseStamped,
        pose_covariance: typing.Optional[typing.List[float]] = None,
        pose_covariance_frame: typing.Optional[str] = None,
    ):
        # Save variables to class
        self.tag_id = tag_id
        self.dim_x = dim_x
        self.dim_y = dim_y
        self.fiducial_pose = fiducial_pose
        self.filtered_fiducial_pose = filtered_fiducial_pose
        if pose_covariance and pose_covariance_frame:
            self.pose_covariance = pose_covariance  # Row-major
            self.pose_covariance_frame = pose_covariance_frame

    def distance_to_fiducial(self, other: "Fiducial"):
        """Returns the distance between two fiducials"""
        return np.linalg.norm(
            np.array(
                [
                    self.fiducial_pose.pose.position.x,
                    self.fiducial_pose.pose.position.y,
                    self.fiducial_pose.pose.position.z,
                ]
            )
            - np.array(
                [
                    other.fiducial_pose.pose.position.x,
                    other.fiducial_pose.pose.position.y,
                    other.fiducial_pose.pose.position.z,
                ]
            )
        )
```

C.2 SpotNav Localization ROS Node

```python
#!/usr/bin/env python3
import typing
import pickle
import time

import rospy
import actionlib
from actionlib_msgs.msg import GoalStatus
from rosservice import get_service_class_by_name

from std_msgs.msg import String
from geometry_msgs.msg import PoseStamped
from spot_msgs.msg import WorldObjectArray, WorldObject
from spot_msgs.msg import AprilTagProperties
from spot_msgs.msg import FrameTreeSnapshot, ParentEdge
from spot_msgs.msg import NavigateInitAction, NavigateInitGoal
from spot_msgs.msg import NavigateToAction, NavigateToGoal
from spot_msgs.srv import ListGraphResponse
from std_srvs.srv import TriggerRequest

from fiducial import Fiducial


class SpotNav:
    def __init__(self):
        self.world_objects = None
        self.fiducials_seen: typing.Dict[int, typing.List["Fiducial"]] =
            {}
        self.waypoint_fiducial: typing.Dict[str, typing.List["Fiducial"]]
             = {}

    def initialize_subscribers(self):
        """Initialize ROS subscribers"""
        # Create a subscriber for the /spot/world_objects topic for
            WorldObjectArray messages
        self.world_objects_sub = rospy.Subscriber(
            "/spot/world_objects", WorldObjectArray, self.
                world_objects_callback
        )

    def initialize_publishers(self):
        """Initialize ROS publishers"""
        self.reached_waypoint_pub = rospy.Publisher(
            "/spot/nav/reached_waypoint", String, queue_size=1
        )

    def initialize_action_clients(self):
        """Initialize ROS action clients"""
        # Create an action client for the /spot/navigate_to action
        self.navigate_to_client = actionlib.SimpleActionClient(
            "/spot/navigate_to", NavigateToAction
        )

        # Create an action client for the /spot/navigate_init action
        self.navigate_init_client = actionlib.SimpleActionClient(
            "/spot/navigate_init", NavigateInitAction
        )

    def world_objects_callback(self, msg: WorldObjectArray):
        # Save the message to a class variable tracking the fiducials in the message
        self.world_objects: typing.List[WorldObject] = msg.world_objects

        # Iterate through the fiducials in the message, append the x,y,z coordinates to a dictionary
        for world_object in self.world_objects:
            april_tag: AprilTagProperties = world_object.apriltag_properties
            latest_snapshot: FrameTreeSnapshot = world_object.frame_tree_snapshot

            # Check if april_tag is None
            if april_tag is None:
                continue

            # Create the FrameTreeSnapshot as a dictionary
            frame_tree_snapshot: typing.Dict[str, PoseStamped] = {}
            for child, parent_edge in zip(
                latest_snapshot.child_edges, latest_snapshot.parent_edges
            ):
                parent_edge_transform = PoseStamped()
                parent_edge_transform.header.stamp = world_object.acquisition_time
                parent_edge_transform.header.frame_id = parent_edge.parent_frame_name
                parent_edge_transform.pose = parent_edge.parent_tform_child

                frame_tree_snapshot[child] = parent_edge_transform

            april_tag_pose = frame_tree_snapshot[f"fiducial_{april_tag.tag_id}"]
            april_tag_pose_filtered = frame_tree_snapshot[
                f"filtered_fiducial_{april_tag.tag_id}"
            ]

            # Build the april_tag into the Fiducial class
            fiducial = Fiducial(
                tag_id=april_tag.tag_id,
                dim_x=april_tag.x,
                dim_y=april_tag.y,
                fiducial_pose=april_tag_pose,
                filtered_fiducial_pose=april_tag_pose_filtered,
                pose_covariance=april_tag.detection_covariance,
                pose_covariance_frame=april_tag.
```

```
                    detection_covariance_reference_frame,
            )

            # Save the fiducial to the class variable
            if fiducial.tag_id in self.fiducials_seen:
                self.fiducials_seen[fiducial.tag_id].append(fiducial)
            else:
                self.fiducials_seen[fiducial.tag_id] = [fiducial]

            rospy.logdebug(
                f"fiducial: {fiducial.tag_id}\n position_x: {fiducial.
                    fiducial_pose.pose.position.x:.2f}\n
                    filtered_position_x: {fiducial.filtered_fiducial_pose.
                    pose.position.x:.2f}"
            )

    def call_service(self, service_name: str, *args, **kwargs):
        """Call a service and wait for it to be available"""
        try:
            rospy.wait_for_service(service_name)
            service_type = get_service_class_by_name(service_name)
            proxy = rospy.ServiceProxy(service_name, service_type)
            return proxy(*args, **kwargs)

        except rospy.ServiceException as e:
            rospy.logerr("Service call failed: %s" % e)

    def walk_current_graph(self):
        """Walk the current graph in GraphNav"""
        rospy.loginfo("Walking the current graph")

        # Call the ListGraph service
        list_graph: ListGraphResponse = self.call_service("/spot/
            list_graph")
        waypoints = list_graph.waypoint_ids

        # Call the /spot/navigate_init action
        navigate_init_goal = NavigateInitGoal(
            upload_path="",
            initial_localization_fiducial=True,
            initial_localization_waypoint="mm",
        )
        self.navigate_init_client.send_goal(navigate_init_goal)
        self.navigate_init_client.wait_for_result()

        # Check if the action succeeded
        if self.navigate_init_client.get_state() == GoalStatus.SUCCEEDED:
            rospy.loginfo("NavigateInit action succeeded")

        for waypoint in waypoints:
            # Call the /spot/navigate_to action
            navigate_to_goal = NavigateToGoal(navigate_to=waypoint)
            self.navigate_to_client.send_goal(navigate_to_goal)
```

```python
            self.navigate_to_client.wait_for_result()

            # Check if the action succeeded
            if self.navigate_to_client.get_state() == GoalStatus.SUCCEEDED:
                rospy.loginfo(f"NavigateTo {waypoint} action succeeded")
                latest_world_objects_msg = rospy.wait_for_message(
                    "/spot/world_objects", WorldObjectArray
                )
                latest_fiducial = self.extract_fiducials(latest_world_objects_msg)
                if waypoint in self.waypoint_fiducial:
                    self.waypoint_fiducial[waypoint].append(latest_fiducial)
                else:
                    self.waypoint_fiducial[waypoint] = [latest_fiducial]

                # Publish to the /spot/nav/reached_waypoint topic
                self.reached_waypoint_pub.publish(waypoint)
                time.sleep(0.5)

    def extract_fiducials(self, msg: WorldObjectArray) -> typing.List["Fiducial"]:
        self.world_objects: typing.List[WorldObject] = msg.world_objects
        fiducial_list = []

        # Iterate through the fiducials in the message, append the x,y,z coordinates to a dictionary
        for world_object in self.world_objects:
            april_tag: AprilTagProperties = world_object.apriltag_properties
            latest_snapshot: FrameTreeSnapshot = world_object.frame_tree_snapshot

            # Check if april_tag is None
            if april_tag is None:
                continue

            # Create the FrameTreeSnapshot as a dictionary
            frame_tree_snapshot: typing.Dict[str, PoseStamped] = {}
            for child, parent_edge in zip(
                latest_snapshot.child_edges, latest_snapshot.parent_edges
            ):
                parent_edge_transform = PoseStamped()
                parent_edge_transform.header.stamp = world_object.acquisition_time
                parent_edge_transform.header.frame_id = parent_edge.parent_frame_name
                parent_edge_transform.pose = parent_edge.parent_tform_child

                frame_tree_snapshot[child] = parent_edge_transform

            # Use tf2 to get the april_tag pose in the body frame
            april_tag_pose = frame_tree_snapshot[f"fiducial_{april_tag.tag_id}"]
            april_tag_pose_filtered = frame_tree_snapshot[
                f"filtered_fiducial_{april_tag.tag_id}"
            ]

            # Build the april_tag into the Fiducial class
            fiducial = Fiducial(
                tag_id=april_tag.tag_id,
                dim_x=april_tag.x,
                dim_y=april_tag.y,
                fiducial_pose=april_tag_pose,
                filtered_fiducial_pose=april_tag_pose_filtered,
                pose_covariance=april_tag.detection_covariance,
                pose_covariance_frame=april_tag.detection_covariance_reference_frame,
            )

            # Append the fiducial to the list
            fiducial_list.append(fiducial)

        return fiducial_list

    def startup(self):
        rospy.loginfo("SpotNav robot starting up")

        # Call the /spot/claim, /spot/power_on, /spot/stand service
        self.call_service("/spot/claim", TriggerRequest())
        self.call_service("/spot/power_on", TriggerRequest())

    def shutdown(self):
        rospy.loginfo("SpotNav node shutting down")

        # Save the fiducials to a pickle file
        with open(f"fiducials_seen.pickle_{time.time()}", "wb") as f:
            pickle.dump(self.fiducials_seen, f)
        with open(f"waypoint_fiducial.pickle_{time.time()}", "wb") as f:
            pickle.dump(self.waypoint_fiducial, f)

        # Call the /spot/sit, /spot/power_off, /spot/release service
        self.call_service("/spot/sit", TriggerRequest())
        self.call_service("/spot/power_off", TriggerRequest())
        self.call_service("/spot/release", TriggerRequest())

    def main(self):
        rospy.init_node("spot_nav", anonymous=True)

        self.rates = rospy.get_param("~rates", {})
        if "loop_frequency" in self.rates:
            loop_rate = self.rates["loop_frequency"]
        else:
            loop_rate = 50

        # Initialize the node
        rate = rospy.Rate(loop_rate)
        rospy.loginfo("SpotNav node started")

        self.initialize_subscribers()
        self.initialize_publishers()
        self.initialize_action_clients()

        rospy.on_shutdown(self.shutdown)

        # Walk the current graph
        self.startup()
        self.walk_current_graph()

        while not rospy.is_shutdown():
            rate.sleep()


if __name__ == "__main__":
    spot_nav = SpotNav()
    spot_nav.main()
```

D RTABMap

D.1 Launch file used to run RTABMap with SpotROS

```xml
<launch>
    <!-- Spot camera names -->
    <arg name="camera_name1" default="frontleft" />
    <arg name="camera_name2" default="frontright" />
    <arg name="camera_name3" default="left" />
    <arg name="camera_name4" default="right" />
    <arg name="camera_name5" default="back" />

    <!-- Common frame name -->
    <arg name="master_frame" default="body" />

   <!-- sync rgb/depth images per camera -->
    <group ns="$(arg camera_name1)">
        <node pkg="nodelet" type="nodelet" name="rgbd_sync" args="standalone rtabmap_ros/rgbd_sync camera1_nodelet_manager">
            <remap from="rgb/image"         to="/spot/camera/frontleft/image"/>
            <remap from="depth/image"       to="/spot/depth/frontleft/depth_in_visual"/>
            <remap from="rgb/camera_info"   to="/spot/camera/frontleft/camera_info"/>
        </node>
    </group>

    <group ns="$(arg camera_name2)">
        <node pkg="nodelet" type="nodelet" name="rgbd_sync" args="standalone rtabmap_ros/rgbd_sync camera2_nodelet_manager">
            <remap from="rgb/image"         to="/spot/camera/frontright/image"/>
            <remap from="depth/image"       to="/spot/depth/frontright/depth_in_visual"/>
            <remap from="rgb/camera_info"   to="/spot/camera/frontright/camera_info"/>
        </node>
    </group>

    <group ns="$(arg camera_name3)">
        <node pkg="nodelet" type="nodelet" name="rgbd_sync" args="standalone rtabmap_ros/rgbd_sync camera3_nodelet_manager">
            <remap from="rgb/image"         to="/spot/camera/left/image"/>
            <remap from="depth/image"       to="/spot/depth/left/depth_in_visual"/>
            <remap from="rgb/camera_info"   to="/spot/camera/left/camera_info"/>
        </node>
    </group>

    <group ns="$(arg camera_name4)">
        <node pkg="nodelet" type="nodelet" name="rgbd_sync" args="
```

```
            standalone rtabmap_ros/rgbd_sync camera4_nodelet_manager">
            <remap from="rgb/image"          to="/spot/camera/right/image"/>
            <remap from="depth/image"        to="/spot/depth/right/depth_in_visual"/>
            <remap from="rgb/camera_info"    to="/spot/camera/right/camera_info"/>
        </node>
    </group>

    <group ns="$(arg camera_name5)">
        <node pkg="nodelet" type="nodelet" name="rgbd_sync" args="standalone rtabmap_ros/rgbd_sync camera5_nodelet_manager">
            <remap from="rgb/image"          to="/spot/camera/back/image"/>
            <remap from="depth/image"        to="/spot/depth/back/depth_in_visual"/>
            <remap from="rgb/camera_info"    to="/spot/camera/back/camera_info"/>
        </node>
    </group>

    <!-- Launch rtabmap node -->
    <arg name="strategy"             default="0"  />
    <arg name="feature"              default="6"  />
    <arg name="nn"                   default="3"  />
    <arg name="max_depth"            default="4.0"  />
    <arg name="min_inliers"          default="20"  />
    <arg name="inlier_distance"      default="0.02"  />
    <arg name="local_map"            default="1000"  />
    <arg name="odom_info_data"       default="true"  />
    <arg name="wait_for_transform"   default="true"  />

    <node name="rtabmap" pkg="rtabmap_ros" type="rtabmap" output="screen" args="--delete_db_on_start">

      <!-- Set the frame_id and the camera number. -->
      <param name="frame_id" type="string" value="$(arg master_frame)"/>
      <param name="rgbd_cameras"     type="int"     value="5"/>

      <!-- Subscribe to topics -->
      <param name="subscribe_depth" type="bool" value="false"/>
      <param name="subscribe_rgbd" type="bool" value="true"/>
      <param name="subscribe_rgb"   type="bool" value="false"/>
      <param name="subscribe_scan_cloud" type="bool" value="true"/>

      <remap from="odom"                to="/spot/odometry"/>
      <remap from="rgbd_image0"         to="/$(arg camera_name1)/rgbd_image"/>
      <remap from="rgbd_image1"         to="/$(arg camera_name2)/rgbd_image"/>
      <remap from="rgbd_image2"         to="/$(arg camera_name3)/rgbd_image"/>
      <remap from="rgbd_image3"       to="/$(arg camera_name4)/rgbd_image
         "/>
      <remap from="rgbd_image4"       to="/$(arg camera_name5)/rgbd_image
         "/>

      <remap from="scan_cloud" to="/spot/lidar/points"/>
      <param name="queue_size" type="int" value="100"/>

      <!-- RTAB-Map's parameters -->
      <param name="RGBD/NeighborLinkRefining" type="string" value="true
         "/>
      <param name="RGBD/ProximityBySpace"     type="string" value="true
         "/>
      <param name="RGBD/AngularUpdate"        type="string" value
         ="0.01"/>
      <param name="RGBD/LinearUpdate"         type="string" value
         ="0.01"/>
      <param name="RGBD/OptimizeFromGraphEnd" type="string" value="false
         "/>
      <param name="Grid/Sensor"               type="string" value="false
         "/> <!-- occupancy grid from lidar -->
      <param name="Reg/Force3DoF"             type="string" value="true
         "/>

      <param name="Reg/Strategy"              type="string" value="1"/>
         <!-- 1=ICP -->

    <!-- Reg/Strategy=0: all transforms are computed with only visual
       features
    Reg/Strategy=1: loop closure is computed with visual->icp (visual
       gives the rough guess to ICP), but proximity detection and
       neighbor refining are ICP only based on guess from odometry.
    Reg/Strategy=2: loop closure, proximity detection and neighbot
       refining are computed with visual->icp (visual gives the rough
       guess to ICP) -->

      <param name="Optimizer/Strategy" type="string" value="1"/> <!-- g2o
         =1, GTSAM=2 -->
      <param name="Optimizer/Robust" type="string" value="true"/>
      <param name="RGBD/OptimizeMaxError" type="string" value="0"/> <!--
         should be 0 if RGBD/OptimizeRobust is true -->
      <param name="RGBD/ProximityPathMaxNeighbors" type="string" value
         ="1"/>

      <!-- ICP parameters -->
      <param name="Icp/VoxelSize"                    type="string" value
         ="0.05"/>
      <param name="Icp/MaxCorrespondenceDistance" type="string" value
         ="0.1"/>
    </node>

    <node pkg="rtabmap_ros" type="rtabmapviz" name="rtabmapviz" args="-d
       $(find rtabmap_ros)/launch/config/rgbd_gui.ini" output="screen">
```

```
        <param name="frame_id"          type="string" value="$(arg
            master_frame)"/>
        <param name="rgbd_cameras"      type="int"     value="5"/>
        <param name="subscribe_depth" type="bool" value="false"/>
        <param name="subscribe_rgbd"  type="bool" value="true"/>
        <param name="subscribe_rgb"   type="bool" value="false"/>
        <param name="subscribe_scan_cloud" type="bool" value="true"/>

        <remap from="odom"                to="/spot/odometry"/>
        <remap from="rgbd_image0"         to="/$(arg camera_name1)/
            rgbd_image"/>
        <remap from="rgbd_image1"         to="/$(arg camera_name2)/
            rgbd_image"/>
        <remap from="rgbd_image2"         to="/$(arg camera_name3)/
            rgbd_image"/>
        <remap from="rgbd_image3"         to="/$(arg camera_name4)/
            rgbd_image"/>
        <remap from="rgbd_image4"         to="/$(arg camera_name5)/
            rgbd_image"/>
        <remap from="scan_cloud"          to="/spot/lidar/points"/>

        <param name="queue_size" type="int" value="10"/>

        <!-- RTAB-Map's parameters -->
        <param name="RGBD/NeighborLinkRefining" type="string" value="true
            "/>
        <param name="RGBD/ProximityBySpace"     type="string" value="true
            "/>
        <param name="RGBD/AngularUpdate"        type="string" value="0.01
            "/>
        <param name="RGBD/LinearUpdate"         type="string" value="0.01
            "/>
        <param name="RGBD/OptimizeFromGraphEnd" type="string" value="
            false"/>
        <param name="Grid/Sensor"               type="string" value="
            false"/> <!-- occupancy grid from lidar -->
        <param name="Reg/Force3DoF"             type="string" value="true
            "/>

        <param name="Reg/Strategy"              type="string" value="1"/>
            <!-- 1=ICP -->
    </node>

</launch>
```

D.2 Static transformations used to run RTAB-Map with multiple cameras and LIDAR

```
transforms:
  -
    header:
      seq: 0
      stamp:
        secs: 1679301766
        nsecs: 540450967
      frame_id: "odom"
    child_frame_id: "sensor_origin_velodyne-point-cloud"
    transform:
      translation:
        x: 0.0
        y: 0.0
        z: 0.0
      rotation:
        x: 0.0
        y: 0.0
        z: 0.0
        w: 1.0
  ---
    header:
      seq: 0
      stamp:
        secs: 1679301766
        nsecs: 540450967
      frame_id: "body"
    child_frame_id: "sensor"
    transform:
      translation:
        x: -0.20249999999999999
        y: 5.889326457099621e-17
        z: 0.15140000000000006
      rotation:
        x: -1.2246467991473532e-16
        y: 1.2246467991473532e-16
        z: -1.2246467991473532e-16
        w: 1.0
  ---
    header:
      seq: 0
      stamp:
        secs: 1679301766
        nsecs: 566367552
      frame_id: "body"
    child_frame_id: "head"
    transform:
      translation:
        x: 0.0
        y: 0.0
        z: 0.0
      rotation:
        x: 0.0
        y: 0.0
        z: 0.0
        w: 1.0
  ---
    header:
      seq: 0
      stamp:
        secs: 1679301766
        nsecs: 566367552
      frame_id: "head"
    child_frame_id: "frontleft"
    transform:
      translation:
        x: 0.41584859624905335
        y: 0.03372758931166825
        z: 0.023396574237304735
      rotation:
        x: 0.15096495861025827
        y: 0.8181056005688829
        z: -0.2243797904925827
        w: 0.5075101153751865
  -
    header:
      seq: 0
      stamp:
        secs: 1679301766
        nsecs: 566367552
      frame_id: "frontleft"
    child_frame_id: "frontleft_fisheye"
    transform:
      translation:
        x: 0.07395234011247184
        y: -0.0021491004112093675
        z: 0.0021008177206913864
      rotation:
        x: -0.005857619976460257
        y: -0.01989237191822775
        z: 0.0018884866640862738
        w: 0.9997831842183573
  -
    header:
      seq: 0
      stamp:
        secs: 1679301766
        nsecs: 566318052
      frame_id: "head"
    child_frame_id: "frontright"
    transform:
      translation:
        x: 0.41574367948389285
        y: -0.040277395005084926
        z: 0.022765156724030566
      rotation:
        x: -0.14218081416413436
        y: 0.8134772956750377
        z: 0.22654286082264116
        w: 0.5164471296416939
  -
    header:
      seq: 0
      stamp:
        secs: 1679301766
        nsecs: 566318052
      frame_id: "frontright"
    child_frame_id: "frontright_fisheye"
    transform:
      translation:
        x: 0.07370037358812871
        y: -0.002572908499647855
        z: 0.001306228715126155
      rotation:
        x: -0.006640041354622794
        y: -0.007809483919858992
        z: -0.0024134113004908478
        w: 0.999944547091292
  ---
    header:
      seq: 0
      stamp:
        secs: 1679301766
        nsecs: 613055120
      frame_id: "body"
    child_frame_id: "arm0.link_wr1"
    transform:
      translation:
        x: 0.3577354848384857
        y: 9.638628398533882e-06
        z: 0.2643103897571563
      rotation:
        x: -0.00015957005432197857
        y: 0.006054385450569923
        z: -0.0008680898384561889
        w: 0.999981282511951
  ---
    header:
      seq: 0
      stamp:
        secs: 1679301766
        nsecs: 613055120
      frame_id: "arm0.link_wr1"
    child_frame_id: "hand_depth_sensor"
    transform:
      translation:
        x: 0.13495
        y: 0.0
        z: 0.00799
      rotation:
        x: 0.0
        y: 0.6494478914666875
        z: 0.0
        w: 0.7604060995740853
  ---
    header:
      seq: 0
      stamp:
        secs: 1679301766
        nsecs: 670493912
      frame_id: "arm0.link_wr1"
    child_frame_id: "hand_color_image_sensor"
    transform:
      translation:
        x: 0.13806
        y: 0.020205
        z: 0.02452
      rotation:
        x: -0.45922900808339967
        y: 0.45922900808339967
        z: -0.5376883094644489
        w: 0.5376883094644489
  ---
    header:
      seq: 0
      stamp:
        secs: 1679302032
        nsecs: 888783208
      frame_id: "back"
    child_frame_id: "back_fisheye"
    transform:
      translation:
        x: 0.07250199778049198
        y: -0.004697571151119375
        z: 0.0008090978989160067
      rotation:
        x: -0.0033799982900291742
        y: 0.0016682267870559732
        z: 0.003176679910507995
        w: 0.9999878505940423
  ---
    header:
      seq: 0
      stamp:
        secs: 1679302032
        nsecs: 888783208
      frame_id: "head"
    child_frame_id: "back"
    transform:
      translation:
        x: -0.41801409129365225
        y: -0.0371806812178456
        z: 0.008218024594506435
      rotation:
        x: 0.5601189910358034
        y: 0.5631057792314684
        z: -0.4350068772936583
        w: -0.42420232670787483
  ---
```

```
    header:
      seq: 0
      stamp:
        secs: 1679302032
        nsecs: 889014208
      frame_id: "head"
    child_frame_id: "left"
    transform:
      translation:
        x: -0.16535957741030677
        y: 0.10950589198696871
        z: 0.03458287977297543
      rotation:
        x: -0.7987032337607332
        y: 0.005621354355423784 4
        z: -0.013281498322848109
        w: 0.6015522808182666
  -
    header:
      seq: 0
      stamp:
        secs: 1679302032
        nsecs: 889014208
      frame_id: "left"
    child_frame_id: "left_fisheye"
    transform:
      translation:
        x: 0.0753340587618107
        y: -0.005158899200750513
        z: 0.004368142951384089
      rotation:
        x: -0.003048791121646806
        y: -0.020996990427553607
        z: 0.002514693270834218
        w: 0.9997717277376049
  -
    header:
      seq: 0
      stamp:
        secs: 1679302032
        nsecs: 888725708
      frame_id: "right"
    child_frame_id: "right_fisheye"
    transform:
      translation:
        x: 0.0743283221971791
        y: -0.0033740900414505923
        z: -0.00036247378805597495
      rotation:
        x: -0.004343771269245429
        y: 0.008378798004595334
        z: 0.0013095968884968211
        w: 0.9999546051452277
```

```
-
  header:
    seq: 0
    stamp:
      secs: 1679302032
      nsecs: 888725708
    frame_id: "head"
  child_frame_id: "right"
  transform:
    translation:
      x: -0.1665919503641412
      y: -0.10984267045277182
      z: 0.03495716058308293
    rotation:
      x: 0.7939795031519209
      y: -0.015558628460717615
      z: -0.00224339791148875
      w: 0.6077412647014031
```

D.3 RTAB-Map publishers

Table D.1: RTAB-Map output publishers [51]

Topic	**Type**	**Description**
info	rtabmap_ros/Info	RTAB-Map's info.
mapData	rtabmap_ros/MapData	RTAB-Map's graph and latest node data.
mapGraph	rtabmap_ros/MapGraph	RTAB-Map's graph only.
grid_map	nav_msgs/OccupancyGrid	Occupancy grid generated with laser scans
cloud_map	sensor_msgs/PointCloud2	3D point cloud generated from the assembled local grids
cloud_obstacles	sensor_msgs/PointCloud2	3D point cloud of obstacles generated from the assembled local grids
cloud_ground	sensor_msgs/PointCloud2	3D point cloud of ground generated from the assembled local grids.
scan_map	sensor_msgs/PointCloud2	3D point cloud generated with the 2D scans or 3D scans
labels	visualization_msgs/MarkerArray	Convenient way to show graph's labels in RVIZ.
global_path	nav_msgs/Path	Poses of the planned global path. Published only once for each path planned.
local_path	nav_msgs/Path	Upcoming local poses corresponding to those of the global path. Published on every map update.
goal_reached	std_msgs/Bool	Status message if the goal is successfully reached or not.
goal_out	geometry_msgs/PoseStamped	Current metric goal sent from the rtabmap's topological planner.
octomap_full	octomap_msgs/Octomap	Get an OctoMap. Available only with octomap.
octomap_binary	octomap_msgs/Octomap	Get an OctoMap. Available only with octomap.
octomap_occupied_space	sensor_msgs/PointCloud2	A point cloud of the occupied space of the OctoMap. Available only with octomap.
octomap_obstacles	sensor_msgs/PointCloud2	A point cloud of the obstacles of the OctoMap. Available only with octomap.
octomap_ground	sensor_msgs/PointCloud2	A point cloud of the ground of the OctoMap. Available only with octomap.
octomap_empty_space	sensor_msgs/PointCloud2	A point cloud of empty space of the OctoMap. Available only with octomap.
octomap_grid	nav_msgs/OccupancyGrid	The projection of the OctoMap into a 2D occupancy grid map. Available with octomap.

D.4 RTAB-Map subscriptions

Table D.2: RTAB-Map subscriptions

Topic	**Type**	**Description**
/spot/odometry	nav_msgs/Odometry	Robot odometry data
/spot/camera/frontleft/image	sensor_msgs/Image	Mono image data
/spot/camera/frontleft/camera_info	sensor_msgs/CameraInfo	Camera metadata
/spot/camera/frontright/image	sensor_msgs/Image	Mono image data
/spot/camera/frontright/camera_info	sensor_msgs/CameraInfo	Camera metadata
/spot/camera/left/image	sensor_msgs/Image	Mono image data
/spot/camera/left/camera_info	sensor_msgs/CameraInfo	Camera metadata
/spot/camera/right/image	sensor_msgs/Image	Mono image data
/spot/camera/right/camera_info	sensor_msgs/CameraInfo	Camera metadata
/spot/camera/back/image	sensor_msgs/Image	Mono image data
/spot/camera/back/camera_info	sensor_msgs/CameraInfo	Camera metadata
/spot/depth/frontleft/image	sensor_msgs/Image	Depth image data
/spot/depth/frontright/image	sensor_msgs/Image	Depth image data
/spot/depth/left/image	sensor_msgs/Image	Depth image data
/spot/depth/right/image	sensor_msgs/Image	Depth image data
/spot/depth/back/image	sensor_msgs/Image	Depth image data
/spot/lidar/points	sensor_msgs/PointCloud2	Laser scan point cloud stream
/frontleft/rgbd_image	rtabmap_ros/RGBDImage	Synchronised RGBD image from Nodelets
/frontright/rgbd_image	rtabmap_ros/RGBDImage	Synchronised RGBD image from Nodelets
/left/rgbd_image	rtabmap_ros/RGBDImage	Synchronised RGBD image from Nodelets
/right/rgbd_image	rtabmap_ros/RGBDImage	Synchronised RGBD image from Nodelets
/back/rgbd_image	rtabmap_ros/RGBDImage	Synchronised RGBD image from Nodelets
/tag_detections	apriltag_ros/AprilTagDetectionArray	Apply AprilTag constraints
/imu	sensor_msgs/Imu	[Unused] Apply gravity constraints
/gps/fix	sensor_msgs/NavSatFix	[Unused] Apply GPS constraints
/global_pose	geometry_msgs/PoseWithCovarianceStamped	[Unused] Apply global prior constraints
/fiducial_transforms	fiducial_msgs/FiducialTransformArray	[Unused] Apply fiducial constraints
/goal	geometry_msgs/PoseStamped	Plan a path to reach this goal using the current online map
/initialpose	geometry_msgs/PoseStamped	Set initial pose of the robot
/tf	tf2_msgs/TFMessage	Reference frame transforms
/tf_static	tf2_msgs/TFMessage	Reference frame transforms

D.5 RTAB-Map services

Table D.3: RTAB-Map services

Topic	**Type**	**Description**
get_map	nav_msgs/GetMap	Get the standard 2D occupancy grid
get_map_data	rtabmap_ros/GetMap	Get the RTAB-Map map data
publish_map	rtabmap_ros/PublishMap	Call this service to publish the map data
list_labels	rtabmap_ros/ListLabels	Get current labels of the graph
update_parameters	std_srvs/Empty	The node will update with current parameters of the rosparam server
reset	std_srvs/Empty	Delete the map
pause	std_srvs/Empty	Pause mapping
resume	std_srvs/Empty	Resume mapping
trigger_new_map	std_srvs/Empty	Begin a new map
backup	std_srvs/Empty	Backup the database to `~/.ros/rtabmap.db.back`
set_mode_localization	std_srvs/Empty	Set localization mode
set_mode_mapping	std_srvs/Empty	Set mapping mode
set_label	rtabmap_ros/SetLabel	Set a label to latest node or a specified node
set_goal	rtabmap_ros/SetGoal	Set a goal to plan a path to, navigate with move_base
octomap_full	octomap_msgs/GetOctomap	Get an Octomap
octomap_binary	octomap_msgs/GetOctomap	Get an Octomap

E GraphNav Door Opening ROS Node

```python
#!/usr/bin/env python3
import typing
import pickle
import time

import rospy
import actionlib
from actionlib_msgs.msg import GoalStatus
from rosservice import get_service_class_by_name
from spot_msgs.srv import (
    ArmJointMovement,
    ArmJointMovementResponse,
    ArmJointMovementRequest,
)
from spot_msgs.msg import (
    TrajectoryAction,
    TrajectoryResult,
    TrajectoryFeedback,
    TrajectoryGoal,
)

from std_msgs.msg import String
from geometry_msgs.msg import PoseStamped
from spot_msgs.msg import AprilTagProperties, WorldObjectArray,
    WorldObject
from spot_msgs.msg import FrameTreeSnapshot, ParentEdge
from spot_msgs.srv import NavigateInitRequest, NavigateInitResponse
from spot_msgs.msg import NavigateToAction, NavigateToGoal
from spot_msgs.srv import ListGraphResponse
from std_srvs.srv import TriggerRequest

from fiducial import Fiducial


class SpotDoorDemo:
    def __init__(self):
        self.world_objects = None
        self.fiducials_seen: typing.Dict[int, typing.List["Fiducial"]] =
            {}
        self.waypoint_fiducial: typing.Dict[str, typing.List["Fiducial"]]
            = {}

    def initialize_subscribers(self):
        """Initialize ROS subscribers"""
        pass

    def initialize_publishers(self):
        """Initialize ROS publishers"""
        self.reached_waypoint_pub = rospy.Publisher(
            "/spot/nav/reached_waypoint", String, queue_size=1
        )

    def initialize_action_clients(self):
        """Initialize ROS action clients"""
        # Create an action client for the /spot/navigate_to action
        self.navigate_to_client = actionlib.SimpleActionClient(
            "/spot/navigate_to", NavigateToAction
        )

    def call_service(self, service_name: str, *args, **kwargs):
        """Call a service and wait for it to be available"""
        try:
            rospy.wait_for_service(service_name)
            service_type = get_service_class_by_name(service_name)
            proxy = rospy.ServiceProxy(service_name, service_type)
            return proxy(*args, **kwargs)

        except rospy.ServiceException as e:
            rospy.logerr("Service call failed: %s" % e)

    def walk_current_graph(self):
        """Walk the current graph in GraphNav"""
        rospy.loginfo("Walking the current graph")

        # Call the /spot/navigate_init service
        req = NavigateInitRequest(
            upload_path="/home/ming/Desktop/catkin_ws/test_data/door_demo
               ",
            initial_localization_fiducial=True,
            initial_localization_waypoint="np",
        )
        resp = self.call_service("/spot/navigate_init", req)

        # Check if the action succeeded
        if resp.success:
            rospy.loginfo(resp.message)

        # Call the ListGraph service
        list_graph: ListGraphResponse = self.call_service("/spot/
           list_graph")
        waypoints = list_graph.waypoint_ids
        start, end = waypoints[0], waypoints[-1]

        navigate_to_goal = NavigateToGoal(navigate_to=end)
        self.navigate_to_client.send_goal(navigate_to_goal)
        self.navigate_to_client.wait_for_result()

        # Check if the action succeeded
        if self.navigate_to_client.get_state() == GoalStatus.SUCCEEDED:
            rospy.loginfo(f"NavigateTo {end} action succeeded")

        # Press button
        self.move_arm()

        # Go back to start point
```

```python
        navigate_to_goal = NavigateToGoal(navigate_to=start)
        self.navigate_to_client.send_goal(navigate_to_goal)
        self.navigate_to_client.wait_for_result()

    def extract_fiducials(self, msg: WorldObjectArray) -> typing.List["Fiducial"]:
        self.world_objects: typing.List[WorldObject] = msg.world_objects
        fiducial_list = []

        # Iterate through the fiducials in the message, append the x,y,z coordinates to a dictionary
        for world_object in self.world_objects:
            april_tag: AprilTagProperties = world_object.apriltag_properties
            latest_snapshot: FrameTreeSnapshot = world_object.frame_tree_snapshot

            # Check if april_tag is None
            if april_tag is None:
                continue

            # Create the FrameTreeSnapshot as a dictionary
            frame_tree_snapshot: typing.Dict[str, PoseStamped] = {}
            for child, parent_edge in zip(
                latest_snapshot.child_edges, latest_snapshot.parent_edges
            ):
                parent_edge_transform = PoseStamped()
                parent_edge_transform.header.stamp = world_object.acquisition_time
                parent_edge_transform.header.frame_id = parent_edge.parent_frame_name
                parent_edge_transform.pose = parent_edge.parent_tform_child

                frame_tree_snapshot[child] = parent_edge_transform

            # Use tf2 to get the april_tag pose in the body frame
            april_tag_pose = frame_tree_snapshot[f"fiducial_{april_tag.tag_id}"]
            april_tag_pose_filtered = frame_tree_snapshot[
                f"filtered_fiducial_{april_tag.tag_id}"
            ]

            # Build the april_tag into the Fiducial class
            fiducial = Fiducial(
                tag_id=april_tag.tag_id,
                dim_x=april_tag.x,
                dim_y=april_tag.y,
                fiducial_pose=april_tag_pose,
                filtered_fiducial_pose=april_tag_pose_filtered,
                pose_covariance=april_tag.detection_covariance,
                pose_covariance_frame=april_tag.detection_covariance_reference_frame,
            )

            # Append the fiducial to the list
            fiducial_list.append(fiducial)

        return fiducial_list

    def move_arm(self):
        latest_world_objects_msg = rospy.wait_for_message(
            "/spot/world_objects", WorldObjectArray
        )
        fiducials = self.extract_fiducials(latest_world_objects_msg)
        with open(f"end_fiducial.pickle_{time.time()}", "wb") as f:
            pickle.dump(fiducials, f)

        angle = 1.6
        req = ArmJointMovementRequest(joint_target=[-0.2, -angle, angle,
            0.0, 0.0, 0.0])
        self.call_service("/spot/arm_carry", TriggerRequest())
        self.call_service("/spot/arm_joint_move", req)

        time.sleep(1)
        self.call_service("/spot/arm_stow", TriggerRequest())

    def startup(self):
        rospy.loginfo("SpotNav robot starting up")

        # Call the /spot/claim, /spot/power_on, /spot/stand service
        self.call_service("/spot/claim", TriggerRequest())
        self.call_service("/spot/power_on", TriggerRequest())

    def shutdown(self):
        rospy.loginfo("SpotNav node shutting down")

        # Save the fiducials to a pickle file
        with open(f"fiducials_seen.pickle_{time.time()}", "wb") as f:
            pickle.dump(self.fiducials_seen, f)
        with open(f"waypoint_fiducial.pickle_{time.time()}", "wb") as f:
            pickle.dump(self.waypoint_fiducial, f)

        # Call the /spot/sit, /spot/power_off, /spot/release service
        self.call_service("/spot/sit", TriggerRequest())
        time.sleep(5)
        self.call_service("/spot/power_off", TriggerRequest())
        self.call_service("/spot/release", TriggerRequest())

    def main(self):
        rospy.init_node("spot_nav", anonymous=True)

        self.rates = rospy.get_param("~rates", {})
        if "loop_frequency" in self.rates:
            loop_rate = self.rates["loop_frequency"]
        else:
            loop_rate = 50

        # Initialize the node
        rate = rospy.Rate(loop_rate)
        rospy.loginfo("SpotNav node started")

        self.initialize_subscribers()
        self.initialize_publishers()
        self.initialize_action_clients()

        rospy.on_shutdown(self.shutdown)

        # Walk the current graph
        self.startup()
        for i in range(3):
            self.walk_current_graph()

        self.call_service("/spot/sit", TriggerRequest())
        time.sleep(5)
        self.call_service("/spot/power_off", TriggerRequest())
        self.call_service("/spot/release", TriggerRequest())

        while not rospy.is_shutdown():
            rate.sleep()


if __name__ == "__main__":
    spot_nav = SpotDoorDemo()
    spot_nav.main()
```