FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

NavHost - Navigation and Path Planning for a Host Robot Operating in Large Institutions

António Pedro de Andrade Pereira e Oliveira Martins



Mestrado em Engenharia Eletrotécnica e de Computadores

Orientador: Prof. Armando Jorge Miranda de Sousa

October 30, 2026

© António Pedro de Andrade Pereira e Oliveira Martins , 2026

NavHost - Navigation and Path Planning for a Host Robot Operating in Large Institutions

António Pedro de Andrade Pereira e Oliveira Martins

Mestrado em Engenharia Eletrotécnica e de Computadores

October 30, 2026

Abstract

Large public institutions such as hospitals and university campuses are challenging environments for mobile service robots. A host robot must guide visitors across long distances, traverse multiple floors using elevators or ramps, and, in some cases, pass through outdoor segments between building entrances, all while operating safely among people and within an existing, heterogeneous software stack. This dissertation addresses the design and implementation of a navigation layer for such a host robot, built on a wheelchair-derived electric platform and an existing ROS 1 navigation baseline.

The proposed navigation layer extends the existing ROS 1 `move_base`-based navigation system with three main capabilities. First, it introduces a waypoint-graph mission layer that represents the building as a sparse graph of semantically meaningful locations, including doorways, corridor junctions, and elevator lobbies. Long missions are therefore decomposed into ordered sequences of intermediate navigation goals. Second, the system supports multi-map and multi-floor operation through an explicit `MapSwitcher` component, which coordinates map loading, indoor AMCL or GNSS-aided localization, and costmap resets whenever a mission crosses from one floor or area to another. Third, it adds a hierarchical velocity override mechanism beneath the navigation stack, enabling the lower control layer to impose immediate stops and controlled rotations in dangerous situations or when guest tracking is lost, without modifying the planners or the mission logic.

The perception pipeline combines a front-mounted RP-LiDAR with two OAK-D stereo depth cameras, installed at the front and rear of the robot. The depth data from each OAK-D are converted into LaserScan-compatible observations and merged with the RP-LiDAR scan into a common range representation. This representation is used by the localization pipeline and by the global and local costmaps of the navigation stack, extending obstacle coverage beyond the single planar LiDAR scanning plane.

The navigation layer is evaluated in Gazebo using FEUP-derived indoor and outdoor environments. Single-floor experiments show that the system executes corridor missions reliably, with path lengths close to the nominal waypoint-graph distance and velocity profiles compatible with smooth and comfortable motion on the wheelchair-derived platform. Multi-floor missions via a simulated elevator demonstrate that the waypoint mission orchestrator and `MapSwitcher` correctly handle repeated map transitions, allowing navigation to resume on the target floor after the appropriate map and localization configuration have been activated.

An outdoor experiment evaluates GNSS-aided localization over an approximately 478 m route. Conventional AMCL exhibited persistent localization drift in this scenario because the large outdoor map contained comparatively sparse geometric features for scan matching. The GNSS-aided `gps_amcl` configuration provided an additional absolute-position constraint and enabled stable route execution. The indoor and outdoor experiments were repeated five times in the simulated FEUP environments, with all evaluated missions completing successfully.

Finally, override experiments confirm that the velocity override node can temporarily take control of the `cmd_vel` stream to stop or rotate the robot and subsequently return control to

`move_base` without destabilising the active mission.

Taken together, these results indicate that NavHost, the navigation and path planning layer of the Host Robot, meets its core design goals: it enables waypoint-based, multi-map, multi-floor, and indoor–outdoor navigation with a safety-aware control mechanism, while integrating complementary LiDAR and stereo-depth sensing into the existing ROS 1 navigation architecture.

Keywords: mobile robot navigation; host robot; waypoint-based navigation; multi-map navigation; multi-floor navigation; indoor–outdoor localization; GNSS-aided localization; LiDAR; stereo depth cameras; ROS 1; ROS Navigation Stack; wheelchair-derived robot platform.

Resumo

Grandes instituições públicas, como hospitais e campus universitários, constituem ambientes exigentes para robôs móveis de serviço. Um robô anfitrião deve ser capaz de orientar visitantes ao longo de percursos extensos, atravessar vários pisos através de elevadores ou rampas e, em alguns casos, circular por segmentos exteriores entre entradas de diferentes edifícios. Estas tarefas devem ser executadas de forma segura na presença de pessoas e no contexto de uma arquitetura de software heterogénea já existente. Esta dissertação aborda o desenvolvimento e a implementação de uma camada de navegação para um robô anfitrião construído sobre uma plataforma elétrica derivada de uma cadeira de rodas e baseado numa infraestrutura de navegação ROS 1 existente.

A camada de navegação proposta estende o sistema de navegação existente, baseado em ROS 1 e em `move_base`, através de três capacidades principais. Em primeiro lugar, introduz uma camada de missão baseada num grafo de *waypoints*, que representa o edifício através de locais semanticamente relevantes, tais como portas, cruzamentos de corredores e zonas de acesso a elevadores. Desta forma, missões de longa distância são decompostas numa sequência ordenada de objetivos de navegação intermédios. Em segundo lugar, o sistema suporta operação com múltiplos mapas e múltiplos pisos através do componente `MapSwitcher`, responsável por coordenar o carregamento de mapas, a seleção do método de localização interior baseado em AMCL ou de localização assistida por GNSS, e a atualização das *costmaps* sempre que uma missão transita entre pisos ou áreas distintas. Em terceiro lugar, é introduzido um mecanismo hierárquico de controlo de velocidade, designado por *velocity override*, que permite à camada de controlo de baixo nível impor paragens imediatas e rotações controladas em situações de perigo ou quando se perde o acompanhamento do visitante, sem alterar os planeadores ou a lógica de missão.

A cadeia de perceção combina um RP-LiDAR montado na parte frontal do robô com duas câmaras estéreo OAK-D, instaladas nas partes frontal e traseira da plataforma. Os dados de profundidade obtidos por cada câmara OAK-D são convertidos em observações compatíveis com a mensagem `LaserScan` e posteriormente combinados com o varrimento do RP-LiDAR numa representação comum de distâncias. Esta representação é utilizada pela cadeia de localização e pelas *costmaps* globais e locais da pilha de navegação, permitindo aumentar a cobertura de obstáculos para além do plano único de varrimento do LiDAR.

A camada de navegação foi avaliada em Gazebo, utilizando ambientes interiores e exteriores derivados de plantas da FEUP. Os ensaios de navegação num único piso demonstram que o sistema executa missões em corredores de forma consistente, com percursos próximos da distância nominal definida pelo grafo de *waypoints* e perfis de velocidade compatíveis com um movimento suave e confortável na plataforma derivada de cadeira de rodas. Os ensaios multi-piso, realizados através de um elevador simulado, demonstram que o orquestrador de missões e o `MapSwitcher` coordenam corretamente transições repetidas entre mapas, permitindo retomar a navegação no piso de destino depois de ser ativada a configuração adequada de mapa e localização.

Foi igualmente avaliada a navegação exterior com localização assistida por GNSS, ao longo de um percurso de aproximadamente 478 m. Nesta situação, a utilização de AMCL convencional

revelou deriva de localização persistente, devido à grande dimensão do mapa exterior e à menor disponibilidade de elementos geométricos distintivos para correspondência de varrimentos. A configuração `gps_amcl`, que incorpora informação GNSS no processo de localização, forneceu uma restrição adicional de posição absoluta e permitiu a execução estável do percurso. Os ensaios interiores e exteriores foram repetidos cinco vezes nos ambientes simulados da FEUP, tendo todas as missões avaliadas sido concluídas com sucesso.

Por fim, os ensaios do mecanismo de *velocity override* confirmam que este pode assumir temporariamente o controlo do fluxo `cmd_vel`, parar ou rodar o robô e devolver posteriormente o controlo a `move_base`, sem desestabilizar a missão ativa.

Em conjunto, os resultados demonstram que o NavHost, a camada de navegação e planeamento de percursos do Host Robot, cumpre os seus principais objetivos: suportar navegação baseada em *waypoints*, múltiplos mapas, múltiplos pisos e transições entre ambientes interiores e exteriores, integrando simultaneamente um mecanismo de controlo orientado para a segurança e a fusão de informação proveniente de LiDAR e de câmaras estéreo de profundidade numa arquitetura de navegação ROS 1 existente.

Palavras-chave: navegação de robôs móveis; robô anfitrião; navegação baseada em *waypoints*; navegação multi-mapa; navegação multi-piso; localização interior–exterior; localização assistida por GNSS; LiDAR; câmaras estéreo de profundidade; ROS 1; pilha de navegação ROS; plataforma robótica derivada de cadeira de rodas.

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework for achieving a better and more sustainable future for all. They comprise 17 goals[1] that address challenges such as poverty, inequality, climate change, environmental degradation, peace, and justice. Although this dissertation is primarily technical, the Host Robot navigation layer it proposes is intended for deployment in large public institutions, and therefore connects directly to how those institutions organise and deliver services to people.

The main application domains envisioned for the Host Robot are large institutions like hospitals and university campuses. In both settings, the robot acts as a mobile host, guiding visitors between entrances, reception areas, consultation rooms, lecture halls, and exits. By enabling reliable, multi-floor navigation on a platform based on a wheelchair chassis, the navigation layer helps make complex buildings easier to understand and traverse, particularly for users who are unfamiliar with the site or who benefit from clearer guidance and reduced cognitive load. Within the SDG framework, this work is most closely related to:

SDG 3 Ensure healthy lives and promote well-being for all at all ages.

SDG 8 Promote sustained, inclusive and sustainable economic growth, full and productive employment and decent work for all.

SDG 10 Reduce inequality within and among countries.

SDG 3 is addressed through improved access and reduced stress in healthcare and educational facilities: a host robot that can reliably navigate at institutional scale can help patients, students and visitors reach appointments and services more easily and free clinical, administrative and support staff from routine escort and wayfinding duties. SDG 8 is touched through incremental automation of low-value, repetitive transport and guidance tasks and the creation of new technical roles around robot deployment, configuration and supervision, rather than direct substitution of staff. SDG 10 is supported by the explicit focus on robust, elevator-based routes and clear wayfinding on a platform derived from an accessible mobility base, which can lower information and orientation barriers for visitors who currently find large buildings difficult to navigate without assistance.

The qualitative contributions to these SDGs can be summarised as follows:

[1] https://sdgs.un.org/goals

SDG	Target	Contribution	Performance Indicators and Metrics
3	3.8	Reliable robot-guided wayfinding in hospitals and clinics can make services easier to reach, reducing missed or delayed appointments caused by navigation difficulties and lowering anxiety for first-time visitors.	Change in average time for visitors to reach consultation rooms; decrease in missed or late arrivals attributable to navigation issues; survey scores on perceived ease of navigation and clarity of guidance.
	3.c	Automating routine escort and guidance tasks allows clinical and administrative staff to devote more time to core healthcare activities.	Estimated staff time saved per week from reduced escort and wayfinding duties; number of missions handled autonomously by the Host Robot per day.
8	8.2	Deploying host robots based on an open navigation layer supports innovation in service delivery and creates new technical roles in configuration, monitoring, and maintenance, rather than replacing staff outright.	Number of robots deployed using the NavHost navigation layer; number of staff trained in robot operation and supervision; reduction in time spent on repetitive wayfinding support.
10	10.2	Providing a consistent navigation service throughout the building helps visitors who are less familiar with the institution, or who face language or informational barriers, move independently between services.	Proportion of missions initiated by visitors reporting difficulties with wayfinding; user satisfaction scores regarding accessibility of information and perceived independence.
	10.3	A standardised host-robot navigation service reduces disparities between regular users and occasional visitors in how easily they can find and access services inside large buildings.	Survey-based measures of perceived fairness and accessibility; fraction of visitors who report being able to reach destinations without needing human escort.

In summary, the navigation layer developed in this dissertation contributes to SDG 3, SDG 8, and SDG 10 by enabling host robots that make large institutions more navigable, reduce routine workload on staff, and lower access and information barriers for visitors who currently face the greatest difficulty moving through these spaces.

Acknowledgements

I would first like to express my deepest gratitude to my supervisor, Professor Armando Jorge Miranda de Sousa, whose guidance and support were fundamental throughout this project. His availability, patience, and willingness to go far beyond what is usually expected of a supervisor made a definite impact in the work presented and to my own learning. For all the time he devoted to discussing ideas, clarifying doubts, helping me navigate setbacks, and always doing so with good humour, I am sincerely thankful.

I am also grateful to my colleagues on the Host Robot project, Daniel, António, and Catarina. Although each of us was responsible for different parts of the system, their collaboration, feedback, and constant willingness to help shaped many of the ideas and solutions that appear in this dissertation.

To my friends who have accompanied me on this academic journey, in particular Miguel, Inês, Mariana, and António, thank you for your support, encouragement, and for all the moments of sanity along the way. I am no less thankful to the friends with whom my academic paths diverged after high school, especially David, Tomás, Luís, and Guilherme, whose friendship and belief in me have also contributed to bringing me here.

I owe a special debt of gratitude to my family — my mother, father, and brother — for their unconditional support, understanding, and encouragement throughout these years.

Finally, to Spencer, thank you for always being by my side and for the patience, care, and motivation you have given me throughout this process.

To all of you, thank you.

António Pedro de Andrade Pereira e Oliveira Martins

Contents

1 Introduction **1**
- 1.1 Motivation and Context . . . 1
  - 1.1.1 Application Scenario . . . 1
  - 1.1.2 Existing Solutions and Limitations . . . 2
- 1.2 Problem Statement and Objectives . . . 2
- 1.3 Scope, Limitations, and Contributions . . . 3
- 1.4 Dissertation Structure . . . 4

2 State of the Art and Problem Deconstruction **5**
- 2.1 Chapter Overview . . . 5
- 2.2 Problem Deconstruction . . . 5
  - 2.2.1 The Host Robot Navigation Problem . . . 5
  - 2.2.2 Decomposition of the Navigation Problem . . . 6
- 2.3 Intellwheels as a Starting Point . . . 7
  - 2.3.1 Repository and Platform Overview . . . 7
  - 2.3.2 Baseline Navigation and Capability Gaps . . . 8
- 2.4 Path Planning and Navigation Stacks . . . 8
  - 2.4.1 Global Planning Approaches . . . 8
  - 2.4.2 Local Planning and Reactive Trajectory Control . . . 9
  - 2.4.3 Integrated Navigation Stacks (e.g., `move_base`) . . . 9
- 2.5 Route Planning in Large Multi-Floor Institutions . . . 10
  - 2.5.1 Navigation Tree / Graph Over the Whole Building . . . 10
  - 2.5.2 2D-Per-Floor Maps with Staircase / Elevator Connectors . . . 10
  - 2.5.3 Integrated Multifloor Map . . . 11
  - 2.5.4 Hierarchical Topological–Metric Planners for Complex Indoor Spaces . . . 12
- 2.6 Robotic Middleware and Layered Architectures . . . 12
  - 2.6.1 Middleware Paradigms for Robotic Software . . . 12
  - 2.6.2 ROS1 and ROS2 in Distributed Robot Systems . . . 13
  - 2.6.3 Interoperability Between ROS Generations . . . 13
- 2.7 Localization in Mobile Service Robots . . . 14
  - 2.7.1 Odometry and Inertial-Based Estimation . . . 14
  - 2.7.2 Probabilistic Localization (Particle Filters, AMCL) . . . 14
  - 2.7.3 Outdoor and GNSS-Based Localization . . . 15
  - 2.7.4 Multi-Sensor Fusion Approaches . . . 15
- 2.8 Perception and Dynamic Obstacle Handling . . . 16
  - 2.8.1 Range-Sensing Modalities (LiDAR and Depth Cameras) . . . 16
  - 2.8.2 Depth Sensing and Conversion to Planar Laser Scans . . . 17
  - 2.8.3 Multi-Sensor Data Fusion for Obstacle Detection . . . 18

2.9 Comparable Guide and Host Robot Systems . . . 19
2.9.1 Indoor Guide Robots in Public Institutions . . . 19
2.9.2 Commercial Service Robots with Navigation . . . 20
2.9.3 Open and ROS-Based Host Platforms . . . 20
2.9.4 Positioning of the Host Robot . . . 21
2.10 Gap Analysis and Positioning of This Work . . . 21
2.10.1 Capability Gaps Relative to the State of the Art . . . 22
2.10.2 Relation to Navigation Requirements . . . 22
2.10.3 Scope and Contributions of This Dissertation . . . 23
2.11 Summary . . . 24

3 Solution Architecture and Implementation 25
3.1 Chapter Overview . . . 25
3.2 Navigation Problem and Requirements . . . 25
3.2.1 Functional Requirements (FR) . . . 26
3.2.2 Non-Functional Requirements (NFR) . . . 27
3.3 Navigation Layer Architecture . . . 27
3.3.1 Intellwheels Navigation Baseline . . . 27
3.3.2 Physical Platform and Onboard Hardware . . . 28
3.3.3 Proposed Architecture . . . 35
3.3.4 Communication with Other Robot Layers . . . 39
3.4 Core Navigation Stack . . . 40
3.4.1 `move_base` as Navigation Engine . . . 41
3.4.2 Integration Points in the Architecture . . . 41
3.4.3 Indoor–Outdoor Localization Strategy . . . 42
3.5 Custom Navigation Components . . . 44
3.5.1 Waypoint structure and set up . . . 44
3.5.2 Waypoint Mission Orchestrator (`move_base_handler`) . . . 47
3.5.3 Front-Distance Estimation Node . . . 51
3.5.4 Map Server Handler (`MapSwitcher`) . . . 52
3.5.5 Velocity Override Node . . . 54
3.5.6 Depth Image to LaserScan Conversion . . . 55
3.6 Navigation Execution Flows . . . 58
3.6.1 Waypoint Mission Orchestrator Flow . . . 58
3.6.2 Map Switching Sequence . . . 68
3.6.3 Override node Flow . . . 69
3.7 Implementation Overview . . . 71
3.7.1 ROS Graph and Deployment . . . 71
3.7.2 Node Interfaces and Topics . . . 74
3.8 Design Rationale and Key Decisions . . . 76
3.8.1 Architectural Rationale . . . 76
3.8.2 Key Design Decisions . . . 76
3.9 Chapter Summary . . . 78

4 Results and Discussion 79
4.1 Chapter Overview . . . 79
4.2 Experimental Methodology . . . 79
4.2.1 Test Environments . . . 79
4.2.2 Metrics and Data Collection . . . 81

4.3 Single-Floor Navigation Experiments . . . 83
4.3.1 Experimental Setup . . . 83
4.3.2 Quantitative Results . . . 84
4.3.3 Discussion . . . 85
4.4 Multi-Floor Navigation via Elevator . . . 86
4.4.1 Experimental Setup . . . 86
4.4.2 Quantitative Results . . . 89
4.4.3 Discussion . . . 90
4.5 Outdoor Navigation with GNSS-Aided Localization . . . 91
4.5.1 Experimental Setup . . . 91
4.5.2 Quantitative Results . . . 92
4.5.3 Discussion . . . 93
4.6 Override Module Validation . . . 94
4.6.1 Experimental Setup . . . 94
4.6.2 Observed Behaviour . . . 95
4.6.3 Discussion . . . 96
4.7 Cross-Scenario Analysis . . . 96
4.8 Summary . . . 97

5 Conclusions and Future Work **98**
5.1 Summary of Contributions . . . 98
5.2 Discussion of Results . . . 99
5.3 Limitations . . . 100
5.4 Future Work . . . 101
5.4.1 Deployment on the Physical Robot . . . 102
5.4.2 Additional Odometry Sources . . . 102
5.4.3 Expansion and Improvement of the Map Library . . . 102
5.4.4 Adaptive and Persistent Mission-Level capabilities . . . 103

References **105**

List of Figures

3.1 Corner view of the assembled Host Robot platform. 29
3.2 Front view of the Host Robot platform. 30
3.3 Rear view of the Host Robot platform, showing the upper electronics area and wheelchair-derived base. 31
3.4 Side view of the Host Robot, showing the wheelchair-derived base, upper structural frame, and mounted hardware. 32
3.5 Front OAK-D stereo depth camera mounted using a custom-designed support. . . 33
3.6 Rear OAK-D stereo depth camera installed on the Host Robot. 34
3.7 Onboard computing and electronic hardware based on an NVIDIA Jetson Nano (marked in red). 35
3.8 Overview of the proposed navigation-layer architecture. A destination is decomposed into an ordered waypoint sequence, whose individual goals are executed by the `move_base`-based navigation stack. Map and localization transitions are coordinated through the map server handler, while fused odometry and merged range data support localization and obstacle-aware navigation. The front-distance node provides a simplified forward-obstacle signal to the mission layer, and the velocity override node supports temporary guest-reacquisition manoeuvres before commands reach the execution layer. 36
3.9 Conceptual interaction between robot layers across ROS generations. The ROS 1 navigation and path planning layer exchanges final destination goals and status events with the ROS 2 decision and user interface layer, and velocity commands, GNSS and sensor data, and override commands with the ROS 2 execution and control layer via ROS 1–ROS 2 bridges. 39
3.10 Example waypoint layout on a single-floor map. Waypoints are placed at corridor junctions, on each side of doorways, and at elevator lobbies, with neighbour links forming a sparse navigation graph over the free space. 45
3.11 State machine of the `move_base_handler` waypoint mission orchestrator. The node alternates between waiting for a mission, planning a route over the topological graph, and executing the resulting waypoint sequence while handling map checks, retries, and pre-emption. 47
3.12 High-level execution flow of the waypoint mission orchestrator. A destination is converted into an A*-planned waypoint sequence, whose elements are dispatched individually to `move_base`. Successful waypoint execution may trigger a map-context check and a forward-path check, while persistent blockages and replacement goals cause the route to be re-planned. 59

3.13 Goal reception and topological route planning. The mission handler receives a `NavigationGoal` message, obtains the robot pose from TF, selects the closest graph nodes on the relevant maps, and uses A* to generate an ordered waypoint sequence. 61
3.14 Waypoint dispatch and `move_base` outcome handling. The execution state machine selects each route element, converts it into a `MoveBaseGoal`, monitors the action result, retries pre-empted goals, and cancels the active goal when a replacement destination is received. 63
3.15 Map and localization context check performed by the waypoint mission orchestrator. After a successful waypoint, the handler compares the active map and localization mode with those required by the following waypoint and requests `/switch_map` when a transition is needed. 65
3.16 Forward-path handling and topological re-planning. The handler monitors the latest `/front_distance` value, waits for temporary obstructions to clear, and removes the blocked graph connection when the configured timeout expires. A* then computes a replacement route over the updated in-memory topology. 67
3.17 Map switching sequence coordinated by `MapSwitcher`. A requested world change triggers map loading, localization restart, initial pose publication, localization confirmation, and a costmap reset before control returns to the mission orchestrator. . 68
3.18 Velocity override sequence. In normal operation, `cmd_vel` messages from `move_base` are forwarded unchanged. When an override mode is active, the node ignores incoming planner commands and publishes stop or rotation commands at a fixed rate. 70
3.19 ROS graph of the navigation-layer implementation in simulation. The graph shows the interaction between the Gazebo simulation, waypoint mission handler, navigation stack, GNSS-aided localization, map-switching components, range-scan merger, and velocity override node. 72

4.1 First-floor FEUP occupancy map used in the experiments. White regions correspond to free space, while black regions represent walls and structural obstacles. . 80
4.2 Basement-floor FEUP occupancy map used in the experiments, including the lower corridor and elevator area used for multi-floor missions. 80
4.3 Single-floor experiment: initial pose A and goal B on the first-floor FEUP map used in simulation. 83
4.4 Multi-floor mission used in the elevator experiments. The upper map corresponds to the basement floor and the lower map to the first floor. Red markers indicate the initial and final mission waypoints, while blue markers show the elevator connection points used to teleport the robot between floors. A representation of the selected trajectory is shown with a red line 87
4.5 Outdoor FEUP environment used in the GNSS-aided navigation experiment. The start and end positions are shown in red, while the blue markers represent waypoints placed in the topological map for this area. 91
4.6 Outdoor FEUP route used in the GNSS-aided navigation experiment. The start and end positions are shown in red, while the blue markers represent the intermediate navigation waypoints. The orange line shows the trajectory selected and executed by the navigation system through these waypoints. 92

List of Tables

3.1 Main ROS nodes and topics in the navigation-layer implementation. 74

4.1 Summary of mission- and command-level metrics for the single-floor A-to-B navigation experiment (five runs). 85
4.2 Summary of mission- and command-level metrics for the outdoor navigation experiment using GNSS-aided localization. 93

List of Acronyms

ROS Robot Operating System

ROS1 Robot Operating System version 1

ROS2 Robot Operating System version 2

AMCL Adaptive Monte Carlo Localization

GNSS Global Navigation Satellite System

GPS Global Positioning System

RTK Real-Time Kinematic

DDS Data Distribution Service

LiDAR Light Detection and Ranging

IMU Inertial Measurement Unit

UWB Ultra-Wideband

EKF Extended Kalman Filter

UKF Unscented Kalman Filter

TEB Timed Elastic Band

DWA Dynamic Window Approach

FEUP Faculdade de Engenharia da Universidade do Porto

Chapter 1

Introduction

1.1 Motivation and Context

Large public institutions such as hospitals, universities, and research centres regularly receive visitors who are unfamiliar with the environment and need to move between entrances, reception areas, consultation rooms, offices, and meeting spaces. In such settings, providing effective guidance is not only a matter of helping visitors reach their destination, but also of ensuring that movement through the institution is safe, efficient, and well integrated with the normal activity of the building.

Service robots have increasingly been proposed as assistants in these environments, combining information services with mobility [1], [2], [3], [4]. However, the usefulness of such systems depends to a large extent on the quality of their navigation capabilities. A robot may be able to communicate with users or receive destination requests, but those abilities are of limited value if it cannot move reliably through a large institution, react to dynamic obstacles, and complete missions that extend beyond a single floor or a single map [5], [6], [7].

This dissertation is motivated by the need for a navigation layer that can support that kind of operation. In the target scenario, the robot must guide visitors through a large institutional environment composed of several floors and selected outdoor areas, requiring the navigation system to handle long routes, floor changes, and transitions between indoor and outdoor localization conditions [7], [8]. The focus of the work is therefore not the entire robotic host system, but the navigation layer that enables high-level mission requests to be converted into safe and feasible motion under realistic deployment constraints [9], [10], [11].

1.1.1 Application Scenario

The application scenario considered in this dissertation is a large institution in which visitors may need to be guided between semantically meaningful locations such as entrances, reception desks, consultation rooms, lecture rooms, elevators, and exits. Unlike standard single-goal navigation tasks, these missions may span long distances, involve several intermediate stops, and require

movement across multiple floors and, in some cases, through outdoor segments connecting different access points of the site [3], [7], [12], [13].

Within the broader Host Robot project, this dissertation concerns the middle functional layer of the system. The low-level layer is responsible for the physical platform and sensors, while the high-level layer handles interaction with the guest and determines the desired destination. The role of the middle layer is to ensure that navigation to that destination is executed as smoothly and safely as possible, avoiding static architectural constraints and dynamic obstacles while operating on a converted electric wheelchair base and within an existing ROS1/ROS2 software architecture [9], [10], [11], [14].

This context makes the navigation problem more demanding than conventional single-floor, single-map navigation. The system must represent routes at the scale of the building, coordinate multiple maps, use suitable localization methods indoors and outdoors, and provide clear progress and failure feedback to adjacent layers of the robot software [7], [8], [15], [16].

1.1.2 Existing Solutions and Limitations

Navigation is one of the most fundamental capabilities of mobile service robots, comprising mapping, localization, path planning, and obstacle avoidance [5], [6]. In common ROS-based systems such as `move_base`, navigation is usually structured around two planning levels: a global planner that computes a route over the map and a local planner that reacts in real time to sensor data and dynamic obstacles while tracking the global path as closely as possible [9]. This architecture is effective for many indoor navigation tasks and provides a strong baseline for practical deployments.

At the same time, most established solutions are primarily designed for navigation on a single map or within a limited indoor area [6], [9]. They do not directly address how to structure long-range missions across multiple floors, how to coordinate transitions between floor-specific or outdoor maps, or how to combine indoor map-relative localization with outdoor GNSS-based localization under a single navigation interface [8], [15], [16]. Research literature provides partial answers to these issues through building-wide waypoint graphs, hierarchical topological–metric planners, multi-floor route representations, and sensor-fusion approaches for localization, but these capabilities are often treated separately rather than as parts of one coherent navigation layer [7], [17], [18].

For the present work, this creates a practical need rather than a purely theoretical one. Starting from an existing ROS1 baseline built on a wheelchair-derived platform, the challenge is not to replace standard navigation stacks altogether, but to extend them with the mechanisms required for multi-floor, multi-map, and indoor–outdoor mission execution in a large institution [9], [10], [11], [14]. This need motivates the navigation-layer focus adopted throughout the dissertation.

1.2 Problem Statement and Objectives

The problem addressed in this dissertation is the design and implementation of a navigation layer that allows a service robot to execute long-range missions in a large institutional environment that

includes multiple floors and selected outdoor areas [7], [8]. More specifically, the navigation layer must accept high-level destination or mission requests from an upper layer and convert them into safe, feasible, and executable motion while operating over several maps, handling localization changes, and reacting to dynamic obstacles [5], [10], [16].

This problem can be stated as follows:

> *Given a mobile service robot operating in a large, multi-floor institution with both indoor and outdoor areas, and an existing ROS-based navigation baseline, how can a navigation layer be designed so that it can execute waypoint-based missions across multiple maps and localization regimes while maintaining safe and reliable motion in dynamic environments?*

To address this problem, the dissertation pursues the following objectives:

- **O1: Structure long-range missions for navigation execution.** Define a route representation that allows high-level missions to be decomposed into a sequence of manageable navigation goals associated with meaningful locations in the environment [3], [7], [12], [13], [18].
- **O2: Support multi-map and multi-floor navigation.** Develop a mechanism for coordinating navigation across different floor maps and outdoor maps, including the transitions required when the mission moves from one area to another [7], [13], [15], [18].
- **O3: Integrate indoor and outdoor localization.** Provide a navigation-layer interface that can operate with map-relative localization in indoor areas and GNSS-aided localization in outdoor areas, so that pose estimation remains usable throughout the mission [7], [8], [16].
- **O4: Preserve safe motion within the full software stack.** Ensure that the navigation layer interfaces correctly with higher and lower system layers, including the ability to provide status feedback upward and to support safety-related command handling toward the low-level controller [10], [11].

These objectives do not assume that the navigation layer must solve every navigation problem from scratch. Instead, they reflect the practical need to organise and extend existing navigation components so that they can support missions of greater spatial scale and operational complexity than those addressed by standard single-map navigation alone [7], [9], [16].

1.3 Scope, Limitations, and Contributions

This dissertation is limited to the navigation layer of the overall system. It does not address the complete mechanical design of the robot, the detailed design of the wheelchair chassis, or the conversational and social interaction capabilities handled by the other layers of the Host Robot

project. Hardware components and mounting arrangements are described only to the extent required to explain their relationship with the navigation layer.

The work also adopts several practical constraints. It assumes a pre-mapped institutional environment whose structural layout is known in advance, even though the robot must still deal with dynamic obstacles during operation [5], [9]. It builds on an existing ROS1 navigation baseline rather than replacing it with a completely new stack, and considers outdoor operation only in the limited sense relevant to site access and movement between nearby institutional areas, not full outdoor autonomous navigation at city or campus scale [8], [11].

Within this scope, the contribution of the dissertation lies in the specification and implementation of a navigation layer suited to multi-floor and indoor–outdoor mission execution. In particular, the work contributes a mission-level navigation structure based on meaningful waypoints, a mechanism for managing map transitions, an approach to integrating indoor and outdoor localization within the same navigation workflow, and the interfaces needed for safe interaction with the surrounding software layers [7], [10], [11], [15], [16]. The contribution is therefore an application-driven navigation solution tailored to the needs of large institutional environments, rather than a claim of a new general navigation paradigm.

1.4 Dissertation Structure

The remainder of this dissertation is organised as follows. Chapter 2 presents the state of the art and problem deconstruction, surveying the main areas relevant to the navigation layer, including robotic middleware, environment representation, route structure, path planning, localization, perception, and comparable guide and host robot systems [1], [3], [4], [5], [6], [7], [10], [11], [17], [18]. This chapter also identifies the practical gaps that motivate the requirements of the proposed navigation layer.

Chapter 3 derives the functional and non-functional requirements of the system and presents the navigation-layer architecture proposed in this work as well as describing the implementation of the main components, including mission handling, multi-map coordination, localization integration, and the interfaces to adjacent layers of the robot software. Chapter 4 evaluates the behaviour of the implemented system in representative scenarios involving multi-floor and indoor–outdoor navigation. Finally, Chapter 5 discusses the main findings, the limitations of the work, and directions for future development.

Chapter 2

State of the Art and Problem Deconstruction

2.1 Chapter Overview

This chapter positions the navigation problem addressed in this dissertation within the broader state of the art and decomposes it into the specific subproblems that the proposed solution must tackle. It begins by characterising the Host Robot navigation problem in large, multi-floor institutions and outlining a set of navigation-related subproblems that structure the remainder of the chapter. It then introduces Intellwheels as the practical starting point for this work, describing the existing ROS1 wheelchair-based platform and clarifying which capabilities it already provides and which remain open. The following sections survey path planning and integrated navigation stacks, multi-floor and multi-area route representations, robotic middleware and ROS1/ROS2 interoperability, localisation across indoor and outdoor areas, and perception and dynamic obstacle handling, with particular emphasis on systems comparable to a host or guide robot. The chapter concludes by synthesising these strands into a gap analysis that motivates the requirements and architecture developed in Chapter 3.

2.2 Problem Deconstruction

2.2.1 The Host Robot Navigation Problem

Navigation is one of the most fundamental capabilities of mobile service robots, allowing them to move safely and autonomously within human environments [5]. It encompasses several interdependent processes, including localization, mapping, path planning, and obstacle avoidance. Together, these functions enable a robot to perceive its surroundings, determine its position, plan a feasible route, and adapt its motion in real time to ensure safety and efficiency [5], [6].

For the Host Robot, this general problem is instantiated in a large public institution with multiple floors and both indoor and outdoor areas. The robot must guide visitors between semantically meaningful locations such as reception desks, consultation rooms, elevators, and entrances, while

operating on the chassis and actuation capabilities of an electric wheelchair platform. From the decision layer's perspective, navigation requests arrive as high-level missions (for example, "guide this user from the main entrance to Radiology, then to the exit"), which the navigation layer must transform into a continuous stream of low-level velocity commands that respect the platform's kinematic limits and remain safe and comfortable for nearby people [18].

Several aspects make this navigation problem more demanding than standard single-floor, single-map scenarios. Missions can span long distances and multiple floors, require the use of elevators or ramps, and traverse selected outdoor segments where GNSS is available but map-relative localization is not [7]. The robot must remain robust to varying crowd density and dynamic obstacles, provide meaningful progress and fault feedback to higher layers, and do so within an existing ROS1/ROS2-based software architecture that constrains how navigation components can be organised and deployed [5], [10]. These characteristics motivate a more explicit decomposition of the navigation problem into subproblems that can be treated separately in the remainder of this chapter.

2.2.2 Decomposition of the Navigation Problem

For the purposes of this dissertation, the Host Robot navigation problem is decomposed into a set of interrelated subproblems, each of which is addressed by one or more of the following sections.

- **Robotic middleware and layering.** How navigation components discover each other and exchange data, and how the navigation layer interfaces with the decision and low-level control layers, given a mixed ROS1/ROS2 deployment and the need to balance flexible reconfiguration with predictable timing on safety-critical paths (Section 2.6) [10].
- **Environment representation and route structure.** How the large, multi-floor, partially outdoor environment is represented for navigation purposes, including whether to use single or multiple maps, how to encode vertical connectors such as elevators and stairs, and how to structure long missions as sequences of waypoints or nodes in a building-wide graph (Section 2.5) [7], [18].
- **Path planning and navigation stacks.** How to compute and execute feasible, collision-free paths between goals on a given map, including the choice of global and local planners, their composition within an integrated navigation stack such as `move_base`, and their suitability for a wheelchair-derived base (Section 2.4) [5], [6].
- **Localization across indoor and outdoor areas.** How to estimate the robot's pose reliably throughout the mission by combining odometry and inertial sensing with map-relative localization indoors and GNSS-aided localization outdoors, and how to fuse and switch between these sources in a way that the rest of the stack can treat as a single pose estimate (Section 2.7) [18].

- **Perception and dynamic obstacle handling.** How to sense and represent obstacles at the spatial and temporal resolutions the planners require, using LiDAR and/or depth cameras, and how to maintain layered costmaps or equivalent structures that allow safe motion among people in dynamic environments (Section 2.8) [5].
- **Mission-level navigation and comparable systems.** How long-range, waypoint-based missions are specified, sequenced, and monitored over the underlying planners and maps, and how similar guide and host robot systems have addressed or left open these issues (Section 2.9) [3], [7], [12].

This decomposition serves two purposes. First, it provides a structure for the remainder of this chapter, which surveys the state of the art for each subproblem in turn. Second, it anticipates the functional and non-functional requirements formalised in Chapter 3, where these subproblems are recomposed into the concrete specification that the Host Robot navigation layer implemented in this work must satisfy.

2.3 Intellwheels as a Starting Point

Rather than designing a navigation system from a blank slate, this work builds on Intellwheels [14][1], an existing ROS1 navigation repository developed for a converted electrical wheelchair platform. Working from an already-functioning platform rather than starting from scratch is a common and practical choice in mobile robotics research more broadly: it avoids re-solving problems— robot description, basic sensor integration, a working simulation-to-hardware pipeline— that are necessary preconditions for navigation research but are not, by themselves, the contribution of that research [5]. What matters for positioning this work is less that Intellwheels exists, and more precisely what it does and does not already provide, since that is what determines which parts of the navigation problem set out in Section 2.2 still need to be addressed.

2.3.1 Repository and Platform Overview

Structurally, the Intellwheels repository is organized as a small set of ROS1 catkin packages, each responsible for one part of getting the wheelchair moving: a description package defining the robot's 3D model and publishing its joint and robot states; a Gazebo-based simulation package for navigating a simulated lab environment; a Stage-based simulation package for navigating lighter-weight, faster-to-run 2D simulated environments; and an "in real life" package for running navigation on the physical wheelchair itself (referred to in the repository as *jeeves*). This split between simulated and physical navigation packages sharing a common robot description is itself a standard pattern in ROS1 mobile robotics development, allowing planning and control behaviour to be tested in simulation before being deployed on hardware [9].

[1] https://github.com/siferati/intellwheels_nav

2.3.2 Baseline Navigation and Capability Gaps

What Intellwheels provides, in other words, is a validated *platform*—a correctly described robot, a working simulation environment, a functioning bridge from that simulation to the physical wheelchair, and navigation built on the standard ROS1 2D Navigation Stack (`move_base`, discussed in Section 2.4.3)—rather than a complete solution to the navigation problem this dissertation addresses [9]. It establishes that the wheelchair can be made to navigate at all, using standard, widely adopted ROS1 tooling, but does not by itself provide the additional capabilities a multi-area institutional host robot requires: navigating across multiple floors or maps, fusing indoor and outdoor localization sources, sequencing a long route into a series of manageable waypoints handed to `move_base` in turn, or exposing a safety-override path independent of the planner [7], [18]. Each of these gaps is the subject of one of the following sections, and each is discussed there against the relevant state of the art rather than against Intellwheels specifically—but it is Intellwheels, and the `move_base`-based navigation it already provides, that supplies the underlying robot, simulation, and baseline navigation infrastructure that all of that additional work is built on top of [9], [14].

2.4 Path Planning and Navigation Stacks

Path planning computes a route between the robot's current position and a target destination, balancing path length, smoothness, and safety against the robot's physical constraints—turning radius and maximum acceleration among them—to remain feasible for a converted wheelchair base to actually execute [5], [6].

Planning approaches are commonly split into *global* planning, computing a route ahead of time over a complete map, and *local* planning, operating over a short, continuously updated window that reacts to whatever the map does not capture—most importantly, people [5]. In this work, a third, higher tier sits above both: a waypoint-graph layer decomposes a long route into a sequence of shorter, pre-validated hops between known nodes in the building, each handed to the global/local planner pair in turn rather than planning the whole route at once. This kind of hierarchical, topological-over-metric structure is an established pattern for large, structured indoor environments, since it keeps each individual planning problem small and well-constrained [7], [18]—and, as the next two subsections show, it is that structure, more than raw map size, that determines which planning strategies are worth considering at all.

2.4.1 Global Planning Approaches

Global planners are generally grouped into graph-search methods (Dijkstra, A*), sampling-based methods (RRT, RRT*), and heuristic or bio-inspired methods (genetic algorithms, particle swarm and ant colony optimization). Broadly, graph-search methods are complete but scale poorly with map size and update frequency; sampling-based methods trade that completeness for efficiency in large, high-dimensional, or frequently changing spaces; and bio-inspired methods trade both for

the ability to balance several soft objectives in a single cost function, at the price of less predictable convergence [5], [6].

Sampling-based and bio-inspired methods earn their keep specifically in environments that are large, only partially known, or subject to frequent structural change—none of which describes a single hop of a pre-surveyed, static waypoint graph. Once that graph has already reduced the problem to short, well-constrained segments between known nodes, graph search's main weakness (poor scaling) is largely absent, while its main strength—a deterministic, complete path to an exact goal pose—is exactly what each hop needs, consistent with graph-search planners such as Dijkstra and A* remaining the most widely deployed choice in practical ROS-based systems generally [5], [9].

2.4.2 Local Planning and Reactive Trajectory Control

Local, reactive trajectory generation is likewise a broader space than any single algorithm: velocity-space sampling methods (the Dynamic Window Approach), trajectory-optimization methods (Timed Elastic Band, model-predictive control), and social-force-inspired methods that explicitly model surrounding agents' motion all address the same problem of turning a global path into feasible, moment-to-moment velocity commands, differing mainly in how far ahead they reason and how expensive that reasoning is [5], [6].

The Dynamic Window Approach (DWA) and Timed Elastic Band (TEB) are the two most directly relevant here. DWA samples achievable velocities given the robot's kinematic state, forward-simulates each briefly, and scores the results against obstacle and goal proximity, which is cheap to run and folds kinematic limits directly into the sampling step, but—by reasoning only a short horizon ahead—can settle into locally sensible but globally awkward motion such as oscillation near obstacles or in tight corridors [5]. TEB instead optimizes a sequence of timed poses against transit time, obstacle clearance, and kinodynamic feasibility jointly, and can compare multiple topologically distinct routes around an obstacle before committing to one [6]. It generally produces smoother, more explicitly feasible trajectories, including through narrow passages, at a higher per-cycle computational cost [5], [6]. This leaves the two on opposite sides of a fairly direct cost-versus-trajectory-quality trade-off, with the better choice for a given deployment depending on factors—corridor width, available controller-frequency headroom, and how heavily the local planner is relied on relative to the waypoint-graph and global planner above it—that are generally better resolved empirically on the target platform than assumed from the literature alone [5].

2.4.3 Integrated Navigation Stacks (e.g., `move_base`)

Global and local planners are rarely deployed on their own; they are composed within a broader stack managing costmaps, plugin interfaces, and failure recovery. Several such stacks exist across the ROS ecosystem—`move_base` for ROS1, its more modular successor `move_base_flex`, and Nav2 for ROS2—but `move_base` remains the most widely adopted for ROS1-based systems, and is the stack underlying the Intellwheels baseline extended in this work [9].

In stacks like `move_base`, two levels of planning run concurrently: a global plan, concerned mainly with pre-mapped obstacles, lays out the route from start to end, while a local plan looks only a few seconds ahead but is far more dependent on live sensor data, reacting to dynamic obstacles in real time while tracking the global plan as closely as possible [5], [9]. Both stages are exposed as pluggable interfaces (`nav_core::BaseGlobalPlanner` and `nav_core::BaseLocalPlanner`), which is what allows the planner pairing discussed above to be selected without altering the surrounding stack, and is also what makes it straightforward to drip-feed the stack a sequence of waypoint-graph goals one at a time rather than a single distant target—each hop is simply issued as an independent goal once the previous one completes [9], [18].

2.5 Route Planning in Large Multi-Floor Institutions

2.5.1 Navigation Tree / Graph Over the Whole Building

Navigation-tree or building-wide graph approaches model the institution as a set of discrete locations and connections rather than a single monolithic grid map. In this paradigm, each meaningful place (e.g., rooms, corridor junctions, elevator lobbies, service areas) becomes a node in a graph, and edges represent feasible traversable links between them, often annotated with costs such as distance, travel time, or elevator usage penalties. Global path planning is then reduced to a shortest-path problem (typically Dijkstra or A*) on this sparse graph, yielding an ordered sequence of nodes that describe the route at a human-interpretable level ("Room A $\rightarrow$ Corridor 1 $\rightarrow$ Elevator $\rightarrow$ Corridor 3 $\rightarrow$ Room B") [7], [13], [18].

Systems such as campus and building guide or delivery robots implement this concept by defining a static navigation tree spanning all floors and using Dijkstra's algorithm to compute optimal routes between arbitrary points, with elevator transitions encoded as edges in the tree [3], [12], [13]. This building-wide graph abstraction offers several advantages over pure metric planning when institutions are large and structurally complex. First, it dramatically reduces the search space for global path planning: instead of searching over thousands or millions of grid cells, the planner searches over tens or hundreds of graph nodes, enabling fast re-planning when destinations or intermediate waypoints change [13], [18]. Second, it naturally supports multi-floor navigation, since floors are just additional node clusters and vertical connections (stairs, elevators) become special edges that can carry additional semantic or accessibility information. Finally, because the graph deals in human-meaningful places, it forms a convenient interface between high-level goal selection and low-level local planning, which subsequently handles the detailed motion between consecutive graph nodes on each floor [7], [13], [15].

2.5.2 2D-Per-Floor Maps with Staircase / Elevator Connectors

Another approach to multi-storey navigation involves maintaining independent 2D maps for each floor and linking them via explicit vertical connectors such as elevators or staircases. In this

paradigm, global planning is decomposed into two stages: first, a high-level search selects a sequence of floors and the specific connector required to transit between them; second, a standard 2D path planner (such as A* or Dijkstra) generates the metric trajectory from the robot's current position to the connector on the starting floor, and subsequently from the connector exit to the destination on the target floor [15], [18]. This strategy effectively decouples the complexity of 3D space from the navigation logic, allowing the robot to leverage mature, robust 2D navigation stacks like the standard ROS `move_base` without the computational overhead of maintaining a fully volumetric 3D map [9].

At the same time, this decoupled approach introduces blind spots in optimization. By treating the vertical transition as a portal between disjoint 2D maps, the planner often fails to evaluate the true cost of the transition itself—ignoring factors such as elevator wait times, crowd density at landing areas, or the energy cost of ramp traversal [7], [15]. A purely topological link assumes that arriving at the elevator door is equivalent to arriving at the next floor, masking the complex temporal and social interactions required to actually board and travel. Compared to unified multifloor planners that model these costs explicitly, the 2D-per-floor method risks generating paths that are geometrically shortest but temporally inefficient or socially disruptive [15], [18].

2.5.3 Integrated Multifloor Map

Integrated multifloor mapping approaches represent the entire building—rooms, corridors, stairs, and elevators—within a single coherent 2.5D or 3D map, instead of splitting navigation into separate 2D floor plans [17]. In these systems, vertical structures such as staircases and elevator shafts are explicitly modelled as traversable volumes or annotated connectors embedded in the same global map that encodes horizontal free space [17], [18]. The planner can then operate directly on this unified structure using graph search or hybrid metric–topological methods, reasoning over paths that mix horizontal and vertical movement in a single optimization problem rather than composing multiple independent 2D plans with ad hoc switching logic at floor boundaries [7], [13], [17].

An advantage of integrated multifloor maps is that they can attach more accurate costs to vertical transitions, such as elevator waiting time, energy expenditure on stairs, or accessibility constraints, and treat these costs on the same footing as horizontal travel distance [7], [13]. However, this increased sensibility comes at the price of higher mapping and planning complexity. Building and maintaining a consistent multifloor map requires reliable height estimation and cross-floor registration, and searching in 3D can be computationally heavier than searching per-floor graphs. Compared to 2D-per-floor plus connector schemes, integrated multifloor maps offer more globally optimal and refined plans, but are more demanding in terms of sensing, map management, and planner complexity [13], [17], [18].

2.5.4 Hierarchical Topological–Metric Planners for Complex Indoor Spaces

Hierarchical topological–metric planners address the challenge of navigating large, complex indoor environments by decomposing the environment into multiple abstraction levels, each with its own representation and planning algorithm [17], [18]. At the lowest level, a detailed metric map (for example, an occupancy grid) captures fine geometric structure and supports local path generation and obstacle avoidance. Above this, intermediate levels group free-space cells into larger regions like rooms, corridors, or zones, connected by bridge nodes that act as portals between regions, forming a sparse topological graph [13], [18]. At the highest level, planning is performed on this graph using graph-search algorithms such as A* or D*, producing a coarse route that specifies a sequence of regions or portals the robot must traverse rather than a full-resolution path [17].

Constructing these hierarchies from floor plans or occupancy grids can be automated, yielding hierarchical graphs that scale to very large environments such as factory halls or campus buildings [7], [13]. Such methods achieve near-optimal paths with much lower computational effort than flat A* on the full map, while preserving a clear separation between high-level route selection and low-level motion. They do, however, introduce approximations in the abstraction process, trading some representational simplicity for scalability and interpretability [17], [18].

2.6 Robotic Middleware and Layered Architectures

2.6.1 Middleware Paradigms for Robotic Software

Robotic middleware generally follows one of a few communication paradigms—client–server request/response, message-queue-based communication, and publish/subscribe—differing in how tightly they couple producers and consumers of data in time and in identity [11]. Publish/subscribe has become the dominant paradigm for general-purpose robotic middleware, since it lets a component publish data to a named topic without needing to know who, if anyone, is listening, and lets that set of listeners change at runtime without disrupting the publisher [11].

This is the property that matters most for the Host Robot's navigation layer, where several components—localization, map management, path planning, perception, execution—must be addable, restartable, or reconfigurable independently, without the rest of the stack needing to be aware that anything changed underneath it [11]. The trade-off is that publish/subscribe communication is asynchronous by construction, with no built-in guarantee about when a subscriber sees a given message relative to when it was published—an acceptable cost for most of the stack, but more relevant for a safety-critical override path where a stop or turn-in-place command needs to reach the base with low, predictable latency rather than best-effort delivery [10]. That tension between flexible communication and timing predictability is one of the motivations for ROS2's DDS-based design, discussed next [10].

2.6.2 ROS1 and ROS2 in Distributed Robot Systems

ROS1, the middleware underlying the Intellwheels baseline this work builds on, centres around a `roscore` master for node discovery, with publishers and subscribers then connecting directly over a custom TCP/UDP transport (TCPROS/UDPROS) [10]. This has proven sufficient for a large body of ROS1-based navigation research, but has two well-documented properties: the master is a single point of failure for discovery, and the transport offers no configurable Quality of Service (QoS), so a component cannot request stronger delivery or timing guarantees on a topic that needs them [10].

ROS2 replaces both with the Data Distribution Service (DDS) standard: discovery is decentralised, with no equivalent single point of failure, and every topic can carry its own QoS policy governing reliability, history, and timing [10]. In principle this makes ROS2 the better fit for a timing-sensitive path like a safety override, but that benefit only accrues to components actually running on ROS2—and the navigation baseline extended here (Intellwheels, and `move_base` itself) is ROS1-native, with no first-party ROS2 port of `move_base` to migrate to (its ROS2 successor, Nav2, is a distinct codebase rather than an upgrade path) [9]. ROS2 documentation and comparative evaluations also note that DDS introduces its own serialization and discovery overhead relative to ROS1's simpler transport, so the choice between them is a trade-off rather than a strict improvement in either direction [10].

2.6.3 Interoperability Between ROS Generations

Because ROS1 and ROS2 nodes cannot communicate directly, systems spanning both generations generally take one of two approaches: full migration, rewriting affected ROS1 nodes as native ROS2 nodes, or bridging, running both generations side by side and translating specific topics and services between them at runtime without modifying either system internally [10].

Migration gives unrestricted access to ROS2's discovery and QoS features but requires re-implementing and re-validating every affected component, complicated further by mature ROS1 packages that have not been, and in some cases will not be, ported to ROS2. For a navigation stack like `move_base` and the broader Intellwheels baseline—already tuned and validated for the target platform—that cost is substantial, since it is not a translation exercise so much as re-verifying planner and controller behaviour from scratch on a different codebase [9]. Bridging, implemented for ROS1/ROS2 pairs by the `ros1_bridge` package (dynamically, auto-detecting matching topics, or statically, via an explicit list), avoids this by letting the validated ROS1 baseline keep running unmodified while still allowing newer, ROS2-based components elsewhere in the system to participate, at the cost of the extra latency and configuration overhead the bridge itself introduces at the translation points [10]. For a system extending an already-functioning ROS1 codebase rather than building one from scratch, this incremental approach is therefore the one generally recommended in practice, precisely because it avoids forcing a full re-validation of components that already work [10].

2.7 Localization in Mobile Service Robots

Localization refers to the robot's ability to determine its position within a known environment. Accurate localization ensures that the robot can understand where it is relative to its surroundings, using information from odometry, inertial sensors, LiDAR, or visual observations [5].

None of these methods is sufficient on its own across an entire institutional building and its grounds—some estimate motion rather than absolute position, some need a pre-built map, some need an open sky. The following subsections survey the landscape of options for each of these narrower sub-problems, then look more closely at the specific approaches whose trade-offs matter most for a robot that has to operate across both indoor and outdoor areas of a single site [5], [7].

2.7.1 Odometry and Inertial-Based Estimation

Proprioceptive motion estimation spans wheel odometry (integrating encoder counts), inertial estimation (integrating IMU angular velocity and acceleration), visual and visual-inertial odometry (tracking feature displacement across camera frames, optionally fused with inertial data), and LiDAR-inertial variants that extend the same principle to scan-matching-derived motion [5], [16]. All of them share the same structural weakness: each estimates a change in pose rather than an absolute pose, so error compounds without bound the longer the robot moves without an external correction [5].

Wheel and inertial estimation matter most here, since together they typically form the continuous, high-rate backbone that fusion-based localization relies on between absolute corrections. Wheel odometry is cheap and self-contained, but slip and surface irregularity become a permanent offset rather than a transient error, with drift compounding fastest during rotation; IMU integration is similarly self-contained and higher-frequency, but position requires double-integrating acceleration, so bias and noise compound quadratically, and IMU-only position becomes unusable within seconds [5]. Visual and LiDAR-inertial odometry generally drift more slowly than either, which is their appeal, but that comes at the cost of dependence on scene texture or lighting and substantially higher per-estimate computation—a cost more easily absorbed by a well-resourced platform than assumed by default [5], [16].

2.7.2 Probabilistic Localization (Particle Filters, AMCL)

Localizing against a known map is itself an active area with several families of solutions— Kalman-filter-based map-relative localization, scan-matching and feature/landmark-based methods, graph /pose-optimization approaches, and particle-filter-based methods such as Monte Carlo Localization — differing mainly in how they represent and update the robot's pose hypothesis as new sensor data arrives [5].

Adaptive Monte Carlo Localization (AMCL) is the most directly relevant of these, being the standard choice in ROS-based systems: it represents pose as a set of weighted particles, each a hypothesis, which are reweighted and resampled against incoming laser scans and motion data

until the set converges on the true pose relative to a fixed map [5], [9]. Because every correction is anchored to the map rather than the previous estimate, AMCL does not drift the way proprioceptive methods do—but it depends entirely on that map's quality and on the environment being visually distinguishable enough for the filter to converge, degrading in long, symmetric corridors where many poses look equally plausible, and scaling in computational cost with the number of particles needed for a large or cluttered map [5]. More fundamentally, it has nothing to offer in an unmapped area, or in an outdoor setting where a laser-scan-matchable occupancy grid is not a natural representation to begin with [5], [7].

2.7.3 Outdoor and GNSS-Based Localization

Outdoor localization without a prior map draws on a different set of approaches— standalone GNSS, differential and carrier-phase-corrected GNSS (RTK), and infrastructure- or sensor-aided alternatives such as UWB or LiDAR-aided GNSS—most of which trade cost and setup complexity for positioning accuracy [8], [16].

GNSS is the default among these because it needs no local infrastructure or map at all; standard positioning is typically only accurate to several metres, while RTK-corrected GNSS can reach centimetre-level accuracy under right conditions by using carrier-phase measurements against a reference station [8]. That accuracy is conditional on largely unobstructed sky visibility, however: near or between buildings, multipath propagation and non-line-of-sight reception degrade positioning significantly, and RTK is particularly sensitive to this since its accuracy assumes a maintained carrier-phase fix—exactly what gets interrupted by the signal blockage and reflection common around large structures [8]. The result is a method whose strengths and weaknesses mirror AMCL's almost exactly—no map dependency, but accuracy dependent on conditions the robot cannot control rather than on a map it can build in advance—which is why indoor and outdoor localization tend to be treated as related but distinct problems rather than expecting one method to cover both [5], [7].

2.7.4 Multi-Sensor Fusion Approaches

Combining proprioceptive and absolute localization sources is itself a design space: the Kalman filter family (EKF, UKF), particle-filter-based fusion, and factor-graph/pose-graph optimization approaches all address the same underlying problem—reconciling a continuous but drifting estimate with an intermittent but drift-free one—with different assumptions about noise, nonlinearity, and computational budget [5], [19].

The Extended Kalman Filter (EKF) is the most common choice in practice, recursively combining measurements with different noise characteristics into a single state estimate, and underlies widely used ROS fusion tools such as `robot_localization` [19]. Its main limitation is relying on local linearization of what is often a nonlinear motion model; the Unscented Kalman Filter (UKF) avoids this by propagating sample points through the true nonlinear model instead, at higher computational cost, and is generally preferred when motion or noise departs further from

the EKF's assumptions [19]. For a robot whose absolute-position source changes depending on where it is on-site—map-relative indoors, GNSS-relative outdoors—this fusion layer's role is less about smoothing a single fixed sensor set and more about providing the rest of the navigation stack a consistent estimator interface regardless of which absolute source is currently active [5], [7], [8].

2.8 Perception and Dynamic Obstacle Handling

Obstacle avoidance ensures that the robot can move safely even when unexpected objects or people enter its path. In structured environments, this can rely on pre-existing maps, but in dynamic human environments real-time sensing becomes critical: the robot must continuously process sensor data, identify potential collisions, and modify its trajectory accordingly [5], [6]. Perception for navigation is therefore less about recognising objects in detail and more about maintaining a conservative, up-to-date estimate of free and occupied space at the spatial and temporal scales the local planner requires [5].

2.8.1 Range-Sensing Modalities (LiDAR and Depth Cameras)

Several sensing modalities can support obstacle detection for mobile robots. Planar 2D LiDAR has long been the default in indoor navigation, providing accurate range measurements over a single scan plane at high frequency, which integrate naturally into occupancy grids and scan-matching localisers [5]. Its advantages include geometric precision, robustness to lighting conditions, and mature algorithmic support; its main limitation is that it only "sees" obstacles intersecting the chosen scan plane, which can miss overhanging objects or low elements below the scan height [5], [6].

Depth cameras (RGB-D or time-of-flight) offer a different trade-off: they deliver dense depth images over a field of view, potentially capturing obstacles both above and below a single laser plane and, when paired with RGB, supporting higher-level semantic perception [5]. They are, however, more sensitive to lighting and reflective surfaces, and their output is computationally heavier to process, especially on platforms without powerful GPUs [5], [6]. Outdoors, depth cameras often degrade more severely than LiDAR due to sunlight interference, whereas LiDAR suffers less from illumination but can be more expensive and mechanically demanding to mount on compact bases such as converted wheelchairs [5].

Ultrasonic sensors and bumpers form a third class of options aimed at low-cost platforms: they provide very local obstacle detection but poor angular resolution and limited range, which makes them ill-suited as the sole source of obstacle data in cluttered human environments, though they can usefully complement other sensors as last-resort safety checks [5]. Vision-only approaches, relying on monocular or stereo cameras to infer free space, avoid dedicated range hardware but require substantial computation and are far more sensitive to texture and lighting, making them less attractive as the primary obstacle sensor in mixed indoor and semi-outdoor environments [5], [6].

2.8.2 Depth Sensing and Conversion to Planar Laser Scans

Stereo depth cameras and planar LiDAR sensors represent complementary approaches to obstacle perception for mobile robots. A planar LiDAR provides direct range measurements in a single horizontal plane, typically with high angular resolution and reliable performance over a wide range of distances. In contrast, stereo depth cameras produce dense two-dimensional depth images that encode distance information over both the horizontal and vertical dimensions. This makes depth cameras particularly useful for detecting obstacles that may not intersect the height of a LiDAR scanning plane, including objects located above or below the primary laser scan.

Stereo depth is obtained by observing the same scene from two spatially separated cameras and estimating the disparity between corresponding image features. Depth is inversely related to disparity and depends on the focal length and stereo baseline. The resulting depth map is sensitive to image texture, illumination, surface reflectivity, and calibration quality. Consequently, stereo depth tends to perform well in textured, well-lit environments, but can degrade on textureless surfaces, under strong specular reflections, or in low-light conditions [20].

Many existing navigation stacks, including widely used ROS-based frameworks, are designed around planar range data in the form of a `sensor_msgs/LaserScan` message. To integrate stereo depth cameras into such systems, a common approach is to convert depth images into a synthetic laser scan. The ROS `depthimage_to_laserscan` package exemplifies this strategy: it receives a depth image and the associated camera calibration data, reconstructs a three-dimensional point cloud, and projects it into a two-dimensional scan suitable for navigation and localization [21]. The resulting message contains angular limits, angular resolution, valid range limits, and one range value for each angular bin, as specified by the ROS LaserScan message definition [22].

The conversion process typically follows several conceptual steps. First, each valid depth pixel is transformed into a three-dimensional point in the camera frame using the intrinsic camera parameters. The point cloud is then transformed from the camera frame into a common robot frame, using the known calibration between the camera and the robot base. This transformation is important because multiple cameras, if used, generally do not share the same sensor origin or field of view. Expressing all measurements in a common frame allows them to be interpreted consistently by the navigation system.

The three-dimensional points are subsequently projected onto the horizontal plane to produce a planar scan. For each point, the planar range and bearing are calculated, and the point is assigned to the corresponding angular bin. When several points fall into the same bin, the closest valid measurement is retained, since it represents the first obstacle encountered along that direction. Invalid depth values, measurements outside the configured range, and points that belong to the ground plane are typically removed before this projection. Filtering the ground is particularly important for mobile platforms because the floor would otherwise generate a dense collection of short range returns and could be incorrectly interpreted as an obstacle.

This approach enables depth cameras to be used as additional range sources alongside conventional planar LiDAR. In multi-sensor configurations, the scans produced by one or more depth cameras can be merged with the output of a physical LiDAR into a unified range representation. The fused scan can then be consumed by localization algorithms such as AMCL and by the costmaps used by `move_base`, without requiring a separate navigation implementation for each sensor type. This preserves compatibility with established two-dimensional navigation architectures while benefiting from the three-dimensional sensing capability of depth cameras.

However, converting depth images to laser scans does not make stereo cameras equivalent to physical planar LiDARs. The camera-based scans inherit the limitations of stereo depth, including missing measurements on textureless or highly reflective surfaces, sensitivity to lighting conditions, and reduced reliability at the extremes of the operating range. In addition, the converted scan represents only the camera fields of view rather than a complete 360° observation. These limitations are typically managed by retaining a physical LiDAR as an independent range source and by merging multiple sensors only after each has been transformed into a common robot coordinate frame.

From a system-design perspective, the use of depth-image-to-laser-scan conversion can be viewed as a practical compromise between richer three-dimensional perception and the established two-dimensional interfaces used by many mobile-robot navigation stacks. It allows platforms to augment a conventional planar LiDAR with additional vertical coverage and complementary fields of view, while continuing to use mature navigation components that expect a single LaserScan input. This pattern is particularly relevant for platforms that must operate in complex indoor environments, where obstacles may appear at various heights and where the robot must navigate safely despite the limitations of any single sensing modality.

2.8.3 Multi-Sensor Data Fusion for Obstacle Detection

Once suitable sensors are chosen, the question becomes how to fuse their outputs into a single representation that planners can use. One broad approach is to project all obstacle detections into a common 2D occupancy grid or costmap, essentially taking a union of obstacles from all sensors and leaving temporal smoothing and clearing to simple rules such as ray tracing and time-based decay [5]. This is the model adopted by standard ROS navigation stacks, where a layered costmap combines a static map with one or more obstacle layers fed by laser scans or virtual scans derived from depth images [9]. Its strengths are strong tool support and predictable interaction with widely used planners; its main drawback is expressiveness, since inherently 3D obstacles and semantics are flattened onto a 2D grid [5].

A second family of approaches maintains richer, often 3D, representations—such as voxel grids or signed-distance fields—and projects them down only when needed for specific planners or tasks [5], [6]. These methods can represent complex environments more accurately and support 3D motion planning and explicit reasoning about overhanging obstacles, but they are correspondingly heavier in memory and computation and less directly compatible with legacy 2D navigation

stacks. For large service robots operating primarily on flat floors, the practical benefit of full volumetric obstacle maps often does not justify their complexity, unless they also support tasks beyond navigation, such as manipulation [5].

Dynamic obstacles add another layer of design choices. A minimalist strategy is to treat all detections as anonymous obstacles and let the local planner react purely based on short-term geometry, relying on frequent sensor updates and conservative inflation to maintain safety in crowded spaces [5]. More sophisticated systems apply explicit human detection and tracking, assigning identities and motion predictions to people and using these to bias costmaps or trajectory optimisers towards socially acceptable motion, for example by modelling personal space or predicting crossing trajectories [7], [18]. While the latter can improve comfort and predictability, it introduces additional perception failure modes (false positives/negatives in detection and tracking) and computational load, and is more common in research prototypes than in deployed building-scale guide robots [5], [18].

Implications for the Host Robot

Taken together, these trade-offs suggest that, for a wheelchair-based host robot built on a 2D costmap and `move_base` stack, a conservative, geometry-first perception design is the most practical choice: LiDAR or LiDAR-equivalent range data provide the backbone for obstacle detection, possibly augmented by depth sensors in regions critical for human comfort, while sensor fusion remains within the 2D layered costmap framework and treats most moving agents as generic dynamic obstacles [5], [9]. Any human-aware refinements can then be introduced at the planning and mission layers rather than by significantly complicating the perception stack itself [18].

2.9 Comparable Guide and Host Robot Systems

The Host Robot envisioned in this work is not the first platform to guide people through public spaces. Museum tour guides, hospital transport robots, commercial service robots in hotels and shopping malls, and research prototypes deployed in university buildings all combine autonomous navigation with some form of human interaction [2], [3], [4], [12]. What distinguishes them is less the existence of a navigation stack than how deeply that stack is integrated with building infrastructure, how far it must travel, and how openly its design is described. This section highlights representative systems along that spectrum and positions the Host Robot relative to them.

2.9.1 Indoor Guide Robots in Public Institutions

Early museum and campus guide robots such as RHINO, Minerva, and later campus tour platforms demonstrated continuous autonomous tours in crowded exhibition spaces and university buildings, navigating using 2D occupancy maps and laser-based localisation while interacting with visitors through speech and on-screen content [3], [12]. Their routes were largely confined to a single floor, with carefully curated exhibit areas and staff-supervised deployment. Similar systems have since

appeared in science centres and corporate lobbies, often as temporary demonstrations rather than long-term infrastructure [3]. In most published accounts, the navigation component resembles the standard pattern outlined in Sections 2.4 and 2.7: a global planner over a static map, a reactive local planner for dynamic obstacles, and map-relative localisation, all tuned to a specific building [5], [9].

Hospital and clinic robots form a related class, typically focused on transporting supplies or guiding patients to consultation rooms [1], [2], [4]. They operate in more constrained but also more regulated spaces than museums, with stricter safety and availability requirements. Several reported systems integrate directly with elevator controllers or use dedicated robot lifts, allowing multi-floor operation but at the cost of site-specific engineering and limited portability to other buildings without repeating that integration effort [7], [18]. Where navigation is described, it is again predominantly 2D, with multi-floor routing realised as a combination of per-floor maps and hard-coded elevator use rather than a generic building-wide representation [18].

2.9.2 Commercial Service Robots with Navigation

Commercial service robots such as hotel delivery and shopping-mall guides combine a socially expressive body or screen with an embedded navigation stack that is typically customised per deployment site [1], [4]. Vendors seldom disclose full technical details, but available descriptions indicate a common pattern: an initial mapping and configuration phase to create a 2D map of the site, followed by daily operation in which the robot moves among a fixed set of named locations (rooms, kiosks, reception desks) using a proprietary global/local planner pair conceptually similar to those surveyed in Section 2.4 [5], [6]. Where multi-floor capability is offered, it is generally tied to specific supported elevators and requires per-building installation and testing, making it a product feature rather than a reusable architectural pattern [7].

These commercial systems demonstrate that robust navigation in populated public spaces is commercially viable, but they are closed platforms: their mapping formats, planning algorithms, and interface to building infrastructure are not openly available [1]. As a result, they serve more as evidence that such systems can be built than as directly reusable designs for constructing a research host robot on an open ROS-based stack [3], [12].

2.9.3 Open and ROS-Based Host Platforms

Closer to the present work are ROS-based host and guide robots described in the academic literature [3], [12]. These systems typically adopt the standard ROS navigation stack (`move_base` or its ROS2 successor Nav2), together with AMCL and layered costmaps, and often add a semantic layer mapping named places or regions in the building to navigation goals [9]. Some use topological graphs of corridors and rooms to simplify long-range planning on large floor plans, or to provide human-readable route descriptions alongside navigation [7], [18]. However, most reported platforms operate either within a single floor or treat multi-floor operation as a special case, with

elevator and stair traversal handled by bespoke scripts or operator intervention rather than as part of a general-purpose, multi-map navigation architecture [7].

Outdoor-capable service robots on ROS exist as well, particularly in campus-delivery and last-metre logistics contexts [3], [6]. These platforms emphasise GNSS-based localisation and navigation on outdoor maps, sometimes with limited indoor docking capability, but they rarely address the mixed indoor–outdoor, multi-floor missions that a host robot for a large institution must support [8], [18].

2.9.4 Positioning of the Host Robot

Within this landscape, the Host Robot targeted in this dissertation fills a specific gap. Like museum and hospital guide robots, it must move safely among people in structured indoor spaces, but it must also:

- traverse multiple floors and selected outdoor segments of a single site using a combination of per-floor maps and GNSS-aided localisation;
- run on a wheelchair-derived base with the constraints and advantages that implies; and
- be built explicitly on an open, ROS1/ROS2 hybrid software stack, extending rather than replacing an existing ROS1 navigation baseline (Intellwheels).

Existing commercial systems demonstrate that multi-floor host robots are feasible, but do not provide an open architecture that can be adapted to a wheelchair platform [1], [4]. Existing ROS-based research platforms typically address either indoor or outdoor navigation, or single-floor operation, but not the combination of multi-map, multi-floor, and indoor–outdoor routing under the middleware and hardware constraints described earlier [3], [7], [8]. The remainder of this dissertation can therefore be understood as designing and evaluating a navigation layer that brings these capabilities together on top of the Intellwheels baseline, using standard ROS components where possible and introducing new ones only where the surveyed systems leave a gap [5], [9].

2.10 Gap Analysis and Positioning of This Work

The preceding sections have decomposed the Host Robot navigation problem and surveyed relevant state of the art in path planning and navigation stacks, multi-floor and multi-area route planning, robotic middleware, localisation, perception, and comparable guide and host robot systems. Taken together, they show that the individual capabilities required by the Host Robot are well studied in isolation—robust 2D navigation on a single map, probabilistic indoor localisation, GNSS-based outdoor positioning, multi-floor route abstractions, and ROS1/ROS2 interoperability—but that they are rarely integrated into a single navigation layer running on a wheelchair-derived platform and an open ROS-based software stack [5], [7], [8], [10].

2.10.1 Capability Gaps Relative to the State of the Art

Standard ROS1 navigation stacks such as `move_base` provide reliable 2D navigation on a single occupancy grid map, with pluggable global and local planners and layered 2D costmaps, but assume a single static map and do not natively support missions that span multiple floors or maps [9]. Multi-floor planning work, in turn, often relies on integrated 2.5D/3D maps and custom planners able to reason jointly over horizontal and vertical motion [7], [17], which is significantly heavier in sensing and implementation terms than is practical on the existing Intellwheels baseline. Lighter-weight approaches based on per-floor 2D maps and abstract staircase or elevator connectors avoid this complexity but seldom address issues that matter in practice for a host robot, such as coherent map switching in the navigation stack or explicit modelling of elevator waiting time and other vertical-transition costs [18].

At the same time, localisation methods suitable for indoor and outdoor segments make complementary assumptions. AMCL and related map-relative approaches deliver drift-free pose estimates indoors on a known map but have nothing to offer in unmapped outdoor areas [5], while GNSS and RTK-GNSS provide absolute outdoor positioning without any site-specific map, but degrade in the urban-canyon and building-proximate conditions where indoor maps would be most useful [8]. Multi-sensor fusion frameworks in ROS (for example, `robot_localization`) address the algorithmic side of combining these estimates, but leave open how to switch and weight indoor and outdoor sources over the course of a mixed indoor–outdoor mission on a single site [3], [8].

On the middleware side, ROS2 and DDS-based transports offer Quality-of-Service (QoS) controls that benefit a safety-critical override path and low-latency command channel, but the validated navigation baseline and many mature navigation components in this work (`move_base`, Intellwheels) are ROS1-native and lack direct ROS2 ports [9], [10]. Full migration to ROS2 is therefore costly, requiring re-validation of planner and controller behaviour on the target platform, while ROS1–ROS2 bridging introduces additional latency and configuration effort at the translation points [10]. The related systems reviewed in Section 2.9 either avoid this issue by being closed commercial platforms or focus on single-generation ROS deployments that do not face this particular interoperability constraint [1], [3].

Finally, while many guide and host robots demonstrate safe indoor navigation and some support multi-floor operation or outdoor segments, they are typically either closed commercial products or research prototypes tuned to a single building, with elevator integration and route structure implemented as site-specific logic rather than as a reusable multi-map navigation architecture on an open ROS stack [1], [3], [12]. There is limited published guidance on how to extend a standard ROS1 navigation baseline to support the combination of multi-map, multi-floor, and indoor–outdoor missions on a wheelchair-based platform [7], [18].

2.10.2 Relation to Navigation Requirements

Chapter 3 reformulates these qualitative gaps as concrete functional and non-functional requirements on the Host Robot navigation layer. In particular, the system must support waypoint-based

missions across several maps and floors (FR8, FR9), maintain reliable pose estimates indoors and outdoors using different absolute references (FR2), integrate obstacle perception into a layered costmap suitable for `move_base` (FR3, FR5, FR6), expose a low-latency safety override path from the lower control layer (FR11), and perform robust map transitions without corrupting the navigation state (NFR6) [5], [9]. The surveyed literature shows how each of these requirements can be addressed in isolation, but not how to satisfy them jointly under the specific constraints of the existing Intellwheels platform and ROS1/ROS2 split [7], [10].

More concretely, no single existing system simultaneously offers: (i) a waypoint-graph mission layer over multiple 2D maps; (ii) coordinated map switching across map server, localisation, and costmaps in a ROS1 `move_base`-based stack; (iii) integrated indoor AMCL-based and outdoor GNSS-aided localisation on a single site; and (iv) a safety-oriented velocity override path that operates below the planner and spans a ROS1–ROS2 boundary [3], [8]. These are precisely the capabilities needed to realise the Host Robot navigation problem set out in Section 2.2.

2.10.3 Scope and Contributions of This Dissertation

Within this landscape, the contribution of this dissertation is the design and implementation of a navigation layer that extends the Intellwheels ROS1 navigation baseline to meet the Host Robot requirements, without discarding the validated components that baseline already provides. In particular, the work:

- introduces a waypoint-graph-based mission layer that decomposes long, multi-floor routes into sequences of shorter segments, each handed in turn to a standard global/local planner pair in `move_base`;
- implements a map-switching mechanism (MapSwitcher) that coordinates map server, localisation, and costmaps to support consistent multi-map operation across indoor floors and selected outdoor areas;
- integrates indoor AMCL-based localisation and outdoor GNSS-aided localisation under a common fusion interface, allowing the navigation layer to operate across both building interiors and site grounds; and
- provides a safety-oriented velocity override path, implemented over a ROS1–ROS2 bridge, that gives the lower control layer deterministic control over stop and recovery behaviours without modifying the core navigation stack.

The next chapter details the architecture and implementation of this navigation layer, showing how these design choices instantiate the requirements identified above on the concrete Host Robot platform and how they build on, rather than replace, the Intellwheels baseline.

2.11 Summary

This chapter positioned the Host Robot navigation problem within the broader state of the art and decomposed it into a set of subproblems that structure the rest of the dissertation. It first characterised navigation in large, multi-floor, partially outdoor institutions on a wheelchair-derived platform, arguing that missions expressed as high-level guidance tasks must be translated into safe, comfortable, low-level motion commands under mixed ROS1/ROS2 middleware constraints. On this basis, it introduced a problem decomposition covering middleware and layering, environment representation and route structure, path planning and navigation stacks, localisation across indoor and outdoor areas, perception and dynamic obstacle handling, and mission-level navigation in comparable guide and host robot systems.

The chapter then surveyed each of these areas in turn. It reviewed building-wide graphs, per-floor maps with staircase and elevator connectors, and integrated multifloor maps as alternative structures for large institutional environments, and discussed hierarchical topological–metric planners as a way to balance global optimality with computational tractability. In path planning, it contrasted graph-search, sampling-based, and bio-inspired global planners, and examined local planners such as DWA and TEB within integrated stacks like `move_base`, emphasising how a waypoint-graph layer can simplify planning by reducing each hop to a small, well-constrained problem. The chapter also compared ROS1 and ROS2 middleware, discussed ROS1–ROS2 bridging as a pragmatic interoperability strategy, and surveyed localisation methods from proprioceptive odometry and AMCL to RTK-GNSS and multi-sensor fusion, together with range-sensing and costmap fusion approaches for perception.

Finally, the chapter analysed representative museum, campus, hospital, and commercial service robots, as well as open ROS-based host platforms, to identify capability gaps relative to the Host Robot's requirements. It concluded that, while robust single-map navigation, multifloor route abstractions, indoor and outdoor localisation, and ROS1/ROS2 interoperability are each well studied, they are rarely integrated into a single, open navigation layer on a wheelchair-derived platform. In particular, no existing system simultaneously offers a waypoint-graph mission layer over multiple 2D maps, coordinated map switching in a `move_base`-based stack, integrated indoor AMCL-based and outdoor GNSS-aided localisation, and a safety-oriented velocity override path spanning ROS1 and ROS2. Addressing this gap—by extending the Intellwheels baseline to provide exactly these capabilities—defines the scope and contribution of the navigation architecture developed in Chapter 3.

Chapter 3

Solution Architecture and Implementation

3.1 Chapter Overview

This chapter specifies and implements the navigation layer that enables NavHost to execute long, multi-floor missions on the Host Robot platform. It begins by refining the navigation problem into concrete functional and non-functional requirements, taking into account the wheelchair-based base, the FEUP deployment environment, and the mixed ROS 1/ROS 2 software stack. It then presents the overall navigation-layer architecture, showing how the existing Intellwheels move_base configuration is reused as the core single-map navigation engine and extended with a waypoint-graph mission layer, a MapSwitcher for multi-map and multi-floor operation, and a safety-oriented velocity override path. Subsequent sections detail the configuration of the core navigation stack and the design of each custom component, including waypoint representation, the move_base_handler mission orchestrator, the MapSwitcher, the override node, and the depth-to-laserscan perception pipeline, followed by the main execution flows for missions, map transitions, and overrides. The chapter concludes with an overview of the ROS graph and deployment used in simulation and a discussion of the key design decisions that link this architecture back to the requirements defined here and to the experimental evaluation presented in Chapter 4.

3.2 Navigation Problem and Requirements

Large public institutions such as hospitals, university campuses, and administrative buildings impose demanding conditions on autonomous navigation. The host robot must safely guide users across long distances, traverse both indoor and outdoor areas, and cope with varying crowd and object density, all while operating on the chassis and actuation capabilities of an electric wheelchair platform. In this context, the navigation layer is responsible for transforming high-level mission goals, provided by the decision layer, into continuous, safe motion commands that can be executed by the low-level control layer.

Building on the problem characterization and limitations discussed in the previous chapter, this section identifies the functional and non-functional requirements of the navigation layer, taking into account the existing robot hardware, software stack, and deployment environment. It then discusses the main constraints imposed by the wheelchair-based platform and the current software architecture, and concludes by clarifying the specific problem that the new navigation layer is intended to solve.

3.2.1 Functional Requirements (FR)

Environment Representation and Localization

FR1 – Map-based navigation. The system shall use a map of the environment for navigation, including distinct maps for different floors and relevant outdoor areas.

FR2 – Pose estimation. The system shall estimate the robot's pose in the environment by fusing odometry, onboard sensors, and a privileged GNSS-based particle when available in outdoor segments.

FR3 – Obstacle perception. The system shall maintain an up-to-date representation of obstacles by integrating data from onboard sensors (e.g., depth cameras and LiDAR sensors).

Path Planning and Obstacle Avoidance

FR4 – Path computation. The system shall compute a feasible, collision-free path between the current robot pose and the target destination.

FR5 – Static obstacle avoidance. The planner shall avoid static obstacles represented in the environment map.

FR6 – Dynamic obstacle avoidance. The planner shall avoid dynamic obstacles detected during operation, such as pedestrians or other moving objects.

FR7 – Reactive re-planning. The system shall re-plan the path when the current path becomes blocked, unsafe, or otherwise untraversable.

Waypoint and Multi-floor Support

FR8 – Waypoint-based missions. The system shall represent missions as sequences of waypoints, each associated with a pose and, when needed, semantic information (e.g., elevator lobby, ramp entrance).

FR9 – Map and floor transitions. The system shall support changing maps when transitioning between different floors or between indoor and outdoor areas, updating localization and planning consistently.

Inter-layer Communication

FR10 – Strategic upward communication. The navigation layer shall communicate with the decision layer at strategic points in the mission (e.g., waypoint reached, unrecoverable blockage) to allow high-level supervision and decision-making.

FR11 – Downward command interface. The navigation layer shall send velocity commands to the execution layer whenever they are available and shall receive GNSS data and override commands when needed.

3.2.2 Non-Functional Requirements (NFR)

Performance

NFR1 – Real-time localization. The navigation layer shall update localization estimates at a frequency sufficient to support real-time autonomous navigation.

NFR2 – Responsive planning. The navigation layer shall compute or update navigation plans within a time compatible with real-time operation.

Safety and Comfort

NFR3 – Kinematic feasibility. Generated paths shall respect the kinematic and dynamic constraints of the wheelchair platform, including maximum linear and angular velocities and minimum turning radius.

NFR4 – Safe and smooth motion. The navigation layer shall generate velocity commands that produce smooth and predictable motion while maintaining configurable safety distances from pedestrians and obstacles.

NFR5 – Recovery from local failures. Whenever possible, the navigation layer shall recover autonomously from recoverable navigation failures, such as temporary path blockage or moderate localization degradation.

Robustness and Maintainability

NFR6 – Robust map transitions. Map transitions shall preserve navigation consistency by ensuring that localization and planning components are correctly synchronized after a map change.

NFR7 – Modularity. Localization, mapping, planning, and motion control components shall be modular and replaceable.

3.3 Navigation Layer Architecture

3.3.1 Intellwheels Navigation Baseline

The navigation layer developed in this dissertation does not start from a blank slate. Instead, it builds directly on the Intellwheels ROS1 navigation repository introduced in Chapter 2, which already provided a validated mapping between the wheelchair platform, its sensors, and the standard 2D Navigation Stack. In practical terms, this means that the new navigation architecture inherits a working robot description, simulation environment, and `move_base`-based navigation pipeline, and focuses on reorganising and extending these capabilities rather than replacing them.

Within the overall system layering, Intellwheels occupies most of the lower half of the navigation stack. The description and hardware-related packages expose the wheelchair's kinematic

model, frames, and sensor topics to the rest of the ROS1 graph, while the original navigation package configures `move_base` with appropriate global and local planners, costmaps, and recovery behaviours for a single indoor map. On top of this baseline, the Host Robot navigation layer retains the existing planner configuration and costmap structure and treats `move_base` as the component responsible for point-to-point motion on one map, accessed via its standard goal and feedback interfaces.

The new components introduced in this work therefore use Intellwheels as the execution substrate for more complex missions. The waypoint mission orchestrator decomposes each high-level mission into a sequence of waypoints and sends them one at a time as goals to the Intellwheels-configured `move_base` instance, monitoring success or failure and deciding when to advance, retry, or abort. The MapSwitcher operates alongside the Intellwheels navigation stack, coordinating changes to the active map, localization source, and costmaps so that `move_base` continues to operate as if it were always running on a single, consistent map, even though the mission spans several floors and selected outdoor areas.

Finally, the safety-oriented velocity override path interacts with the Intellwheels baseline at the command level. Rather than modifying the existing planner or controller plugins, the override mechanism intercepts and, when required, replaces the velocity commands produced by `move_base` before they reach the wheelchair's low-level control layer. This preserves the tuning effort already invested in the Intellwheels configuration, while allowing the Host Robot navigation layer to enforce safety stops and controlled recoveries under conditions that the original baseline did not have to address, such as mixed indoor–outdoor missions, crowded institutional corridors, and multi-floor journeys.

3.3.2 Physical Platform and Onboard Hardware

As mentioned previously, the navigation layer is intended for deployment on the Host Robot, a wheelchair-derived mobile platform adapted for autonomous operation. The platform combines the original electric wheelchair base and its actuation system with additional sensing, computing, communication, and safety hardware. A three-quarter view of the assembled robot is shown in Figure 3.1.



Figure 3.1: Corner view of the assembled Host Robot platform.

The platform is based on an electric wheelchair chassis and includes a structural upper frame supporting the perception and computing hardware. The front and rear views in Figures 3.2 and 3.3 show the relative arrangement of the wheelchair base, upper structure, sensor supports, control hardware, and external safety components.

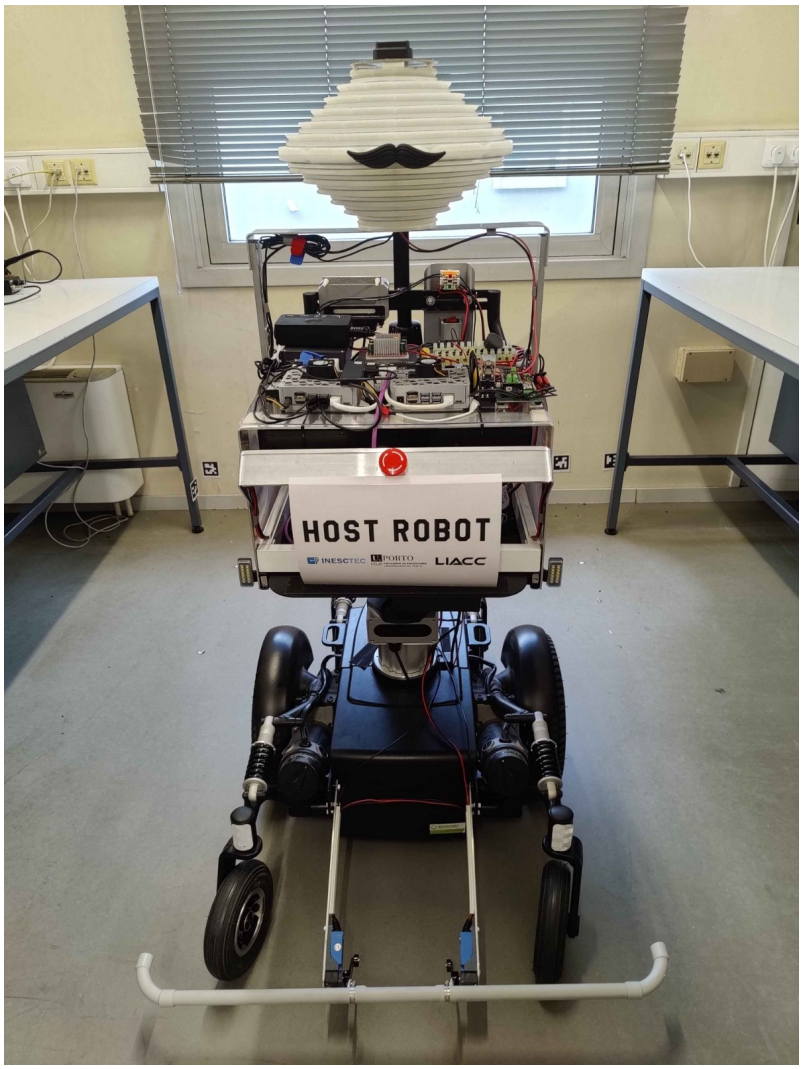


Figure 3.2: Front view of the Host Robot platform.



Figure 3.3: Rear view of the Host Robot platform, showing the upper electronics area and wheelchair-derived base.

The side profile in Figure 3.4 illustrates the vertical arrangement of the platform and the way in which the additional robot structure is integrated with the wheelchair base. The upper frame provides mounting locations for sensors and electronics while preserving the mobility and footprint of the original platform.

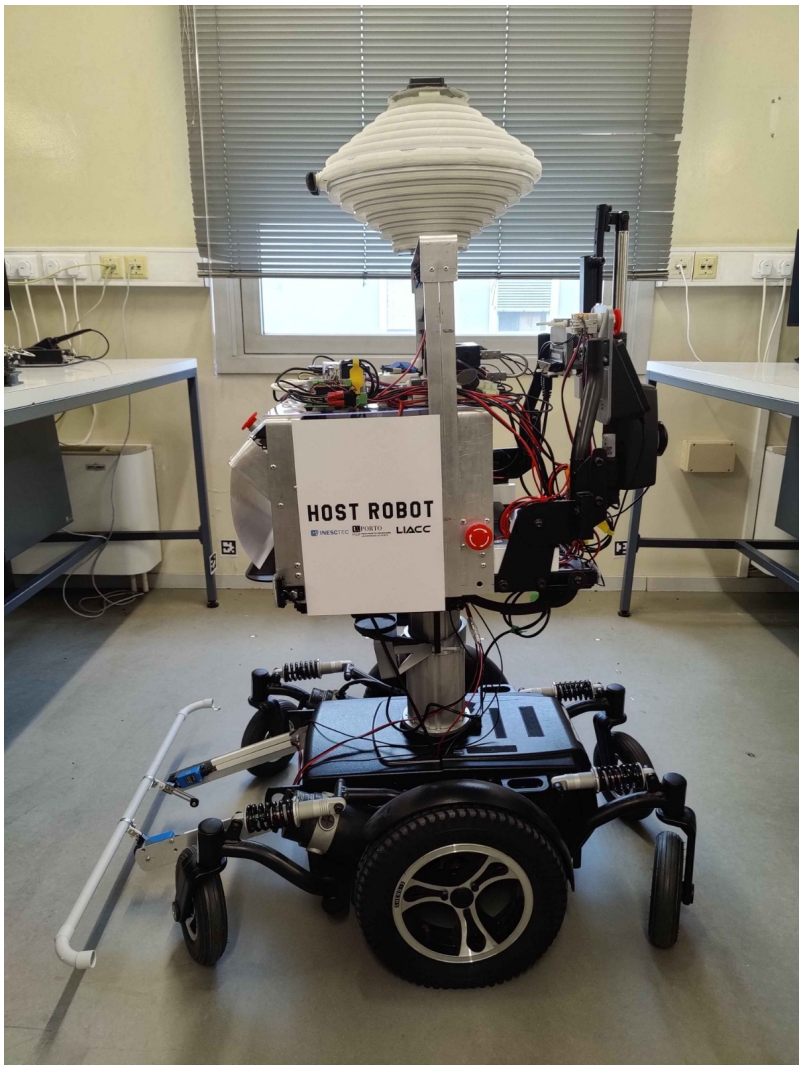


Figure 3.4: Side view of the Host Robot, showing the wheelchair-derived base, upper structural frame, and mounted hardware.

The perception system includes two OAK-D stereo depth cameras and an RP-LiDAR. The OAK-D devices provide depth measurements through stereo vision and extend the robot's perception beyond the single horizontal plane of a conventional planar LiDAR. One OAK-D is mounted at the front of the robot, on the central structural pillar between the drive or motor assembly and the base of the chair. The second OAK-D is mounted at the rear of the chair. Their opposing positions provide complementary coverage of the regions in front of and behind the platform.

The two cameras are attached using custom-designed supports produced through 3D printing. These supports provide a dedicated mechanical interface between the cameras and the robot structure while maintaining a repeatable sensor position and orientation. Stable mounting is important because the camera measurements must be associated with known coordinate frames before they can be used for localization or obstacle detection.

The front OAK-D camera and its custom support are shown in Figure 3.5. The camera is connected to the central pillar by a rigid support that positions the sensor towards the front of the

platform. This location gives the camera a forward-facing field of view while keeping the sensor integrated with the robot's structural frame. In addition, it is angled in such a way to capture the ground as close to the robot as possible, while minimizing the capture of the safety front bumper or parts of the frame itself.

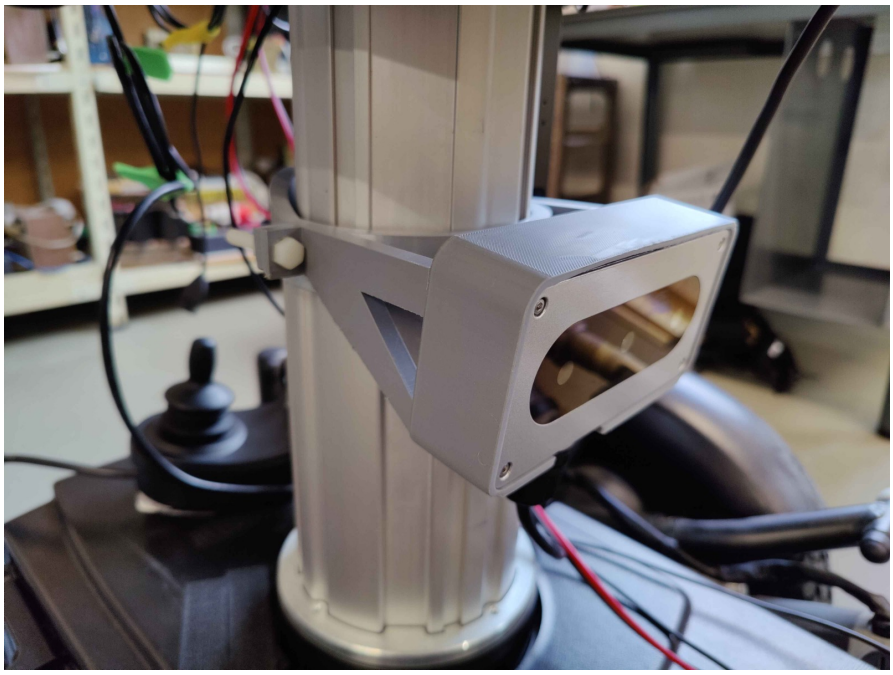


Figure 3.5: Front OAK-D stereo depth camera mounted using a custom-designed support.

The second OAK-D is installed at the rear of the chair, as shown in Figure 3.6. Its position provides depth coverage behind the robot and supports rear-facing perception tasks, including monitoring the area around the platform and assisting guest-reacquisition behaviours.

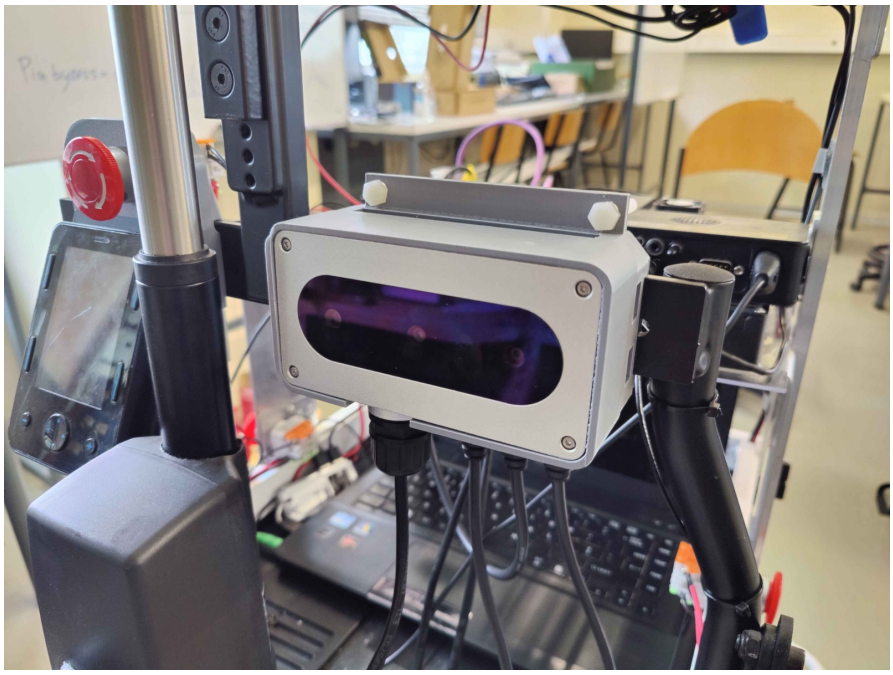


Figure 3.6: Rear OAK-D stereo depth camera installed on the Host Robot.

An RP-LiDAR will also be installed at the front of the platform on top of the front OAK-D camera mount and will supply planar range measurements for navigation and obstacle detection as well as to help amcl with localization with broader coverage for matching with the given maps. The range data from the RP-LiDAR are combined with the depth-derived scans produced from the two OAK-D cameras. This provides the navigation stack with complementary measurements from a conventional planar range sensor and from two cameras that can observe obstacles outside the LiDAR scanning plane. The resulting scans are merged into a common range representation before being supplied to the localization and costmap components.

The computing and power electronics are installed on the upper platform of the robot. The onboard arrangement includes the processing computer, power conversion components, wiring, connectors, cooling hardware, and interfaces for the sensors and low-level control system. These components are integrated into the platform so that the system can operate as a self-contained robot while remaining accessible for maintenance and further development.

The navigation software is intended to run on an NVIDIA Jetson Nano installed on the robot. The Jetson Nano acts as the onboard computer for the navigation layer and provides the execution environment for the perception, localization, planning, mission-coordination, and communication processes. The relevant computing hardware is shown in Figure 3.7.

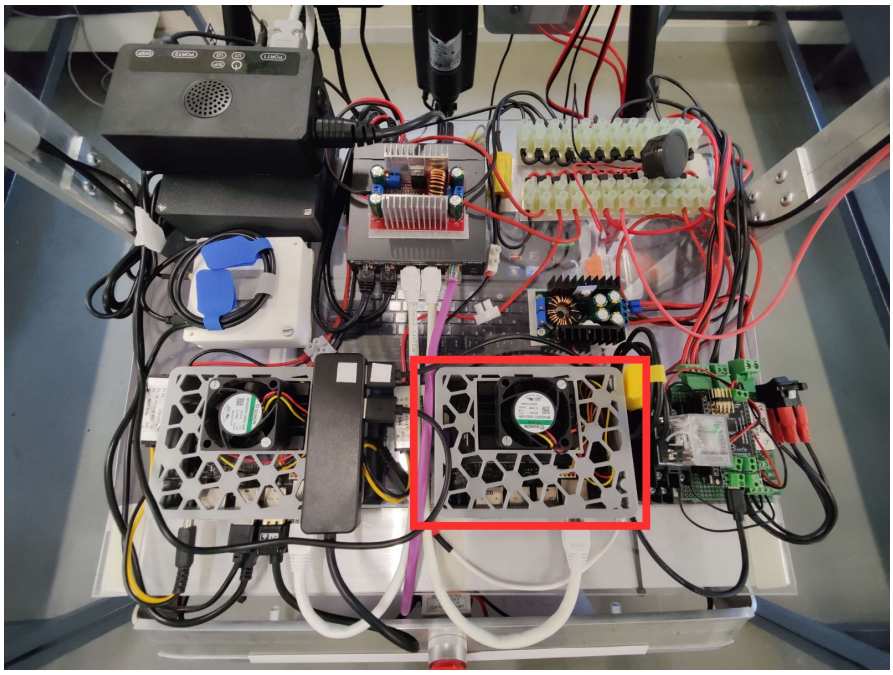


Figure 3.7: Onboard computing and electronic hardware based on an NVIDIA Jetson Nano (marked in red).

Using the Jetson Nano as the onboard computer keeps the main navigation processes close to the robot and avoids requiring an external computer during normal operation. It also imposes practical constraints on the implementation. Depth-image processing, scan conversion, sensor fusion, localization, costmap updates, and path planning must share the available computational resources. The navigation architecture therefore uses separate ROS nodes and standard topic and service interfaces, allowing individual components to be configured or adjusted without redesigning the complete system.

The physical arrangement shown in these figures is directly reflected in the software architecture presented in the following sections. The two OAK-D cameras provide complementary depth observations, the RP-LiDAR supplies planar range data, and the Jetson Nano executes the software required to transform sensor measurements, estimate the robot pose, plan motion, and coordinate the navigation mission.

3.3.3 Proposed Architecture

The proposed navigation layer extends the Intellwheels ROS 1 baseline around the standard `move_base` navigation stack. Its purpose is to support long-range missions over several maps while retaining the mature local-navigation capabilities provided by `move_base`. Figure 3.8 presents the main components and the information flow between them, from a high-level destination request to the velocity commands delivered to the execution and control layer.

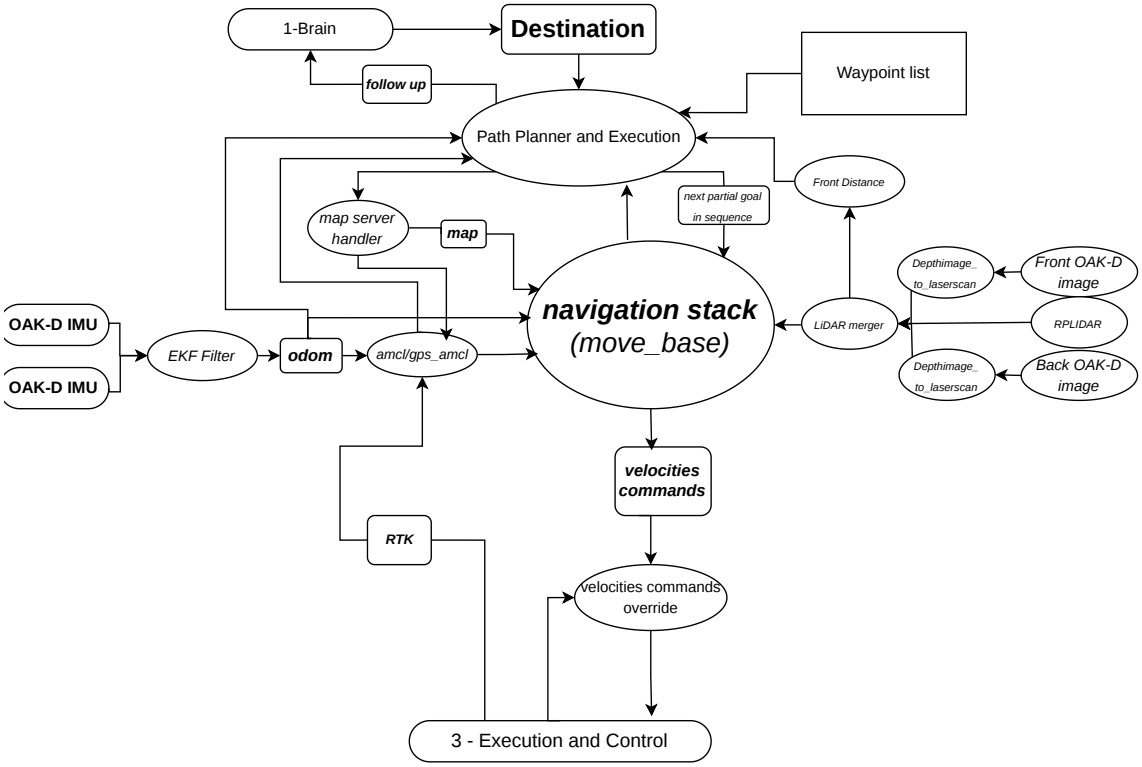


Figure 3.8: Overview of the proposed navigation-layer architecture. A destination is decomposed into an ordered waypoint sequence, whose individual goals are executed by the `move_base`-based navigation stack. Map and localization transitions are coordinated through the map server handler, while fused odometry and merged range data support localization and obstacle-aware navigation. The front-distance node provides a simplified forward-obstacle signal to the mission layer, and the velocity override node supports temporary guest-reacquisition manoeuvres before commands reach the execution layer.

At the centre of the architecture is the navigation stack, implemented using `move_base`. For the currently active map, it receives a pose goal, combines the available localization and obstacle information, and generates velocity commands that drive the robot towards that goal. The navigation stack is therefore responsible for detailed motion within the current navigation context. The surrounding components provide the higher-level route structure, the mechanisms required to change that context, and the additional perception and control signals needed by the Host Robot scenario.

Mission planning and waypoint execution A mission enters the navigation layer as a destination supplied by the decision layer. The destination is passed to the Path Planner and Execution block, implemented by the waypoint mission orchestrator. Rather than sending the destination directly to `move_base`, the orchestrator uses the building-wide waypoint graph to calculate an ordered route through meaningful passage points, such as corridor junctions, door approaches, and elevator lobbies. The graph route is computed using A* and becomes the waypoint list shown in Figure 3.8.

The orchestrator then selects one waypoint at a time and submits it to `move_base` as the next partial goal in the sequence. This separates long-range mission structure from local motion: the waypoint layer determines the route through the building and the locations where transitions must be handled explicitly, while `move_base` plans and executes the detailed path between consecutive waypoint goals on the active map. The orchestrator also monitors the outcome of each goal and can retry, terminate, or re-plan the mission when the current route can no longer be followed. The detailed waypoint representation, state machine, and execution flow are described in Sections 3.5.1, 3.5.2, and 3.6.1.

Map and localization context The map server handler, implemented by `MapSwitcher`, provides the transition mechanism required when consecutive waypoints belong to different maps. This may correspond to movement between floors, the use of an elevator, or a transition between an indoor and an outdoor area. The orchestrator detects these transitions from the map identifiers associated with the waypoint list and requests a map switch before navigation continues in the new context.

As indicated in Figure 3.8, a map switch affects more than the occupancy grid supplied to `move_base`. It also selects the appropriate localization configuration for that map and reinitializes the navigation context before the next goal is executed. Indoor maps use the map-relative localization pipeline, whereas outdoor operation can use the GNSS-aided configuration. After the transition has completed, the navigation stack continues to operate through the same interface, receiving the next partial goal from the waypoint orchestrator. The exact map-switching sequence is detailed in Section 3.5.4.

Odometry, localization, and perception The lower part of Figure 3.8 shows the sensor and localization inputs that support pose estimation and obstacle-aware navigation. Inertial measurements from the two OAK-D devices are fused by an EKF to produce the odometry estimate used by the active localization component. Depending on the current map, this component is either the indoor AMCL configuration or the GNSS-aided `gps_amcl` configuration. During outdoor operation, RTK measurements provide an absolute-position reference for the GNSS-aided localization process. The indoor–outdoor localization strategy is discussed further in Section 3.4.3.

The perception pipeline combines range information from two OAK-D stereo depth cameras and one front-mounted RP-LiDAR. One OAK-D is mounted at the front of the wheelchair base, on the central structural pillar between the drive assembly and the base of the chair. A second OAK-D is mounted at the rear of the chair. Custom-designed, 3D-printed mounts secure both cameras to the platform and position them to provide complementary forward and rear coverage. The RP-LiDAR is mounted at the front of the robot and provides conventional planar range measurements in the direction of travel.

Each OAK-D camera produces depth data that are processed by an independent instance of the depth-image-to-LaserScan node. The front and rear instances convert their respective depth streams into separate depth-derived LaserScan topics. Their outputs are kept distinct through ROS

topic remapping, allowing each camera to retain its own observation source and frame relationship before fusion. The detailed depth-to-LaserScan conversion process, including point-cloud reconstruction, floor removal, obstacle-height filtering, and planar scan projection, is described in Section 3.5.6.

The two depth-derived scans and the front RP-LiDAR scan are combined by the LiDAR merger into a single `/base_scan` topic. This merged scan is the common range representation provided to the localization pipeline and to the global and local costmaps of `move_base`. The front-mounted LiDAR provides robust planar obstacle measurements near the direction of travel, while the front and rear OAK-D cameras extend obstacle coverage above or outside the LiDAR scan plane and contribute information from both sides of the wheelchair platform. Consequently, the navigation stack receives one standard LaserScan interface while benefiting from the complementary coverage of all three sensors.

In addition to the merged scan used by `move_base`, the architecture includes the front-distance node. This node reduces the forward portion of `/base_scan` to a scalar `/front_distance` measurement. The waypoint mission orchestrator uses this signal between waypoint goals to determine whether the passage ahead is available. A temporarily blocked passage causes the orchestrator to wait; a blockage that persists beyond the configured timeout causes the affected topological connection to be removed and the route to be re-planned. The front-distance processing and its use by the mission layer are described in Sections 3.5.3 and 3.5.2.

Velocity commands and guest reacquisition The final command path begins with the velocity commands produced by `move_base`. Under normal conditions, these commands pass through the velocity override node unchanged and continue to the execution and control layer. The override node is included primarily to support guest-reacquisition behaviour rather than to replace the navigation stack's own safety mechanisms.

More specifically, when the lower guest-monitoring layer determines that the person being guided has left the rear camera's field of view, it can request a temporary override. The override node can stop the robot and rotate it left or right, according to the direction indicated by the lower layer, allowing the camera system to search for and reacquire the guest. Once the guest has been detected again, normal `move_base` command forwarding is restored. The command-selection mechanism also ensures that navigation commands and guest-recovery commands are not sent to the base simultaneously, providing a secondary safety benefit. Its detailed behaviour is presented in Section 3.5.5.

Overall, the architecture combines a conventional local-navigation stack with the additional coordination required by a large, multi-map institution. The waypoint mission orchestrator structures long-range navigation, `MapSwitcher` manages transitions between navigation contexts, the localization and perception pipelines provide the pose and obstacle data required by `move_base`, the front-distance node supports blocked-passage handling, and the override node enables temporary guest-reacquisition manoeuvres. The following sections describe these components and their execution flows in greater detail.

3.3.4 Communication with Other Robot Layers

The navigation layer does not operate in isolation. It is embedded in a broader software architecture in which the user interface and decision layer and the low-level motion control layer each implement distinct responsibilities. In the current robot platform, these components are distributed across ROS 1 and ROS 2 environments, and communication between them is mediated by a ROS 1–ROS 2 bridge. Figure 3.9 provides a conceptual overview of these interactions, showing the ROS 1 navigation layer between ROS 2 decision and execution layers and the ROS 1–ROS 2 bridges through which goals, feedback, velocity commands, and sensor data flow.

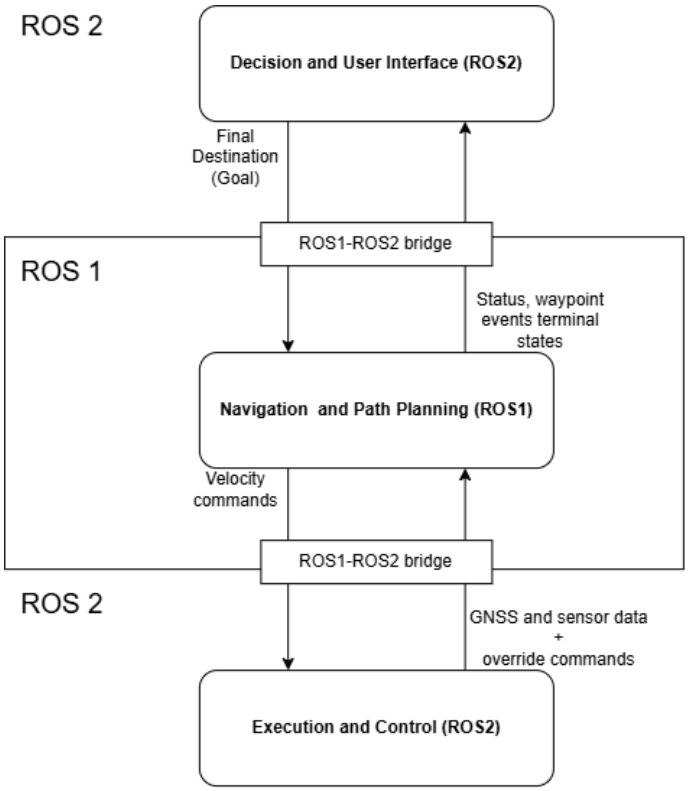


Figure 3.9: Conceptual interaction between robot layers across ROS generations. The ROS 1 navigation and path planning layer exchanges final destination goals and status events with the ROS 2 decision and user interface layer, and velocity commands, GNSS and sensor data, and override commands with the ROS 2 execution and control layer via ROS 1–ROS 2 bridges.

Within this context, the navigation layer acts as the intermediary that converts high-level navigation requests into motion commands while reporting meaningful progress and fault information back to the upper layers.

From the navigation layer's perspective, the user interface and decision layer, implemented in ROS 2, are high-level clients that issue navigation requests and consume navigation events. Navigation requests are represented as final destination goals, transmitted as ROS topics that include position, orientation, and an associated floor or map identifier. In response, the navigation layer publishes progress updates indicating, for example, when intermediate waypoints are reached and when terminal states such as "goal reached", "navigation aborted", or "failure due to persistent

blockage" occur, allowing the decision layer to re-plan the mission, request assistance, or inform the user about delays.

The low-level motion layer, together with the robot's sensors, is responsible for executing velocity commands and providing the information required for localization and obstacle avoidance. In the current architecture, this layer runs in ROS 2 and is also connected to the navigation layer by means of the bridge. The navigation layer outputs velocity commands as ROS topics (e.g., linear and angular velocities) that are consumed by the low-level controller of the wheelchair chassis whenever a valid local plan is available, and these commands are suspended or modified in response to override signals or safety conditions.

In the opposite direction, the low-level layer publishes sensor-derived information required by the navigation stack, such as odometry estimates for local motion, GNSS measurements used by the `amcl_gps` module when the robot is outdoors, and additional feedback signals such as wheel status or execution health. These data are made available as ROS topics and routed across the bridge into the ROS 1 domain where the navigation algorithms execute, allowing the navigation layer to maintain a consistent estimate of the robot state and to update its internal costmaps and planners.

The low-level motion layer can also issue override commands that temporarily modify the behaviour of the navigation layer, including requests to stop the robot temporarily (e.g., due to a safety trigger), commands to bias the robot's motion to the left or right when a human operator needs to adjust course, and signals to resume normal autonomous operation after a temporary override. These overrides are modelled as ROS messages or services that the navigation layer monitors; when an override is active, the navigation layer may pause goal tracking, suspend or modify velocity command generation, and later resume nominal navigation behaviour when the override is cleared.

Although the technical details of the bridge configuration are described later in the implementation chapter, its role is conceptually important to the architectural view: it provides transparent translation between ROS 1 and ROS 2 message types and communication primitives, so that high-level navigation goals, feedback, velocity commands, odometry, GNSS, and override signals can be exchanged reliably between the ROS 1 navigation algorithms and the ROS 2 decision, control, and sensing infrastructure. From the perspective of this chapter, the bridge is therefore part of the communication infrastructure that allows the navigation layer to interact with other robot layers as if they were all within a single ROS ecosystem, despite being implemented in different ROS versions.

3.4 Core Navigation Stack

The navigation layer builds on the standard ROS 1 2D Navigation Stack, centred on the `move_base` node, as a robust and well-established foundation for navigation on individual occupancy-grid maps. This choice takes advantage of the Intellwheels platform's existing, tuned global-planning, local-planning, costmap, and recovery capabilities, while focusing the contribution of this work

on extending those capabilities to the larger institutional setting. The architecture therefore retains `move_base` as the reliable single-map navigation engine and augments it with a waypoint mission layer, coordinated map switching, indoor–outdoor localization support, and a velocity-command override interface for guest reacquisition. In this way, the system combines a proven navigation core with the mission-level coordination required for long-range, multi-floor, and multi-map guidance tasks.

3.4.1 `move_base` as Navigation Engine

On each active map, `move_base` is responsible for turning a goal pose into a continuous stream of velocity commands. It maintains two layered costmaps—a global costmap built from the static map and longer-lived obstacle observations, and a local costmap centred on the robot and updated at higher frequency from live sensor data—and runs a global and a local planner plugin concurrently against these costmaps. In this work, the global planner is configured as a grid-based A* variant operating over the global costmap, while the local planner is configured as the Timed Elastic Band (TEB) planner, selected for its ability to respect the wheelchair's kinodynamic limits and to generate smooth trajectories in narrow institutional corridors.

A key design choice is that `move_base` retains full responsibility for navigating within rooms and along corridors between passage points on a given floor. Once a waypoint has been chosen at a door, junction, or elevator lobby, the path from that passage point to the next is left entirely to `move_base`, which operates in the environment that best matches its design assumptions: a single, static 2D map with layered costmaps and a global–local planner pair. The waypoint layer is therefore not used to micro-manage motion inside rooms, but to ensure that, when the robot approaches structurally or socially tricky locations such as doorways, it does so from a prime position and orientation that gives `move_base` a favourable starting point to execute the actual traversal.

3.4.2 Integration Points in the Architecture

Within the overall navigation layer architecture, three integration points with `move_base` are particularly important: goal management, costmap management, and velocity output.

First, goal management is delegated to the waypoint mission orchestrator (`move_base_handler`, Section 3.5.2). The decision layer never sends arbitrary long-distance goals directly to `move_base`; instead, all goals are issued by the orchestrator as individual waypoints along a precomputed route over the building-wide graph. These waypoints are placed at semantically meaningful passage points—door centrelines, corridor junctions, elevator lobbies—so that each hop corresponds to a floor segment that `move_base` can handle autonomously, from one passage point to the next, without further intervention from the waypoint layer. The orchestrator monitors the standard `move_base`/goal, `move_base`/status, and `move_base`/result action topics and decides when to cancel, retry, or advance, centralising timeout behaviour and high-level failure reporting in a single component.

Second, costmap and map management during floor transitions are handled indirectly via the `MapSwitcher` node (Section 3.5.4). When the waypoint sequence indicates that the next passage point lies on a different map (for example, at the exit of an elevator on another floor), the orchestrator requests a map change before issuing the next goal. `MapSwitcher` then coordinates loading the new occupancy grid, restarting the appropriate localization node (AMCL or AMCL_GPS), seeding it with an initial pose, and finally triggering a costmap reset by calling `/nav/clear_costmaps` on `move_base`. From `move_base`'s perspective, this appears as a standard costmap clear and new goal on a new map; the sequencing of map server, localization, and pose initialization lies entirely outside the stack.

Third, velocity output from `move_base` is not sent directly to the base controller but passes through the velocity override node (Section 3.5.5). The node subscribes to the `cmd_vel` topic published by `move_base` and republishes either those commands unchanged or override commands generated by the lower safety layer, depending on its current mode. This arrangement ensures that the navigation engine remains unaware of safety overrides, while the lower layer can reliably enforce stops or corrective rotations without modifying any `move_base` internals.

Taken together, these integration points allow the navigation layer to extend a conventional `move_base` configuration into a waypoint-driven, multi-map, safety-aware navigation system. The waypoint system is used sparingly and deliberately at floor transitions and tricky passages to set up good approach poses, while the detailed motion between passage points on a given floor is left entirely to `move_base` in the environment where it performs best, simplifying both configuration and future maintenance on the Host Robot platform.

3.4.3 Indoor–Outdoor Localization Strategy

The navigation layer uses different localization configurations for indoor and outdoor parts of a mission. Indoor maps use standard Adaptive Monte Carlo Localization (AMCL), while outdoor maps use `gps_amcl`, a GNSS-aided adaptation of the AMCL approach. This separation reflects the different sources of information available in the two environments. Inside the building, the robot can compare range observations with a detailed occupancy map containing walls, corridors, doors, and other stable geometric features. In outdoor areas, GNSS can provide an additional absolute-position reference, particularly over longer routes where odometry alone would progressively accumulate drift.

For indoor navigation, AMCL estimates the robot pose relative to the active occupancy-grid map. It maintains a set of pose hypotheses, represented as particles, and repeatedly updates their likelihood according to the robot's estimated motion and the agreement between incoming laser observations and the occupancy grid. Odometry provides a continuous short-term motion estimate, while laser scans supply map-relative evidence that corrects accumulated drift. This makes AMCL well suited to structured indoor spaces such as the FEUP corridors and rooms represented by the floor-specific occupancy maps used in this work [5], [9].

Outdoor operation presents a complementary localization problem. The robot may move through larger and less structured areas, where range observations alone can provide weaker or

more ambiguous map evidence than in indoor corridors. At the same time, a GNSS receiver can provide an absolute position reference. The `gps_amcl` package adopted in this work is a modified implementation of the standard ROS AMCL node that incorporates GNSS data into the localization process [23]. It retains the particle-filter structure of AMCL while extending the measurement update with GNSS-based position information.

In standard AMCL, the particle filter first predicts the movement of each pose hypothesis from the odometry estimate. It then evaluates the particles according to the agreement between the laser scan observed by the robot and the scan that would be expected at each hypothesized pose in the occupancy map. Particles that better explain the observed scan receive greater weight, after which the particle set is resampled so that subsequent pose estimates concentrate on more probable regions of the map.

The `gps_amcl` implementation follows this same general cycle but adds a GNSS-related observation to the particle evaluation. The received GNSS position provides an additional criterion for assessing whether a particle represents a plausible robot pose. Particles whose positions are more consistent with the GNSS estimate receive greater weight, while particles that are substantially displaced from the GNSS observation are penalised. The resampling stage then concentrates future pose hypotheses in areas jointly supported by odometry, laser-map agreement, and GNSS information [23].

The resulting outdoor localization strategy combines three complementary forms of information. Odometry provides frequent estimates of incremental motion but accumulates drift over time. Laser observations provide geometric consistency with the active map whenever useful environmental features are visible. GNSS provides an absolute position reference that can constrain accumulated position error during outdoor movement. Rather than treating GNSS as a replacement for odometry or laser observations, the `gps_amcl` configuration uses it as an additional source of evidence within the AMCL-based localization process.

This distinction is particularly relevant for the Host Robot because the intended deployment is not limited to a single type of environment. A mission may begin in an indoor corridor, continue through an outdoor access route, and later return to another indoor area. The navigation layer must therefore change localization context without requiring a different mission format, waypoint representation, or `move_base` goal interface. From the perspective of the waypoint mission orchestrator, each route element still belongs to a map and has a pose goal. From the perspective of `move_base`, the active localization configuration continues to provide the robot pose in the active `map` frame.

The localization mode is associated with the map rather than with an independent mission-level decision. Each map configuration specifies the localization mode required for operation in that environment. Indoor maps select `amcl`; outdoor maps select the GNSS-aided `amcl_gps` configuration, which launches `gps_amcl`. When the waypoint mission orchestrator identifies that the next route element belongs to another map, it can also determine whether the required localization configuration changes. The orchestrator then delegates the operational transition to `MapSwitcher`, which loads the target map and starts the localization configuration associated

with it.

This arrangement keeps the rest of the navigation architecture consistent across indoor and outdoor operation. The waypoint graph continues to represent long-range routes, the mission orchestrator continues to dispatch partial pose goals, and `move_base` continues to plan local motion using the active map, costmaps, and pose estimate. Only the configuration responsible for producing that pose estimate changes. The detailed implementation of map loading, localization restart, initial-pose publication, localization verification, and costmap clearing is described in Section 3.5.4.

The GNSS-aided configuration should not be interpreted as a guarantee that outdoor localization is always more accurate than indoor AMCL. GNSS performance can degrade close to buildings because of reduced sky visibility, multipath propagation, and non-line-of-sight reception. Conversely, AMCL depends on the availability of a suitable occupancy map and sufficiently informative laser observations. The purpose of the proposed strategy is therefore to select the localization configuration most appropriate to the active environment while preserving a common navigation interface above both modes [5], [8].

The indoor–outdoor localization strategy is evaluated in the outdoor navigation experiment presented in Chapter 4. That experiment verifies that the navigation layer can operate with `gps_amcl` on an outdoor FEUP map while retaining the same waypoint-planning and `move_base` interfaces used for indoor AMCL-based missions.

3.5 Custom Navigation Components

3.5.1 Waypoint structure and set up

Waypoints provide the discrete structure on which long-range missions are built. Each waypoint encapsulates a small amount of metric, topological, and semantic information: a 2D position (x, y) in the corresponding map frame, the identifier of the map it belongs to, a type label describing its role (e.g., corridor junction, door approach, elevator lobby), and a set of neighbour links to other waypoints in the building-wide graph. Together, this information allows the mission planner to treat the environment as a sparse graph of meaningful places rather than as a dense grid of individual cells, while still retaining the precise pose needed to issue goals to `move_base`.

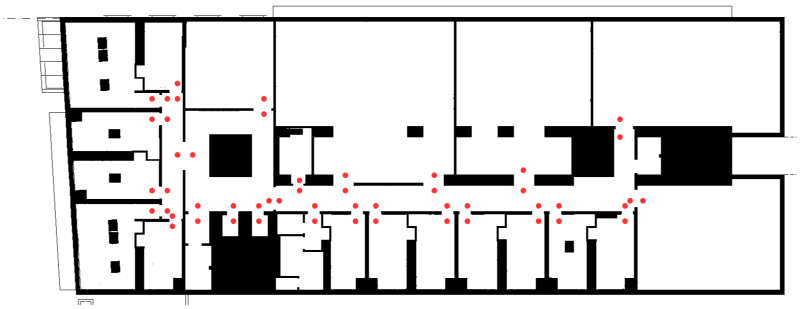


Figure 3.10: Example waypoint layout on a single-floor map. Waypoints are placed at corridor junctions, on each side of doorways, and at elevator lobbies, with neighbour links forming a sparse navigation graph over the free space.

In the physical environment, waypoints are placed deliberately at transition points rather than uniformly over free space. Doors are modelled by a pair of waypoints, one on each side of the doorway, aligned with the door centreline and oriented so that the robot approaches the opening along a favourable heading before attempting to traverse it, as illustrated in Figure 3.10. Similar patterns are used at corridor junctions and elevator lobbies, where the waypoint pose determines how the robot positions itself before turning or entering the lift. For multi-map operation, certain waypoints are duplicated across maps: a waypoint representing the inside of an elevator, for example, appears both in the map of the departure floor and in the map of the arrival floor, with distinct map identifiers but the same logical node in the waypoint graph.

Concretely, waypoints are described in a YAML file in which each entry specifies the following fields:

```
waypoint_id:
  x: <float>          # Map-frame x coordinate
  y: <float>          # Map-frame y coordinate
  map: <string>       # Identifier of the map/floor this waypoint
                        belongs to
  type: <string>      # Semantic role, e.g. 'door', 'junction',
                        'elevator'
  neighbors:          # List of adjacent waypoint IDs in the navigation
                        graph
    - <neighbor_id_1>
    - <neighbor_id_2>
    # ...
```

The `type` field is used to attach additional behaviour to specific classes of waypoints without complicating the core graph structure. For example, waypoints of type `elevator` are treated specially by the waypoint mission orchestrator during testing: when the robot reaches such a node, the orchestrator waits instead of immediately advancing, allowing the simulation to move the robot between maps to emulate an elevator journey before navigation resumes on the new

floor. Similar conventions can be introduced for other waypoint types (such as ramps or narrow doorways) to trigger type-specific checks or timing behaviour at semantically important locations while keeping the underlying waypoint representation uniform.

This waypoint design has two direct benefits for navigation. First, by concentrating waypoints at structurally or socially sensitive locations and encoding neighbours explicitly, the global route-planning problem reduces to finding a path over a relatively small graph of nodes, while the detailed motion between consecutive waypoints is delegated entirely to `move_base` on a single map. Second, embedding the map identifier into each waypoint allows the waypoint mission orchestrator to detect upcoming floor or area transitions simply by comparing successive waypoints, and to request a map switch at the appropriate passage point without any additional logic about building layout.

3.5.2 Waypoint Mission Orchestrator (`move_base_handler`)

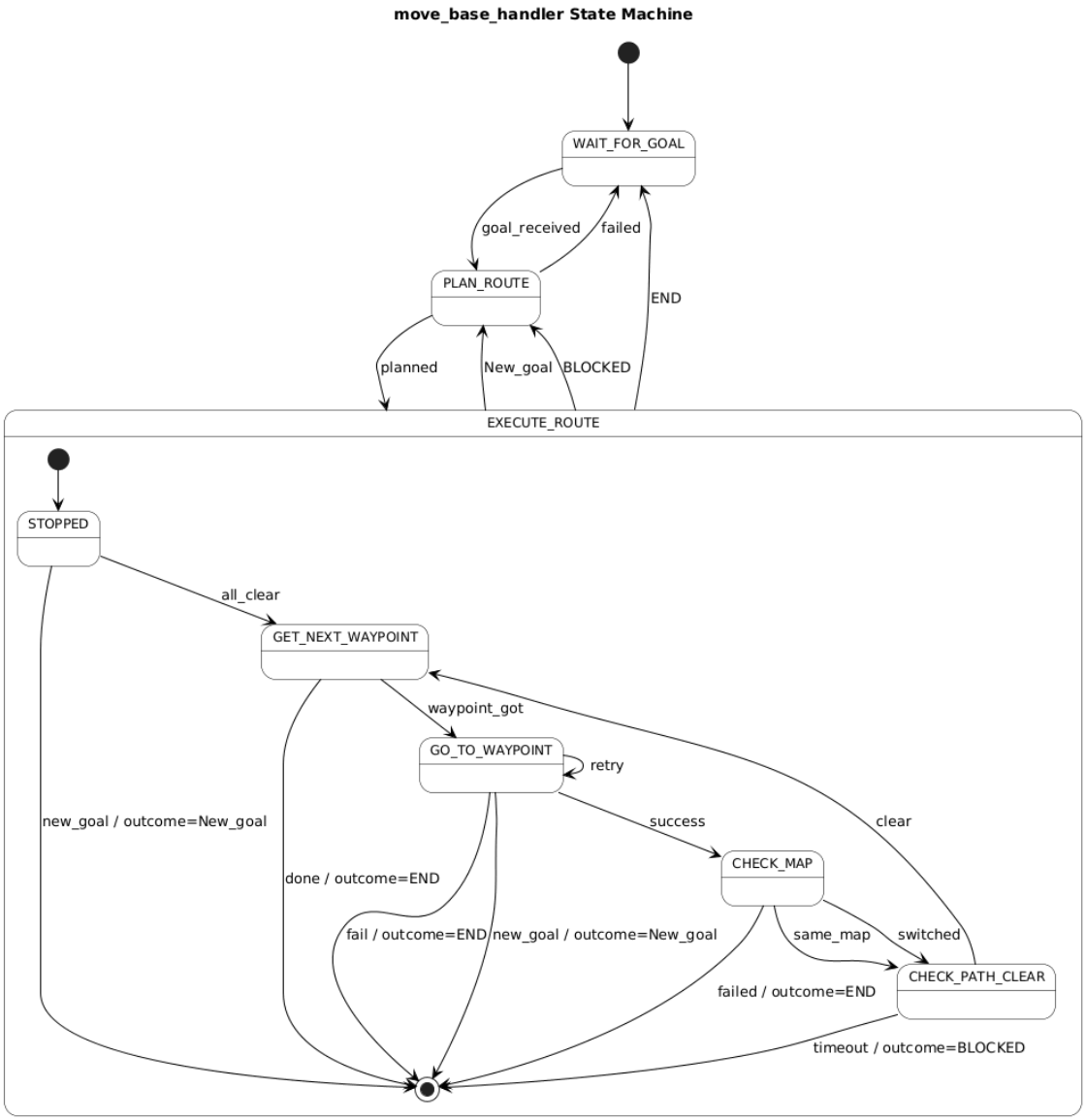


Figure 3.11: State machine of the `move_base_handler` waypoint mission orchestrator. The node alternates between waiting for a mission, planning a route over the topological graph, and executing the resulting waypoint sequence while handling map checks, retries, and pre-emption.

The waypoint mission orchestrator, implemented as the `move_base_handler` node, is the component responsible for converting a high-level navigation request into an executable sequence of local navigation goals. It sits between the decision layer, which provides a destination, and the ROS `move_base` stack, which performs detailed motion planning and control on the currently active map. Rather than sending a single distant destination directly to `move_base`, the handler represents the mission as a sequence of topological waypoints and dispatches them individually. This allows the mission to span multiple maps and floors while keeping each individual

`move_base` goal within the local operating context of a single map.

The node receives destination requests as `NavigationGoal` messages on the `/goal` topic. Each message provides the destination coordinates and the identifier of the map in which those coordinates are defined. The most recently received goal is stored by the `GoalMonitor` helper object together with a flag indicating that a new mission is pending. This flag is used both to start navigation from the idle state and to interrupt an active mission when the decision layer submits a replacement destination.

The node also maintains a lightweight representation of the robot's current navigation context. The `RobotPose` helper uses the TF transform between `map` and `base_link` to obtain the robot's current two-dimensional position. This position is updated periodically and is used as the start position for topological route planning. In addition, `RobotPose` stores the identifier of the currently active map. This identifier is initialized from a ROS parameter and updated only after a successful map switch, allowing the mission logic to determine whether the following route element requires a transition to another map or localization regime.

Internally, the node is implemented as a hierarchical SMACH state machine, shown in Figure 3.11. The outer state machine contains three principal states: *WAIT_FOR_GOAL*, *PLAN_ROUTE*, and *EXECUTE_ROUTE*. The first two states manage the lifetime of a mission, while *EXECUTE_ROUTE* is itself a nested state machine that iterates over the planned waypoint sequence. The nested machine has the outcomes `END`, `New_goal`, and `BLOCKED`. These outcomes allow the outer machine to distinguish between normal completion or failure, receipt of a replacement mission, and the need to re-plan after a topological connection becomes unavailable.

In *WAIT_FOR_GOAL*, the node remains idle until `GoalMonitor` detects a new message on `/goal`. Once a request is available, the outer state machine transitions to *PLAN_ROUTE*. The received goal becomes the active mission destination, and the new-goal flag is cleared so that subsequent goal messages can be recognised as replacements during execution.

Route planning is performed over the topological graph loaded from `config/topological_map.yaml`. Each graph node contains at least a map-frame position, the map to which it belongs, and a list of neighbouring nodes. To create a route, the handler first selects the graph node nearest to the robot's current TF position, considering only nodes that belong to the active map. It similarly selects the graph node nearest to the requested destination, considering only nodes associated with the destination map. This produces a topological start node and goal node even when the robot or destination is not located exactly at a predefined waypoint.

The node uses A* to search for a path between these two graph nodes. The graph edge cost is the Euclidean distance between connected nodes, and the same Euclidean distance is used as the A* heuristic. During planning, the algorithm maintains an open priority queue, cumulative path costs, estimated total costs, and a set of already-expanded nodes. When the goal node is reached, the predecessor relation is followed backwards to reconstruct the ordered node sequence. The requested destination coordinates are then appended as the final element of the path. Consequently, a planned route normally has the form

`start node` → `intermediate nodes` → `goal-area node` → `final coordinate goal`.

If no suitable start or goal node can be identified, if the robot pose is unavailable, or if A* cannot reach the goal node, *PLAN_ROUTE* returns `failed`. The outer state machine then returns to *WAIT_FOR_GOAL*, ready to receive another mission request. If the path is successfully generated, it is stored in the SMACH userdata and passed to the nested *EXECUTE_ROUTE* state machine.

The nested execution machine begins in *STOPPED*. This state provides a controlled entry into route execution and checks whether a new mission has arrived before selecting the next waypoint. If a replacement goal is already pending, the nested machine exits with the `New_goal` outcome and the outer machine returns to *PLAN_ROUTE*. If no replacement goal is pending, the current implementation proceeds immediately to *GET_NEXT_WAYPOINT*. Although the state is named *STOPPED*, it does not currently wait for a separate physical or sensor-derived all-clear signal; it is an execution-control state rather than a braking or obstacle-detection mechanism.

In *GET_NEXT_WAYPOINT*, the first remaining item is removed from the planned path. When that item is a graph-node identifier, its (x, y) position and map identifier are obtained from the topological-map configuration. If another route element follows, the handler computes a yaw angle from the current waypoint towards that following element and converts this yaw into a quaternion. The resulting orientation encourages the robot to arrive at the waypoint facing the direction of the next route segment. This is particularly useful at passage points such as corridor junctions, door approaches, and elevator lobbies. The final route element is represented by a coordinate tuple rather than a graph-node identifier; it is converted into the final `move_base` goal with a default orientation and no associated map identifier. If the path is empty when the state is entered, *GET_NEXT_WAYPOINT* returns `done`, causing the nested machine to exit through `END`.

The pose generated by *GET_NEXT_WAYPOINT* is sent to the `move_base` action server in *GO_TO_WAYPOINT*. The handler creates a `MoveBaseGoal` in the `map` frame, fills its position and orientation from the selected waypoint, and sends it through an `actionlib` client. It then waits for the action result in short intervals. During this period, the handler also checks whether a new destination has been received. If so, it cancels the currently active `move_base` goal and returns `new_goal`. The nested machine exits with outcome `New_goal`, and the outer state machine computes a new path for the updated mission.

A `SUCCEEDED` result from `move_base` indicates that the current waypoint has been reached and causes a transition to *CHECK_MAP*. If the action result is `PREEMPTED`, the state returns `retry`, which immediately re-enters *GO_TO_WAYPOINT* and reissues the same waypoint goal. This retry mechanism is distinct from topological re-planning: it does not recompute the route. An `ABORTED` or `REJECTED` result, as well as other unexpected terminal action states, produces `fail`. This causes the nested state machine to exit with `END` and the outer machine to return to *WAIT_FOR_GOAL*.

Map and localization coordination is handled in *CHECK_MAP*, which runs after successful arrival at the current waypoint. The state examines the next unexecuted item in the path rather than the waypoint that has just been reached. If no route elements remain, or if the next item is the final coordinate tuple, no map switch is necessary. Otherwise, the handler reads the map identifier associated with the following graph node and obtains the localization mode required by that map from its YAML configuration.

The active localization mode is determined by inspecting the running ROS node list. The presence of `/loc` identifies the indoor `amcl` localization pipeline, while the presence of `/gps_amcl` identifies the outdoor or GNSS-aided `gps_amcl` pipeline. If both the active map and active localization mode already match the requirements of the following waypoint, *CHECK_MAP* returns `same_map`. If either differs, the handler calls the `/switch_map` service exposed by `MapSwitcher`. The service request contains the target map, the requested localization mode, and the pose of the waypoint that has just been completed. After a successful response, the handler updates its stored active-map identifier and returns `switched`. Both `same_map` and `switched` continue to *CHECK_PATH_CLEAR*; an unavailable localization mode, unreadable map configuration, or failed map-switch request produces `failed` and terminates the nested machine through `END`.

The *CHECK_PATH_CLEAR* state introduces a topological-level response to obstacles that prevent progression towards the following waypoint. It subscribes to the `/front_distance` topic published by the front-distance estimation node (Section 3.5.3). This topic provides a scalar estimate of the distance to obstacles within the forward LiDAR sector. A finite distance less than or equal to the configurable `~blocked_distance` threshold is interpreted as a blocked forward passage.

If the path is clear when the state is entered, execution proceeds immediately to *GET_NEXT_WAYPOINT*. If an obstruction is present, the state continues monitoring the front-distance value. Once an observed obstruction disappears, the path must remain clear for the configured `~clear_duration` before the state returns `clear`. This confirmation period reduces the likelihood that a short-lived or unstable sensor measurement causes the system to resume prematurely.

If the obstruction persists until the configured `~path_clear_timeout`, the handler assumes that the relevant passage is unavailable. When both the current waypoint and the following path item correspond to graph nodes, the state removes their bidirectional neighbour connection from the in-memory topological graph. It then returns `timeout`, which causes the nested machine to exit through the `BLOCKED` outcome. The outer state machine responds by returning to *PLAN_ROUTE*. A* is therefore executed again over the modified graph, allowing the system to search for an alternative route that avoids the connection identified as blocked.

The execution loop thus has the following general form:

1. Receive a destination and construct a topological path using A*.
2. Select the next waypoint, derive its pose and orientation, and dispatch it to `move_base`.

3. Retry the current waypoint after a pre-emption, terminate the current route after an aborted or rejected action, or cancel the action and re-plan when a new destination arrives.

4. After successfully reaching a waypoint, verify that the map and localization configuration are appropriate for the following route element and request a map switch when required.

5. Check whether the path ahead is clear before continuing; wait for a temporary obstruction to disappear or remove the relevant graph connection and re-plan when the blockage persists.

6. Repeat until the path is exhausted, then return to *WAIT_FOR_GOAL*.

By concentrating these responsibilities in a single hierarchical state machine, `move_base_handler` separates mission-level route management from local path execution. The topological graph provides the structure required for long-range and multi-map missions, `MapSwitcher` handles transitions between navigation contexts, `move_base` remains responsible for local motion between consecutive goals, and the front-distance check provides a simple mechanism for recovering from a passage that cannot be used. The sequence-level interaction between these components is presented in Section 3.6.1.

3.5.3 Front-Distance Estimation Node

The `front_distance` node provides a compact obstacle-proximity signal for the waypoint mission orchestrator. It subscribes to the robot's `/base_scan` LiDAR stream and publishes a single scalar measurement on `/front_distance`, representing the estimated distance to obstacles directly ahead of the robot. This reduces the full laser scan to a simple signal that can be used by higher-level mission logic without duplicating LiDAR-processing code inside the orchestrator.

For each incoming laser scan, the node considers only measurements within a configurable angular sector centred on the forward direction. In the current configuration, this sector extends $\pm 15^{\circ}$ around the robot's forward axis. Readings nearer the centre of the sector receive a larger weight than readings close to its boundaries. The resulting value is therefore a centre-weighted average front distance, rather than the distance measured by a single laser beam. This provides a more stable indication of whether the route immediately ahead is occupied while giving greater importance to obstacles lying directly on the robot's expected direction of travel.

Invalid scan readings are filtered before the estimate is calculated. `NaN` values and measurements below the sensor's minimum range are ignored, while positive infinite measurements are treated as the LiDAR maximum range. Measurements beyond the maximum range are similarly capped at that range. If no valid readings are available, or if the most recent scan is older than the configured timeout, the node publishes -1.0 to indicate that no valid front-distance estimate is currently available.

The estimate is published at a configurable fixed rate, independently of the LiDAR callback. This allows downstream nodes to consume a regular `/front_distance` signal while still detecting stale sensor data. In the waypoint mission orchestrator (Section 3.5.2), the

`CHECK_PATH_CLEAR` state uses this value to determine whether the passage towards the next waypoint is obstructed. A finite distance at or below the configured blocking threshold causes the orchestrator to wait; if the condition persists until the path-clear timeout, the corresponding topological connection is removed and the route is re-planned.

3.5.4 Map Server Handler (`MapSwitcher`)

The `MapSwitcher` node coordinates transitions between navigation maps and their associated localization configurations. It is used when the waypoint mission orchestrator determines that the next portion of a route belongs to another floor, outdoor area, or localization regime. Rather than requiring the mission orchestrator to manipulate the map server, localization nodes, initial pose topic, and navigation costmaps independently, `MapSwitcher` exposes these operations through a single `/switch_map` service.

The service request contains the identifier of the target world, the requested localization mode, and an initial pose in the target map frame. The target world identifies the directory and YAML file containing the map configuration. The requested localization mode is supplied by the orchestrator, while the node also reads the map YAML file to determine which localization mode is configured for that map. The currently active world and localization mode are stored internally. If a request asks for the same world and localization configuration that are already active, the node returns success without restarting any part of the navigation stack.

The node depends on two standard ROS services: `/change_map`, which is provided by the map server and accepts a map YAML file through the `nav_msgs/LoadMap` interface, and `/nav/clear_costmaps`, which clears the global and local `move_base` costmaps. Both services are awaited during node initialization so that map-switch requests are not accepted until the required navigation infrastructure is available. The node also creates a latched publisher on `/initialpose`. Latching is used so that the initial pose remains available until the newly started localization node subscribes to the topic.

Map-switch service callbacks do not perform the full switching sequence directly. Instead, the callback stores the incoming request and waits for a response produced by the main processing loop. The main loop runs at 10 Hz and invokes `process_requests`, which performs the potentially long-running operations involved in loading a map, restarting localization, and waiting for pose estimation. This separation keeps the service interface synchronous from the caller's perspective while centralizing the switching procedure in one execution path.

For a request that differs from the active configuration, `MapSwitcher` first resolves the path to the target map YAML file, located under the navigation package's `worlds` directory. It reads the `localization` entry from that file, using `amcl` as the default when no value is specified. The YAML file path is passed to `/change_map`. If the map server reports an error, the operation stops immediately and the service returns an unsuccessful response.

After the requested occupancy map has loaded successfully, the node shuts down the localization launch instance that was previously started by `MapSwitcher`, when one exists. It then starts the launch file corresponding to the localization mode read from the target map configuration:

`amcl.launch` for `amcl`, or `amcl_gps.launch` for `amcl_gps`. An unsupported localization string produces an error and causes the service request to fail. This launch-based approach allows the system to switch between indoor, map-relative localization and the GNSS-aided localization configuration used for outdoor areas.

Once the appropriate localization process has been started, the node waits until at least one subscriber is connected to `/initialpose`. It then creates and publishes a `geometry_msgs/PoseWithCovarianceStamped` message using the initial pose supplied in the service request. The message is expressed in the `map` frame and includes a fixed diagonal covariance: 0.05 for the x and y position terms, and 0.03 for the yaw term. These values provide the newly started localization component with a bounded initial pose hypothesis at the map transition point.

The node waits for a message on `/amcl_pose` for up to 10 seconds after publishing the initial pose. Receipt of such a message is treated as confirmation that the localization pipeline has initialized sufficiently to provide a pose estimate. If no message is received within this period, the service returns failure. When localization initializes successfully, the node calls `/nav/clear_costmaps`. Clearing the costmaps ensures that `move_base` rebuilds its global and local obstacle representations using the newly loaded map and the newly initialized pose estimate.

During elevator testing, an optional `~testing_elevator` parameter enables an additional simulation-specific action. When enabled, the node publishes an empty message on `/teleport_now` after stopping localization and before starting the target localization configuration. This mechanism supports simulated vertical transport by moving the robot between maps. It is explicitly a testing facility and is not required for the standard map-loading and localization-switching procedure.

The service sequence implemented by `MapSwitcher` can be summarized as follows:

1. Receive a `/switch_map` request containing a target world, requested localization mode, and initial pose.
2. Return success immediately when both the requested world and requested localization mode already match the internally recorded active configuration.
3. Read the requested world's YAML configuration and call `/change_map` to load the corresponding occupancy grid.
4. Shut down the previously launched localization process, if one is running.
5. Optionally publish `/teleport_now` when elevator-testing mode is enabled.
6. Start the localization launch file selected by the localization value stored in the target map's YAML configuration.
7. Wait for a subscriber on `/initialpose`, publish the supplied initial pose with its covariance, and wait for a pose estimate on `/amcl_pose`.

8. Clear the `move_base` costmaps, update the internally stored active world and localization mode, and return success.

By encapsulating this procedure, `MapSwitcher` presents map transition as one service-level operation to the waypoint mission orchestrator. A successful response means that the target map has been loaded, the selected localization pipeline has been started and seeded, and the navigation costmaps have been reset. A failure response prevents the orchestrator from continuing route execution under an uncertain map or localization configuration. The message-level interaction between `move_base_handler`, `MapSwitcher`, the map server, localization, and `move_base` is described further in Section 3.6.

3.5.5 Velocity Override Node

The velocity override node provides a controlled interface between the navigation stack and the low-level motion controller. Its main purpose is to allow the lower perception and guest-monitoring layer to temporarily take control of the robot's motion when the guest being guided is no longer visible in the rear camera. In that situation, continuing to follow the route without checking the guest's position would reduce the robot's ability to maintain an appropriate guiding interaction. The lower layer can therefore request that the robot stop and rotate towards the side in which the guest was last detected, allowing the rear camera to search for and reacquire the guest before normal navigation resumes.

The node is placed between `move_base` and the base controller. It subscribes to the velocity commands generated by the navigation stack on `cmd_vel` and publishes the final commands delivered to the base on `output_cmd_vel`. It also subscribes to `override_mode`, through which the lower layer selects the desired operating mode. This arrangement allows the navigation stack to continue operating normally while ensuring that only one source of motion commands is forwarded to the robot at any given time.

Internally, the node maintains an atomic control mode with four possible values: *PassThrough*, *Stop*, *RotateLeft*, and *RotateRight*. The mode is encoded as an unsigned integer on the `override_mode` topic. It is stored using a C++ `std::atomic` variable because the ROS callback that receives a new override request and the timer callback that publishes replacement commands may execute independently. Atomic updates ensure that both callbacks observe a consistent control mode.

The default operating mode is *PassThrough*. In this mode, each command received on `cmd_vel` is republished unchanged on `output_cmd_vel`. The node neither changes the linear or angular velocity values nor publishes additional timer-driven commands. Consequently, under normal conditions, the robot follows the velocity commands computed by `move_base` without interference.

When the guest-monitoring layer determines that an intervention is required, it publishes an override command. In *Stop* mode, incoming `cmd_vel` commands are no longer forwarded and the node repeatedly publishes a zero-valued `geometry_msgs/Twist` message at the configured

~publish_rate. This first halts the robot's current motion. The lower layer can then select *RotateLeft* or *RotateRight*, causing the node to publish a zero-linear-velocity command with a constant angular velocity. The magnitude of the rotation is configured through ~rotation_speed; its sign determines the rotation direction.

The intended guest-recovery sequence is therefore:

1. The rear-camera or guest-monitoring layer determines that the guest is no longer visible.
2. The lower layer publishes *Stop* on override_mode, preventing further move_base commands from reaching the base and bringing the robot to a halt.
3. Using the last known position or direction of the guest, the lower layer selects *RotateLeft* or *RotateRight*.
4. The override node publishes the corresponding constant rotation command until the guest is reacquired or another mode is selected.
5. Once the guest is visible again, the lower layer publishes *PassThrough*, restoring the normal forwarding of move_base commands.

The node validates each received override command before applying it. Commands corresponding to the four supported modes replace the active mode. Any other value is ignored and produces a ROS warning, leaving the previously selected mode unchanged. This prevents an invalid message from placing the node in an undefined control state.

Although the node is primarily intended to support guest reacquisition and interaction-aware navigation, it also provides a useful safety property. During an override, navigation commands are not combined with stop or rotation commands; instead, the override node acts as the single source of the final velocity command. This avoids conflicting motion requests and ensures that the robot stops before performing a deliberate guest-search rotation.

3.5.6 Depth Image to LaserScan Conversion

The depth-image-to-LaserScan node converts dense depth information generated by an OAK-D stereo camera into a two-dimensional range representation compatible with the ROS navigation stack. Its purpose is to make obstacles detected by the depth cameras available to components that expect conventional planar LiDAR data, including the localization pipeline and the layered costmaps used by move_base. Each node instance receives a depth image and produces a sensor_msgs/LaserScan message expressed in the robot base_link frame.

The node receives two camera data streams: the stereo depth image published on /oak/stereo/image_raw, and the associated camera calibration data published on /oak/stereo/camera_i
The calibration message provides the intrinsic camera parameters f_x, f_y, c_x, and c_y, which are required to reconstruct three-dimensional points from valid depth pixels. The node waits until camera information has been received before processing incoming depth images.

For a depth pixel at image coordinates (u, v), with depth value z, the corresponding point in the camera frame is reconstructed using the pinhole-camera model:

Depth-to-3D coordinate transformation

$$x = \frac{(u - c_x)z}{f_x}, \qquad y = \frac{(v - c_y)z}{f_y}, \qquad z = z. \tag{3.1}$$

When the depth image is received in a 16-bit unsigned format, its values are converted from millimetres to metres before the point reconstruction step. Invalid depth readings, non-finite values, and measurements below the minimum usable depth are discarded. To reduce computational cost, the node subsamples the image by processing every second pixel in both directions. This produces a sufficiently dense point cloud for navigation purposes while avoiding the cost of reconstructing every pixel in a full-resolution depth image.

The reconstructed cloud is initially expressed in the camera optical frame. The node then uses TF to obtain the transform between the camera frame and `base_link` at the timestamp of the depth image. The cloud is transformed into `base_link` before any obstacle reasoning is performed, ensuring that the resulting scan is expressed in a common robot-centred frame. If the required transform is not available, the current depth frame is discarded rather than publishing range data in an inconsistent frame.

A depth image contains both the ground surface and objects that should be treated as navigation obstacles. The node therefore identifies the floor before projecting the cloud into a two-dimensional scan. It applies RANSAC plane segmentation to the point cloud in the `base_link` frame. Candidate planes are extracted iteratively until a sufficiently horizontal plane is found. A plane is accepted as the floor when its normal vector is strongly aligned with the vertical axis. In the current implementation, the required cosine similarity is 0.97, corresponding approximately to a maximum deviation of 14° from horizontal.

Once the floor plane has been identified, each point is evaluated according to its perpendicular distance from that plane. Points that are too close to the floor are rejected, as are points above the configured obstacle height range. Only points satisfying

$$h_{\min} \leq h \leq h_{\max} \tag{3.2}$$

are considered obstacle candidates. The current default limits are $h_{\min} = 0.05\,\text{m}$ and $h_{\max} = 0.50\,\text{m}$. This suppresses floor measurements and retains objects that protrude into the robot's navigable space at heights relevant to collision avoidance.

The remaining obstacle candidates are filtered by planar distance from the robot. Points closer than `~min_radius` or farther than `~max_radius` are removed. The default limits are 0.3 m and 11.0 m, respectively. The lower bound reduces the effect of unreliable near-camera depth readings and prevents parts of the robot from being interpreted as obstacles. The upper bound restricts the output to distances considered useful for the depth sensor and the local navigation task.

The filtered points are projected into the horizontal plane and converted into angular range bins. For each point, its planar distance and bearing are calculated as

$$r = \sqrt{x^2 + y^2}, \qquad \theta = \operatorname{atan2}(y, x). \tag{3.3}$$

The output scan covers the interval from $-\pi$ to $+\pi$ radians around the robot. Each point is assigned to an angular bin according to the configured angular resolution, which is 1° by default. When several points fall into the same bin, only the closest point is kept. This reproduces the visibility model used by a conventional planar LiDAR, where the nearest object in a direction is the relevant obstacle for navigation. Bins without a valid obstacle measurement remain set to infinity.

The resulting `LaserScan` message is published with its frame set to `base_link`, a range interval from 0.1 m to 10.0 m, and the configured angular increment. Although an individual OAK-D camera does not observe the full 360° region represented by the output message, its unobserved bins remain infinite. This allows the scan to be combined cleanly with measurements from other sensors without inventing obstacle information in directions outside the camera field of view.

The navigation layer runs separate instances of this node for the front and rear OAK-D cameras. Each instance publishes initially on the node's default `/scan` interface, but launch-file remapping assigns the resulting streams to distinct topics. The front camera output is remapped to a front depth-derived scan topic, while the rear camera output is remapped to a rear depth-derived scan topic. This prevents the two camera instances from publishing indistinguishable data to the same topic and preserves the identity of each observation source.

The front and rear depth-derived LaserScan streams are subsequently combined with the planar RP-LiDAR scan by the LiDAR merger node. The merger publishes the unified `/base_scan` topic consumed by the localization pipeline and by the global and local costmaps of `move_base`. This arrangement provides a common obstacle representation for the navigation stack: the planar LiDAR supplies conventional range measurements around the base, while the depth cameras contribute obstacle information from their respective fields of view, including obstacles that may not intersect the LiDAR scanning plane.

The depth-image-to-LaserScan node therefore acts as a compatibility layer between the three-dimensional sensing capability of the OAK-D cameras and the two-dimensional LaserScan interface expected by the existing ROS 1 navigation stack. It allows depth-derived obstacle information to be integrated into the established Intellwheels navigation pipeline without modifying AMCL, the costmaps, or `move_base`. The separate front and rear instances can be configured, remapped, enabled, or retuned independently, while their merged output remains transparent to the navigation components that consume `/base_scan`.

3.6 Navigation Execution Flows

The previous sections described the main navigation components as structural elements of the navigation layer. During operation, however, the behaviour of the Host Robot emerges from the way these components interact over time during mission execution, map transitions, and temporary safety overrides. This section therefore presents the main execution flows of the system, with reference to the `move_base_handler` state machine, the waypoint mission orchestrator sequence, the `MapSwitcher` sequence, and the velocity override sequence. Together, these flows show how high-level goals are transformed into executable motion and how exceptional situations are handled.

3.6.1 Waypoint Mission Orchestrator Flow

The waypoint mission orchestrator coordinates the transition from a high-level destination request to a sequence of local navigation goals. It connects the decision layer, the topological graph planner, the `move_base` action server, the map-switching mechanism, and the front-distance monitoring node.

Figure 3.12 presents the complete process at a high level. To preserve readability, the detailed interactions are separated into four additional diagrams. Goal reception and A*-based route generation are shown in Figure 3.13. Waypoint dispatch and the handling of `move_base` action outcomes are shown in Figure 3.14. The map and localization context check performed by the orchestrator is shown in Figure 3.15. Finally, forward-path monitoring and topological re-planning are shown in Figure 3.16.

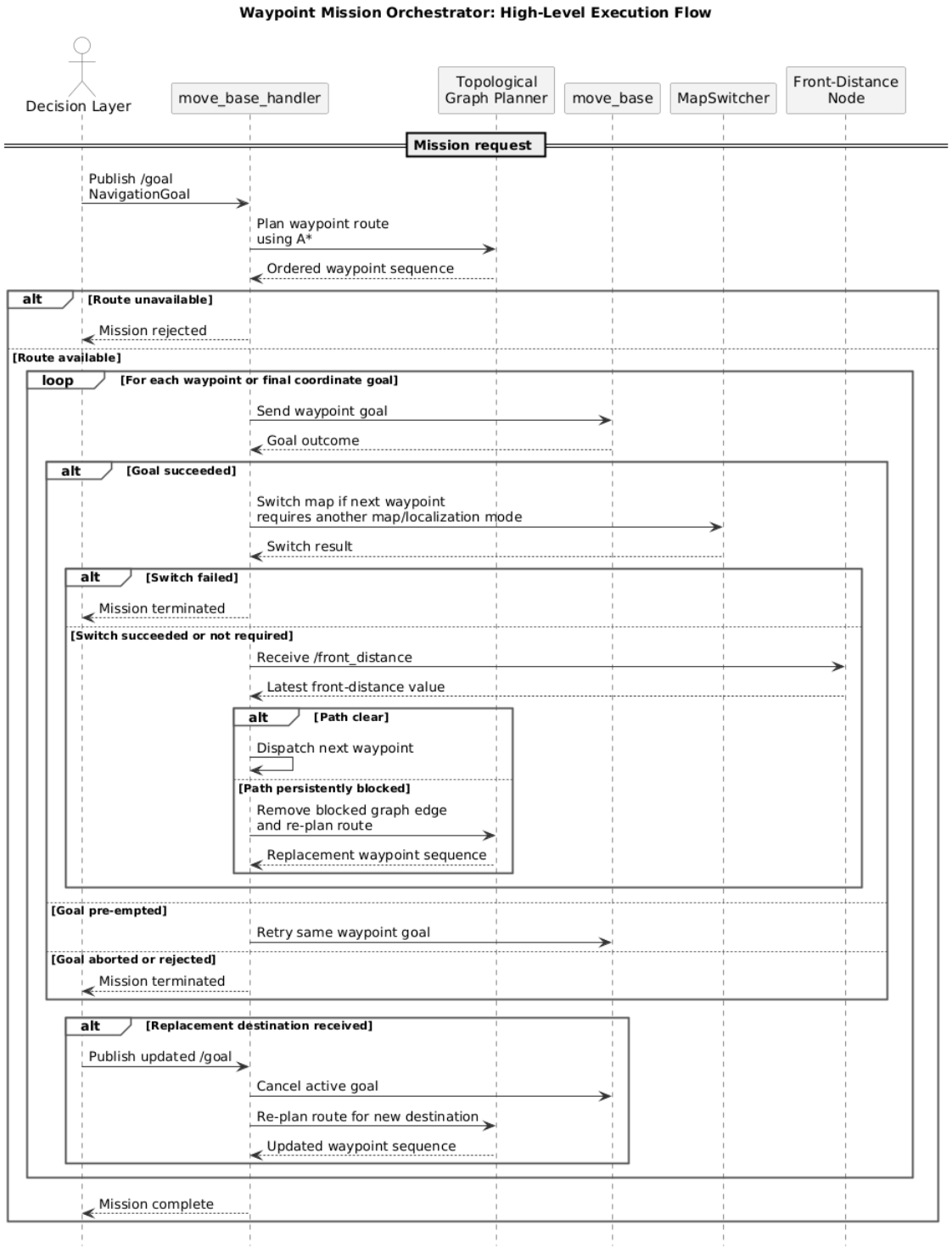


Figure 3.12: High-level execution flow of the waypoint mission orchestrator. A destination is converted into an A*-planned waypoint sequence, whose elements are dispatched individually to `move_base`. Successful waypoint execution may trigger a map-context check and a forward-path check, while persistent blockages and replacement goals cause the route to be re-planned.

The interaction begins when the decision layer publishes a destination through `/goal`. The mission handler stores the request and obtains the current robot position from the TF transform between `map` and `base_link`. It then identifies the graph nodes closest to the current position and the requested destination and asks the topological graph planner to calculate a route using A*. The route consists of graph waypoint identifiers followed by the final coordinate goal. If no route is available, the mission is rejected and the handler returns to its waiting state.

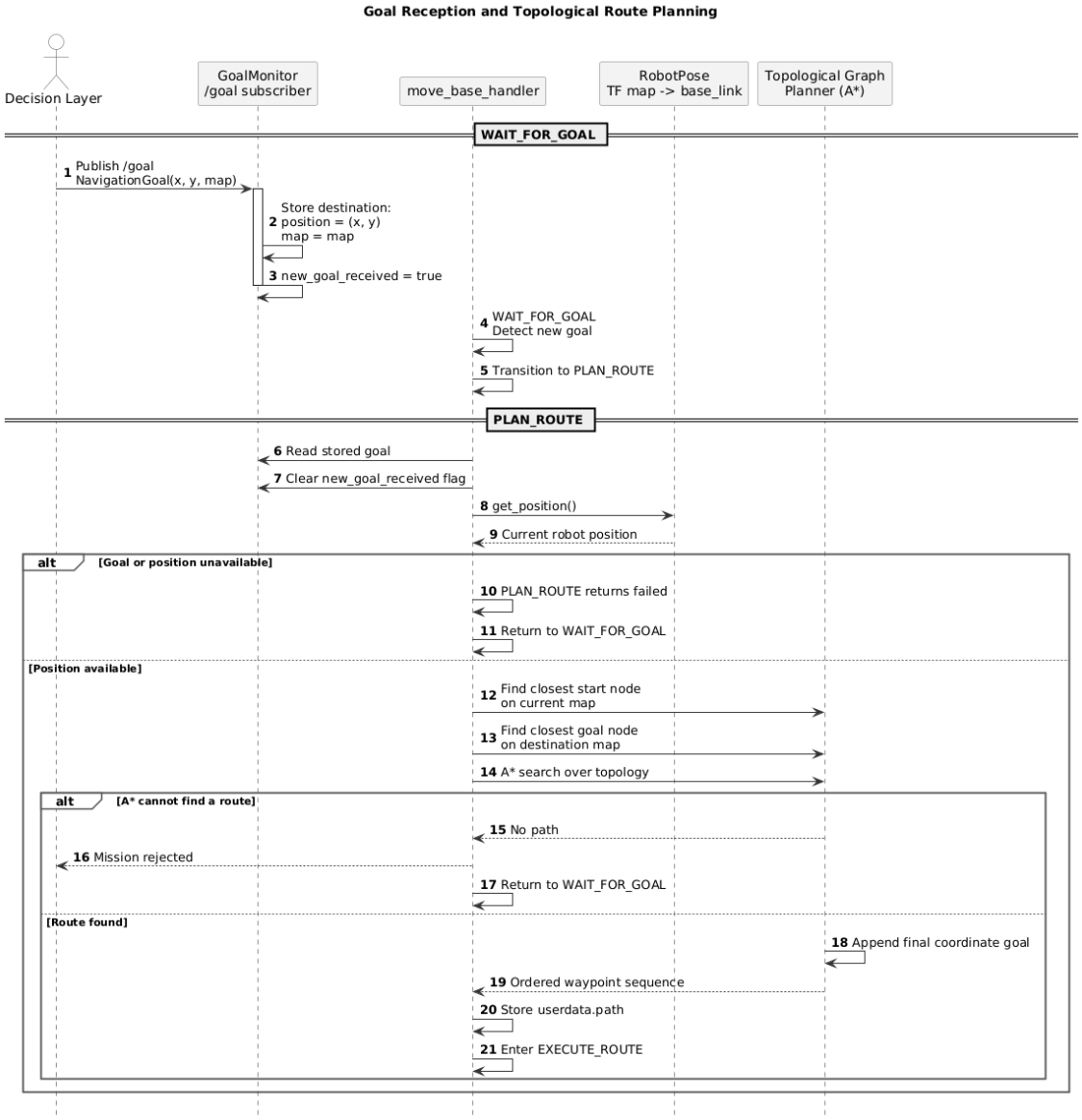


Figure 3.13: Goal reception and topological route planning. The mission handler receives a `NavigationGoal` message, obtains the robot pose from TF, selects the closest graph nodes on the relevant maps, and uses A* to generate an ordered waypoint sequence.

Once a route has been generated, the nested execution state machine processes its elements one at a time. The STOPPED state checks whether a replacement destination has already been received. If not, the GET_NEXT_WAYPOINT state removes the next route element and converts it into a pose goal. For graph waypoints, the orientation is calculated from the current waypoint towards the following route element. The resulting goal is then sent to the `move_base` action server.

Figure 3.14 shows this waypoint-dispatch process and the possible `move_base` outcomes. A successful goal leads to the CHECK_MAP state. A pre-empted goal is retried, while an aborted or rejected goal terminates the current mission. If a new destination is received while a goal is active, the handler cancels the current action and returns to route planning for the replacement destination.

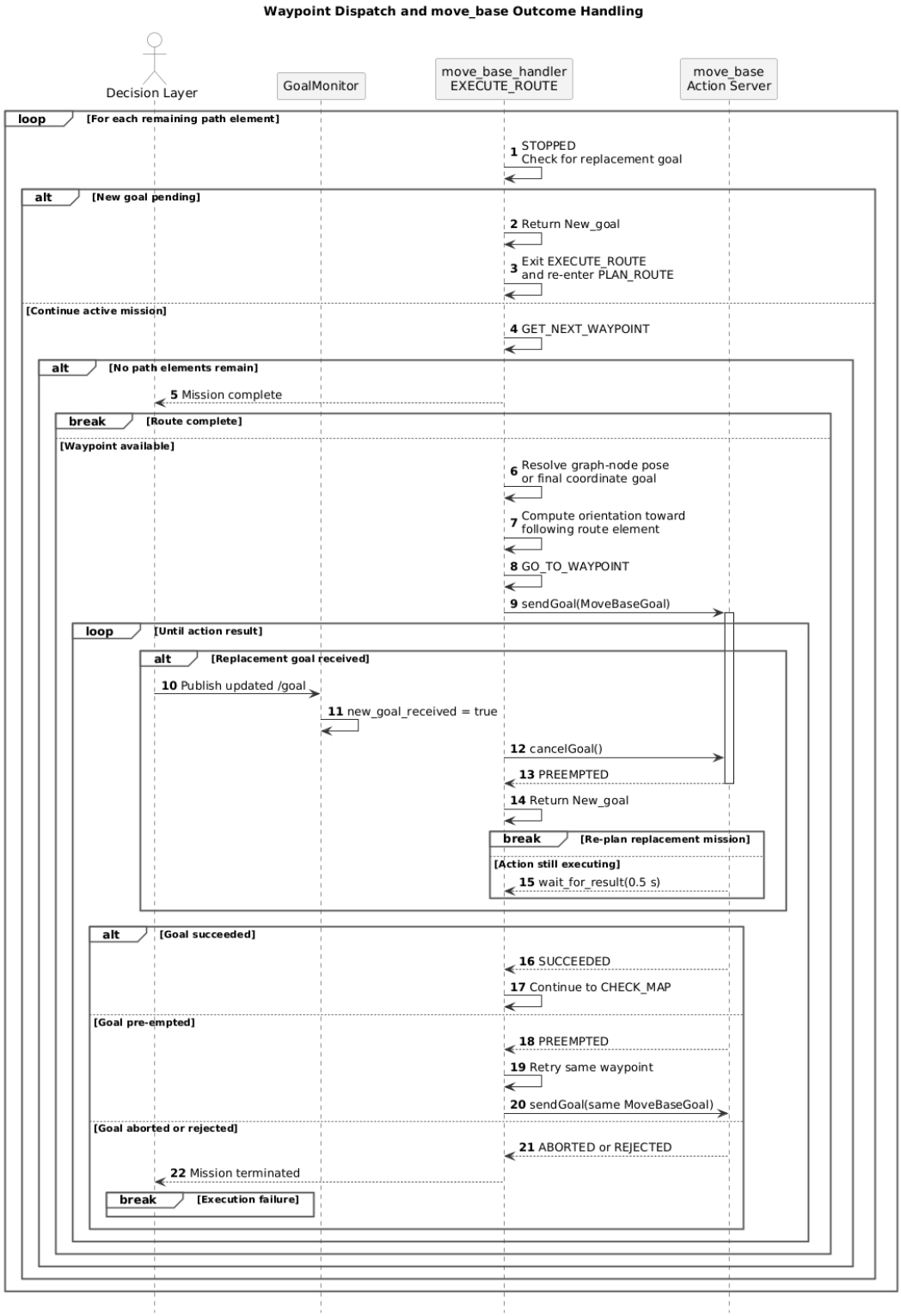


Figure 3.14: Waypoint dispatch and `move_base` outcome handling. The execution state machine selects each route element, converts it into a `MoveBaseGoal`, monitors the action result, retries pre-empted goals, and cancels the active goal when a replacement destination is received.

After a waypoint has been reached, the CHECK_MAP state determines whether the next graph waypoint requires a different map or localization configuration. The handler compares the currently active map and localization process with the configuration associated with the next waypoint. If the context is already correct, execution proceeds without a switch. Otherwise, the handler calls the `/switch_map` service exposed by `MapSwitcher`.

The requested transition includes the target map, the required localization mode, and an initial pose based on the waypoint that has just been reached. A successful service response allows the handler to update its active-map record and continue execution. If the switch fails, the mission is terminated. Figure 3.15 shows the map and localization context check performed by `move_base_handler`. The detailed internal sequence executed by `MapSwitcher` after receiving the service request is presented separately in Section 3.6.2.

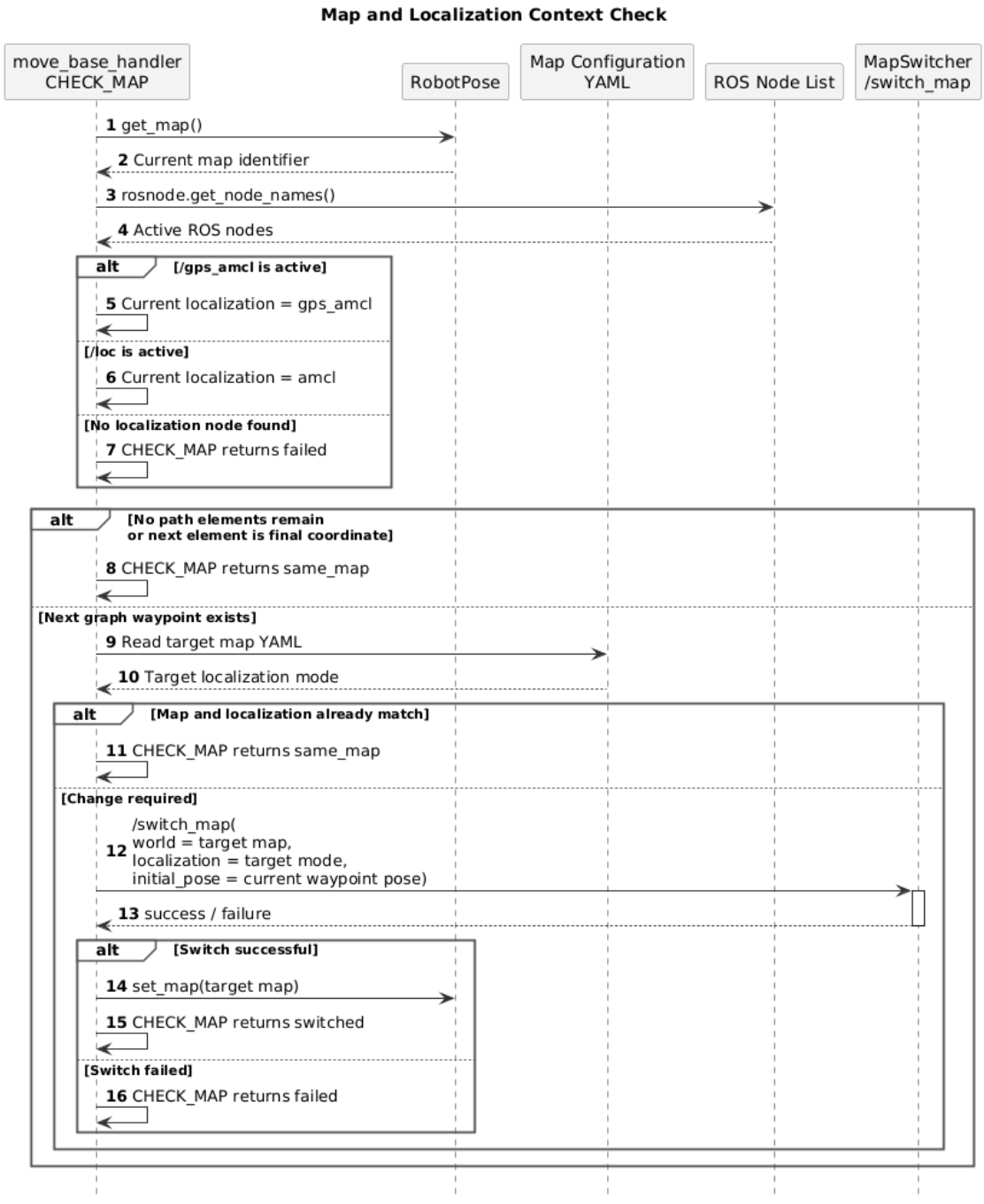


Figure 3.15: Map and localization context check performed by the waypoint mission orchestrator. After a successful waypoint, the handler compares the active map and localization mode with those required by the following waypoint and requests `/switch_map` when a transition is needed.

After the map check, the CHECK_PATH_CLEAR state evaluates the most recent value received on `/front_distance`. This value is provided asynchronously by the front-distance node. If the path is initially clear, the handler immediately proceeds to the next waypoint.

When the forward path is blocked, the handler waits while monitoring the distance measurement. Once the blockage disappears, the path must remain clear for the configured confirmation interval before execution resumes. If the blockage persists until the path-clear timeout expires, the handler removes the bidirectional connection between the current waypoint and the following waypoint from the in-memory topological graph. It then returns to PLAN_ROUTE, where A* searches for an alternative route using the updated graph.

This response is local to the current mission. The graph modification is held in memory and allows the active route to avoid the unavailable connection. Long-term learning from repeated blockages across different missions is considered separately as future work.

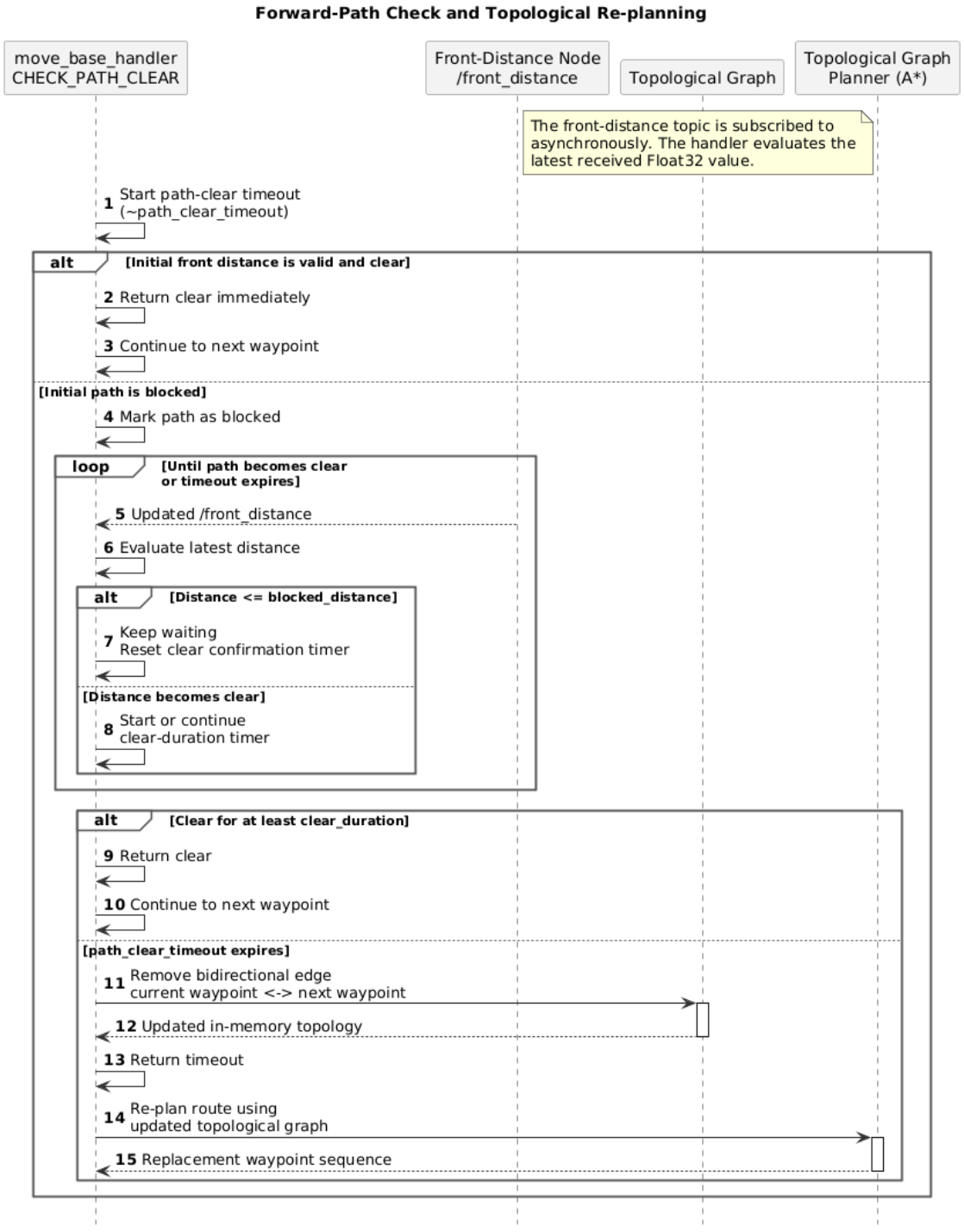


Figure 3.16: Forward-path handling and topological re-planning. The handler monitors the latest `/front_distance` value, waits for temporary obstructions to clear, and removes the blocked graph connection when the configured timeout expires. A* then computes a replacement route over the updated in-memory topology.

The execution cycle continues until all route elements have been processed. When the path is exhausted, the nested execution state machine returns END, the mission is reported as complete, and the outer state machine returns to WAIT_FOR_GOAL. It can then accept a new destination.

Together, Figures 3.12, 3.13, 3.14, 3.15, and 3.16 describe the complete mission-level behaviour of `move_base_handler`. The overview diagram presents the relationship between the main components, while the detailed diagrams explain route generation, waypoint execution, map-context checking, and blocked-path recovery.

3.6.2 Map Switching Sequence

The mission execution flow delegates the implementation of map transitions to the `MapSwitcher` node. Whereas Figure 3.15 shows the decision made by `move_base_handler` to request a transition, the present subsection describes the operations performed internally by `MapSwitcher` after it receives the `/switch_map` request.

Figure 3.17 shows the detailed interaction between `move_base_handler`, `MapSwitcher`, the map server, the active localization subsystem, the initial-pose topic, and the `move_base` costmap-reset service.

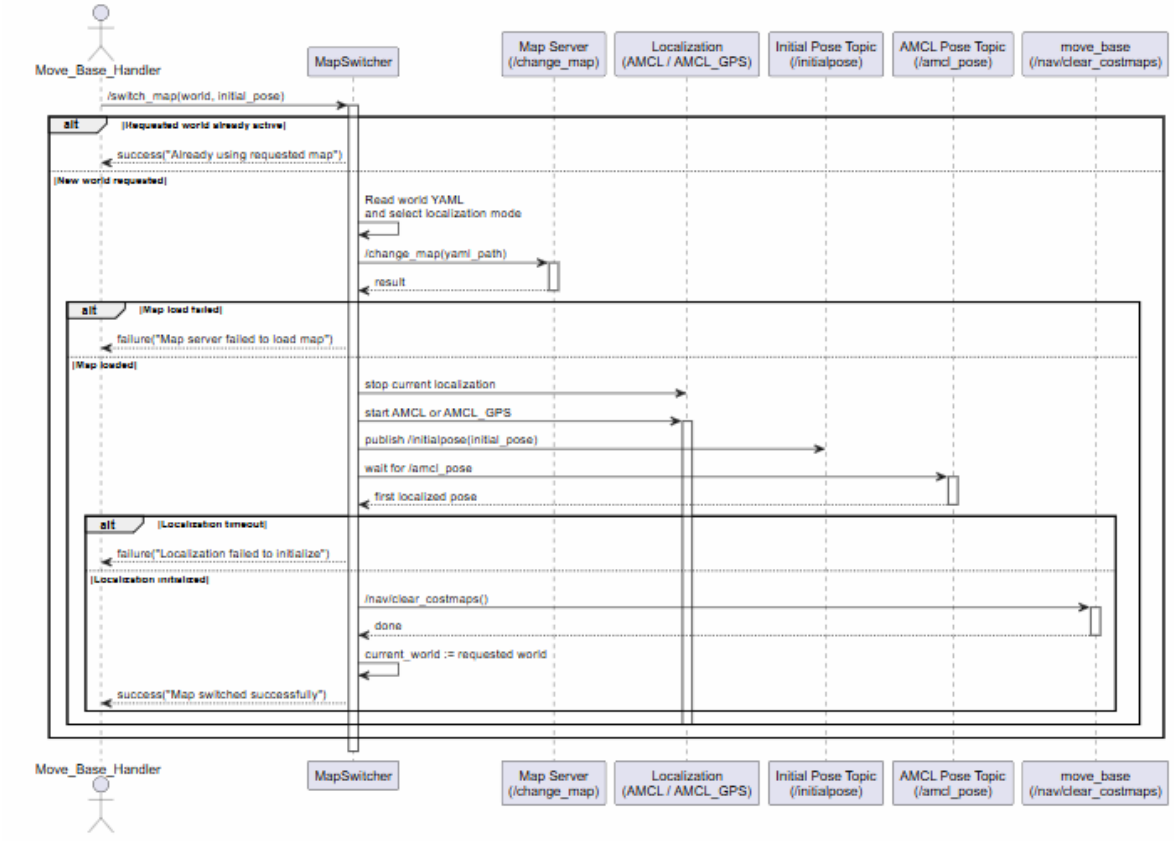


Figure 3.17: Map switching sequence coordinated by `MapSwitcher`. A requested world change triggers map loading, localization restart, initial pose publication, localization confirmation, and a costmap reset before control returns to the mission orchestrator.

A map switch is initiated when CHECK_MAP determines that the next graph waypoint belongs to a different world or requires another localization configuration. The mission handler calls the `/switch_map` service, passing the target world, the requested localization mode, and an initial pose derived from the waypoint that has just been reached. If the requested world and localization configuration are already active, `MapSwitcher` returns success without restarting any navigation components.

When a new world is requested, `MapSwitcher` reads the corresponding YAML configuration, determines the localization mode associated with the target map, and calls the standard `/change_map` service on the map server. This loads the occupancy grid required by the target navigation context. If the map server cannot load the requested map, the switching procedure stops and the service returns failure to the mission orchestrator.

After loading the map, `MapSwitcher` stops the currently active localization pipeline and starts the configuration required by the target world. Indoor maps use standard AMCL, whereas outdoor maps use the GNSS-aided `gps_amcl` configuration. The node then publishes the requested pose on `/initialpose` and waits for a pose estimate on `/amcl_pose`. This confirms that the newly started localization process has initialized sufficiently to provide a robot pose on the target map.

If no localization estimate is received within the configured timeout, the switch is treated as unsuccessful. When localization succeeds, `MapSwitcher` calls `/nav/clear_costmaps` so that the global and local costmaps are rebuilt for the new map and pose estimate. Only after this reset succeeds does the node update its internal record of the active world and return success to the mission handler.

From the point of view of `move_base_handler`, this multi-step operation is exposed as a single synchronous service with either a success or failure outcome. This abstraction allows the mission orchestrator to treat map transitions as atomic checkpoints in the waypoint sequence rather than having to coordinate the map server, localization processes, initial-pose publication, and costmaps directly.

3.6.3 Override node Flow

The final execution flow concerns temporary safety or supervisory intervention in the velocity stream produced by the navigation stack. Figure 3.18 shows the sequence followed by the velocity override node when receiving normal `cmd_vel` messages from `move_base` and override mode commands from the lower layers.

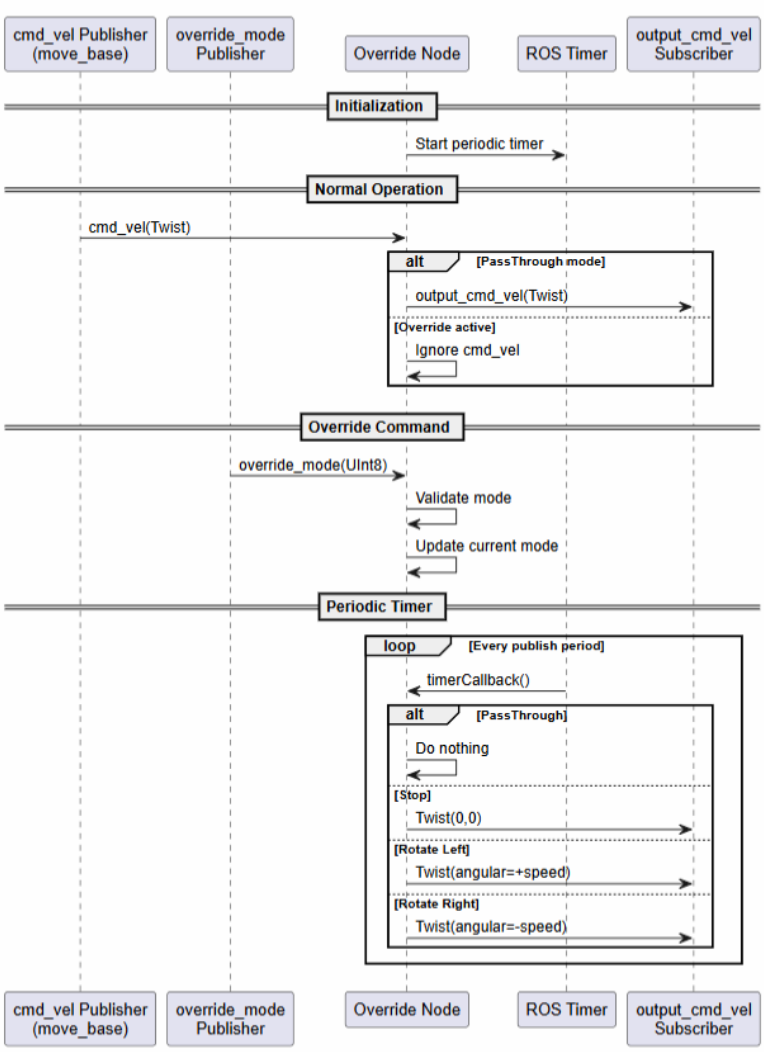


Figure 3.18: Velocity override sequence. In normal operation, `cmd_vel` messages from `move_base` are forwarded unchanged. When an override mode is active, the node ignores incoming planner commands and publishes stop or rotation commands at a fixed rate.

The override node starts in *PassThrough* mode and initializes a periodic timer that is later used to publish override commands when required. During normal autonomous operation, every Twist message received from `move_base` on `cmd_vel` is forwarded directly to the low-level motion controller through the output command topic, and the timer callback performs no action. In this state, the node behaves as a transparent relay between the navigation stack and the wheelchair controller.

When the lower control or safety layer requests an override, it publishes a discrete mode value corresponding to one of the supported modes: *PassThrough*, *Stop*, *RotateLeft*, or *RotateRight*. Upon receiving this command, the override node validates the mode and updates its internal state. From that instant onward, the behaviour of the node changes according to the selected mode.

If the mode remains *PassThrough*, the node continues forwarding `move_base` commands unchanged. If the mode is anything else, incoming `cmd_vel` messages are ignored and the periodic

timer becomes the source of the commands actually sent to the base. In *Stop* mode, the timer publishes a zero-velocity Twist at a fixed rate, forcing the robot to remain stationary regardless of what the navigation stack is doing. In *RotateLeft* and *RotateRight* modes, the timer publishes a constant angular velocity of configurable magnitude and sign, causing the robot to rotate in place until the override is cleared.

This mechanism provides a simple but robust recovery path for short-term situations in which the robot's nominal autonomous behaviour must be temporarily suppressed, for example because a safety condition requires an immediate halt or because the robot must rotate to reacquire a guest in the rear camera field of view. Importantly, this intervention happens downstream of `move_base`, so neither the planner nor the waypoint mission orchestrator needs to change state when an override is applied. Once the override mode returns to *PassThrough*, the node resumes relaying `move_base` commands and nominal autonomous navigation continues.

3.7 Implementation Overview

3.7.1 ROS Graph and Deployment

The implemented system is organised as a ROS 1 graph in which simulation, navigation, map management, localization, sensor fusion, and velocity-command handling are encapsulated in separate nodes connected through standard topics and services. Figure 3.19 shows the graph obtained with `rqt_graph` during a representative navigation experiment.

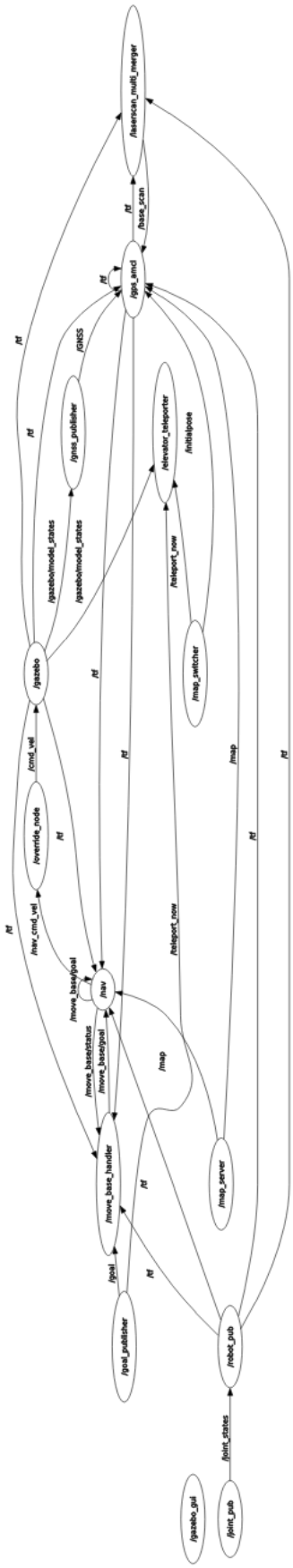


Figure 3.19: ROS graph of the navigation-layer implementation in simulation. The graph shows the interaction between the Gazebo simulation, waypoint mission handler, navigation stack, GNSS-aided localization, map-switching components, range-scan merger, and velocity override node.

The graph is divided conceptually into several functional groups. On the left side, `/gazebo` provides the simulated environment and publishes the robot's simulated state through `/gazebo/model_states`. The simulation also provides the sensor and robot-state information required by the navigation pipeline. The `/robot_pub` and `/joint_pub` nodes publish the robot model and joint-state information, while `/gui` provides a simulation or visualization interface.

Mission execution begins with `/goal_publisher`, which publishes a destination request on `/goal`. This node represents the high-level decision layer during the experiments. The request is received by `/move_base_handler`, which determines the sequence of waypoints required to reach the destination and sends the individual waypoint goals to the `/nav` node. The handler also monitors navigation feedback and coordinates map transitions when the next waypoint belongs to a different navigation context.

The `/nav` node represents the `move_base` navigation stack. It receives waypoint goals through the `/move_base/goal` interface and publishes navigation status and result information back to `/move_base_handler`. The stack also consumes the active map, odometry, localization output, and the merged range data required by its global and local costmaps. The velocity commands generated by the navigation stack are published on `/nav_cmd_vel`.

The range-data branch is represented in the graph by the `/laserscan_multi_merger` node. This component produces the merged `/base_scan` topic used by the localization and navigation components. The merged scan represents the range information available to the navigation stack after the individual sensor streams have been combined. The `/base_scan` topic is consumed by `/gps_amcl` and by the navigation costmaps. The graph therefore exposes the merged range interface, rather than showing every individual sensor-processing stage in detail.

The active localization process is represented by `/gps_amcl` in the shown configuration. It receives the active map from `/map_server`, odometry from the state-estimation pipeline, GNSS information from `/gnss_publisher`, and the merged range data from `/base_scan`. The GNSS publisher provides the simulated GNSS measurements used by the outdoor localization configuration. Depending on the active map, the broader system can use either conventional AMCL or the GNSS-aided `gps_amcl` configuration.

Map transitions are coordinated by `/map_switcher`. The `/move_base_handler` requests a transition when the next waypoint belongs to another map or navigation context. MapSwitcher communicates with the `/map_server`, publishes an updated pose through `/initialpose`, and coordinates the transition of the active localization process. During simulated elevator experiments, it also communicates with `/elevator_teleporter` through the `/teleport_now` topic. The teleporter uses this event to move the simulated robot from the elevator area on one map to the corresponding location on the target map.

The final command path is handled by `/override_node`. Under normal conditions, the node receives the velocity commands generated by `/nav` on `/nav_cmd_vel` and forwards them to the final `/cmd_vel` topic. When an override mode is selected, the node replaces the navigation-generated command with a stop or rotation command. This allows temporary safety or guest-reacquisition behaviours to take control of the robot without modifying the internal operation of

`move_base`.

Overall, the graph shows the main information flow of the implemented navigation layer:

$$\texttt{/goal} \rightarrow \texttt{/move_base_handler} \rightarrow \texttt{/move_base/goal} \rightarrow$$

$$\rightarrow \texttt{/nav} \rightarrow \texttt{/nav_cmd_vel} \rightarrow \texttt{/override_node} \rightarrow \texttt{/cmd_vel}.$$

In parallel, map, odometry, GNSS, and merged range data provide the information required by localization and navigation:

$$\texttt{/map} + \texttt{/odom} + \texttt{/gnss} + \texttt{/base_scan} \rightarrow \texttt{/gps_amcl} \rightarrow \texttt{/tf}.$$

This deployment preserves the modular structure of the navigation layer. Simulation, mission management, localization, map switching, range fusion, navigation, and command override are implemented as separate components and communicate through explicit ROS interfaces.

3.7.2 Node Interfaces and Topics

Table 3.1 summarises the main nodes and interfaces visible in the implementation graph. The table focuses on the components directly involved in navigation execution. Auxiliary visualization and debugging nodes are included only where they clarify the relationship between the simulation and the navigation stack.

Table 3.1: Main ROS nodes and topics in the navigation-layer implementation.

Node	Role	Main subscribed / published topics
`/gazebo`	Simulation backend	Publishes simulated robot state through `/gazebo/model_states` and provides the simulated sensor and environment data consumed by the navigation system.
`/robot_pub`	Robot-state publisher	Publishes robot model and transform information required to represent the simulated robot and its coordinate frames.
`/joint_pub`	Joint-state publisher	Publishes joint-state information for the simulated robot model.
`/goal_publisher`	Mission input	Publishes high-level destination requests on `/goal` to emulate the ROS 2 decision layer during experiments.
`/move_base_handler`	Waypoint mission orchestrator	Subscribes to destination requests on `/goal` and to navigation feedback; computes waypoint routes, sends successive goals through the `move_base` action interface, and requests map changes from `/map_switcher`.

Continued on the next page

Node	Role	Main subscribed / published topics
/nav	Navigation stack	Represents the move_base stack. Receives waypoint goals through the move_base action interface, consumes /map, /odom, and /base_scan, and publishes navigation velocity commands on /nav_cmd_vel.
/map_server	Map provider	Publishes the active occupancy grid on /map and responds to map-change requests issued by /map_switcher.
/gps_amcl	GNSS-aided localization	Consumes the active map, odometry, GNSS data from /gnss_publisher, and merged range data from /base_scan; publishes the estimated robot pose through /tf and localization pose topics.
/gnss_publisher	GNSS data source	Publishes simulated GNSS measurements on /gnss, which are consumed by the GNSS-aided localization configuration.
/laserscan_multi_merger	Range-scan fusion	Combines the available individual range-scan inputs and publishes the unified /base_scan topic used by localization and the navigation costmaps.
/map_switcher	Map management	Receives map-transition requests from /move_base_handler; coordinates map-server changes, localization transitions, and initial-pose publication on /initialpose.
/elevator_teleporter	Floor-change emulator	Subscribes to /gazebo/model_states and /teleport_now; changes the simulated robot pose when a virtual elevator transition is requested.
/override_node	Velocity-command override	Subscribes to /nav_cmd_vel and override-control messages; publishes the final velocity command on /cmd_vel towards the execution layer.

Together, these nodes implement the navigation architecture described in Chapter 3. The graph separates mission-level route management from map and localization management, local navigation, range-data fusion, and final command selection. This separation allows the waypoint mission handler and MapSwitcher to coordinate long-range and multi-map missions while move_base remains responsible for point-to-point motion in the active navigation context. The range merger provides a common /base_scan interface, and the override node provides a final command-selection layer through which safety and guest-reacquisition behaviours can temporarily supersede autonomous navigation.

3.8 Design Rationale and Key Decisions

3.8.1 Architectural Rationale

The architecture presented in Section 3.3 is shaped by three main constraints: the characteristics of the Host Robot environment, the capabilities of the existing Intellwheels baseline, and the requirement to operate across ROS 1 and ROS 2 without discarding validated components. Large, multi-floor public institutions demand navigation that can span long missions, traverse elevators and outdoor segments, and remain safe in crowded spaces, but the available platform is a wheelchair-based robot whose navigation stack is already built around the standard ROS 1 `move_base` node. Rather than replacing this baseline, the proposed architecture keeps `move_base` as the single-map navigation engine and introduces additional layers around it to address the missing capabilities.

This leads directly to a layered design in which responsibilities are separated by spatial and temporal scale. At the lowest level, `move_base` operates on one occupancy grid at a time, using its global and local planners and costmaps to convert a goal pose into a stream of feasible velocity commands. Above it, a waypoint-based mission layer represents the building as a sparse graph of meaningful locations and sequences these waypoints to implement long-range missions across multiple maps, floors, and indoor–outdoor transitions. Orthogonal to both, dedicated modules manage map loading and localization mode (AMCL or `gps_amcl`), fuse odometry and GNSS, and provide a merged laser scan from LiDAR and depth cameras, so that the navigation engine always sees a consistent map, pose estimate, and obstacle field regardless of where the robot is in the institution.

A further rationale for this architecture is maintainability and reuse. By keeping the interface between layers narrow—goals and feedback between the decision layer and the waypoint orchestrator, odometry and `LaserScan` data into localization and costmaps, velocity commands and overrides between navigation and the execution layer—each component can be modified or replaced with limited impact on the others. This modularity is particularly important given the ROS 1/ROS 2 split: the navigation logic remains in ROS 1 alongside Intellwheels, while the decision and low-level control layers run in ROS 2, communicating through ROS 1–ROS 2 bridges that transport only the essential topics and services. The resulting architecture satisfies the functional and non-functional requirements identified earlier—waypoint-based missions, robust map transitions, indoor–outdoor localization, safety-aware motion—by composing existing, well-understood building blocks rather than introducing an entirely new navigation stack.

3.8.2 Key Design Decisions

Several concrete design choices follow from this rationale and are central to how the navigation layer behaves in practice.

First, the decision to retain `move_base` as the core navigation engine was deliberate. The Intellwheels baseline already provided a tuned configuration of global and local planners, costmaps, and recovery behaviours for the wheelchair platform, and replacing it with a different stack would

have required repeating much of that validation work. By treating `move_base` as a black box that solves "single-goal on a single map" navigation, the dissertation concentrates on the missing outer structure: sequencing multiple goals, managing map transitions, and integrating additional sensors and localisation modes.

Second, long missions are expressed in terms of waypoints rather than as raw metric goals. Representing the environment as a waypoint graph allows missions to be planned over a small set of semantically meaningful nodes—door approaches, corridor junctions, elevator lobbies—instead of over every cell in a grid map. This choice simplifies mission-level planning, makes the routing problem more interpretable, and allows the mission layer to attach special behaviour to particular waypoint types (such as waiting at elevators) without complicating the underlying planners. At the same time, the detailed motion between consecutive waypoints is left entirely to `move_base`, which continues to use its global and local costmaps to navigate around local obstacles.

Third, multi-floor and indoor–outdoor operation are handled through map-aware waypoints and an explicit Map Server Handler rather than by extending `move_base` to operate on a monolithic multi-level map. Each waypoint carries a map identifier, and the mission orchestrator compares successive waypoints to detect when a route crosses into a different map; when this happens, it requests a coordinated map switch that updates the map server, restarts or reconfigures the localization node (AMCL versus `gps_amcl`), and refreshes the costmaps before issuing the next goal. This design keeps map transitions explicit and debuggable, and avoids hidden cross-floor dependencies in the navigation engine.

Fourth, the choice to derive odometry from the OAK–D IMUs and fuse it through an EKF reflects both hardware availability and robustness considerations. Using the IMUs embedded in the depth cameras reduces the need for additional dedicated odometry hardware, while the EKF node provides a standard ROS interface (`odom`) that can be consumed by AMCL and `gps_amcl` regardless of the final mix of odometry sources. Similarly, building a merged `/scan` topic from the RP–LiDAR and the two depth-derived laser scans ensures that both low, near-field obstacles and higher or partially occluded objects are visible to the navigation stack, without requiring any modification to components that expect a conventional `LaserScan` input.

Finally, a distinct override node at the velocity-command level was introduced instead of embedding guest-tracking logic inside the navigation stack. This lower-layer module can temporarily take control of the robot's motion—for example, to stop and turn the robot until a guest re-enters the cameras' field of view—by replacing the commands produced by `move_base` with its own, then handing control back once tracking is re-established. Keeping this behaviour in the execution layer, and exposing it to the navigation layer only as an override of `cmd_vel`, avoids entangling guest-tracking policies with path-planning logic and allows changes to the tracking system without reconfiguring the navigation engine. Together, these decisions produce a navigation layer that is tightly integrated with the existing platform but remains modular, testable, and adaptable to future extensions.

3.9 Chapter Summary

This chapter specified and implemented the NavHost navigation layer that allows the Host Robot, built on a wheelchair platform, to execute long, multi-floor missions in large public buildings. Starting from the deployment constraints and mixed ROS 1/ROS 2 software stack, it derived a set of functional and non-functional requirements for map-based navigation, waypoint missions, multi-map operation, safe and comfortable motion, and clean communication with decision and execution layers.

On this basis, the chapter proposed an architecture that reuses the existing Intellwheels ROS 1 navigation stack as the core single-map engine and extends it with three key capabilities: a waypoint-graph mission layer that decomposes high-level destinations into sequences of semantically meaningful waypoints, a MapSwitcher that performs coordinated changes of map, localisation pipeline, and costmaps, and a safety-oriented velocity override path that can temporarily supersede `move_base`'s commands. Around these elements, it defined a perception and localisation pipeline that fuses OAK–D-based odometry, LiDAR and depth-derived scans, and, when available, RTK-quality GNSS to support both indoor AMCL and outdoor GPS-augmented localisation.

The chapter then detailed each custom component and its data flows: the YAML waypoint representation with map identifiers and semantic types, the `move_base_handler` state machine for mission execution and high-level failure reporting, the MapSwitcher sequence for robust map transitions, and the override node that multiplexes between autonomous and safety commands at the `cmd_vel` interface. It also described the ROS graph and bridge configuration that integrate the navigation layer with ROS 2 decision and low-level control nodes in simulation. Together, these design and implementation choices yield a waypoint-driven, multi-map, safety-aware navigation layer that remains modular and closely aligned with standard ROS practice, providing the foundation for the experimental evaluation presented in Chapter 4.

Chapter 4

Results and Discussion

4.1 Chapter Overview

This chapter evaluates the behaviour of the navigation layer proposed in Chapter 3 under representative missions in a FEUP-derived simulation environment. The evaluation considers whether the system can execute long waypoint-based missions reliably, whether the MapSwitcher and waypoint mission orchestrator behave as intended during map transitions, whether the GNSS-aided localization configuration supports outdoor navigation, and how the velocity override module affects navigation when activated by the lower control layer.

The experiments progress from a single-floor indoor baseline to multi-floor elevator missions, followed by an outdoor route evaluated with GNSS-aided localization, and finally to runs in which override episodes are deliberately injected. Together, these scenarios evaluate the principal capabilities introduced by the proposed architecture: waypoint-based mission execution, multi-map and multi-floor coordination, indoor–outdoor localization selection, and temporary safety-oriented control intervention.

Beyond reporting numerical results, the chapter also examines qualitative aspects of behaviour that are important for a host robot. These include whether motion is smooth and predictable, whether pauses and map transitions are understandable from a human observer's perspective, whether the localization configuration remains appropriate for the active environment, and whether failures, when they occur, surface at the appropriate abstraction level. The aim is therefore not only to show that missions succeed, but also to understand how the architectural components introduced in Chapter 3 contribute to that success.

4.2 Experimental Methodology

4.2.1 Test Environments

All experiments are conducted in simulation using Gazebo, with environments derived from floor plans of the FEUP building. The original architectural drawings were first cleaned to remove non-structural details such as furniture, annotations, and decorative elements, and then rasterised into

occupancy bitmaps compatible with the ROS navigation stack (see Figures 4.1 and 4.2). These bitmaps were subsequently converted into vector outlines and extruded to form 3D geometry, yielding Gazebo worlds in which walls, corridors, and rooms reproduce the layout and dimensions of the corresponding floors in the real building while remaining clean enough for controlled evaluation.

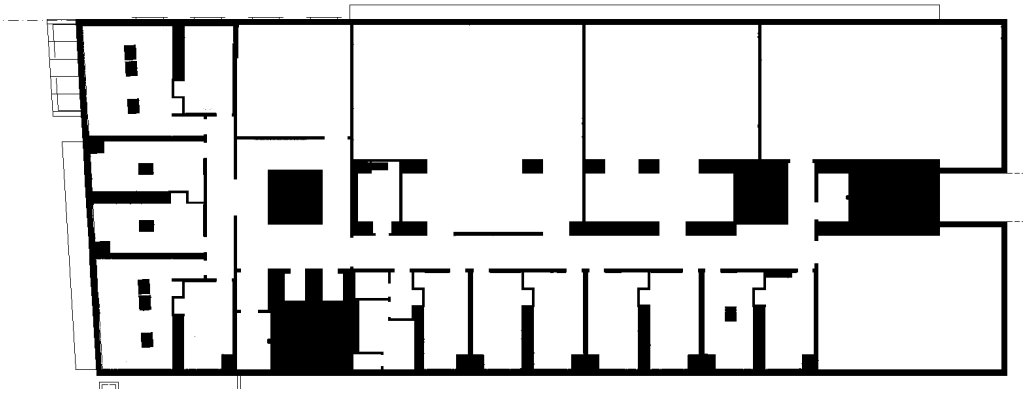


Figure 4.1: First-floor FEUP occupancy map used in the experiments. White regions correspond to free space, while black regions represent walls and structural obstacles.

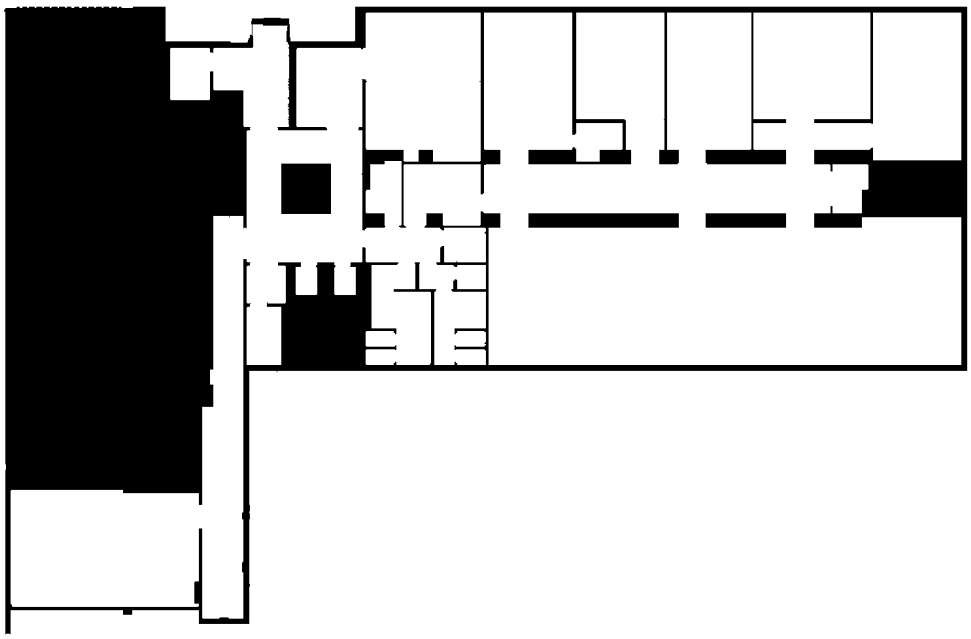


Figure 4.2: Basement-floor FEUP occupancy map used in the experiments, including the lower corridor and elevator area used for multi-floor missions.

The main test environment consists of two floors of the building represented within a single Gazebo world. To simplify multi-floor experiments, both floors are placed at the same height; the distinction between them is handled logically through separate 2D maps in the navigation layer rather than through vertical geometry. In practice, the first-floor map shown in Figure 4.1 and the

basement-floor map shown in Figure 4.2 are treated as distinct navigation worlds, even though they coexist side by side in simulation.

Floor changes are emulated by an auxiliary node that instantaneously moves the robot's pose from the elevator interior on one floor to the elevator lobby on the other, at the same time as the navigation layer performs a map switch. From the perspective of the waypoint mission orchestrator and the MapSwitcher sequence described in Chapter 3, this corresponds to entering a lift on one floor, being transported, and resuming navigation on the new floor, but without modelling elevator motion explicitly.

This design choice keeps the focus of the experiments squarely on the navigation layer. By abstracting away the physical motion of the elevator and the details of door operation, it becomes easier to isolate errors due to map switching, localisation, or waypoint handling, without confounding them with issues in the underlying simulation of lift kinematics. At the same time, the logical separation into two maps preserves the essential challenge that the navigation layer must solve: moving between parts of the building that are not connected by a single 2D occupancy grid.

Inputs that would normally come from other robot layers are also simulated. High-level missions are issued as destination goals or waypoint sequences published on the same ROS topics that the ROS 2 decision layer will use, and guest-tracking or safety-related behaviours are triggered by synthetic override messages that activate the velocity override node. Likewise, GNSS measurements for outdoor segments are emulated as RTK-quality pose updates into the `gps_amcl` module when the "outdoor" map is active, while odometry and range data are generated from the robot model, the OAK–D camera plugins, and the RP–LiDAR plugin configured in Gazebo. This setup allows the navigation layer to be exercised under realistic sensor and mission conditions while keeping the experiments repeatable and independent of hardware availability.

Although the present chapter focuses on indoor and elevator missions, the same infrastructure could in principle be reused to construct outdoor or mixed indoor–outdoor scenarios, simply by adding additional maps and teleport regions. The methodology described here is therefore intended not only to evaluate the current system, but also to provide a foundation for future experiments that extend the Host Robot's operational envelope.

4.2.2 Metrics and Data Collection

The evaluation focuses on whether the proposed navigation layer executes long, multi-floor missions reliably and comfortably, and on how it behaves around the architectural features it was designed to handle: waypoint-based routing, map switching, and override episodes. To this end, each experiment logs a set of ROS topics into `rosbag` files, including robot pose, odometry, velocity commands, waypoint and mission states, map and localization status, and override signals. All quantitative metrics are computed offline from these logs.

At the mission level, three primary metrics are recorded for each run: mission success or failure, total mission duration, and total path length travelled. A mission is considered successful if the final waypoint in the planned sequence is reached without operator intervention and without the waypoint orchestrator declaring a persistent blockage. Mission duration is measured from the

moment the first waypoint is sent to `move_base_handler` until mission completion or abort is reported, while path length is obtained by integrating the robot's odometry over the same interval. For each route, the length of the shortest path over the waypoint graph is also computed, and the ratio between travelled distance and this nominal graph distance is used as an indicator of detours and local re-planning.

At the waypoint level, the experiments track the number of waypoints planned and the number successfully reached for each mission, as well as the time elapsed between successive waypoint completions. For waypoints associated with map transitions, additional events are extracted from the logs: whether the MapSwitcher successfully changed the active map, whether the appropriate localization mode (AMCL or `gps_amcl`) was active afterwards, and whether `move_base` resumed normal operation on the new map without manual intervention. In cases where the mission is aborted, the waypoint at which progress stalled and the outcome reported by the orchestrator (e.g., timeout, repeated goal failure) are recorded to support later qualitative analysis.

Navigation comfort and dynamic behaviour are assessed through velocity-based metrics. From the commanded velocity stream (`cmd_vel`), the experiments compute average and peak linear and angular velocities, approximate peak accelerations and decelerations, and the number of abrupt stops or reversals in direction along each route. These measures provide an indirect assessment of how smoothly the TEB-based local planner handles narrow passages, turns, and dynamic obstacles in the FEUP-derived environment. In addition, the number of times the "no progress" condition in `move_base_handler` is triggered before a waypoint is either reached or aborted is counted, distinguishing transient slowdowns from genuine blockages.

Finally, to characterise the interaction between navigation and the override mechanism, all periods during which the velocity override node takes control of `cmd_vel` are logged with their start and end times. For each override episode, the experiments record its duration, the reason for activation (for example, a simulated loss of the escorted guest from the cameras' field of view), and the additional time and distance incurred compared to a nominal run of the same mission without overrides. Together, these mission-, waypoint-, and command-level metrics provide the basis for comparing different experimental scenarios in the following sections and for discussing how well the navigation layer satisfies the requirements identified earlier in the dissertation.

In addition to these quantitative measures, qualitative observations are made during each run, focusing on behaviours that are difficult to capture in single numbers: whether the robot hesitates at doors, how it approaches tight corners, how natural the pauses around elevator transitions appear, and whether override episodes feel abrupt or smoothly integrated. These observations are used to complement the numerical results and to highlight areas where further refinement might improve user perception, even if raw performance metrics are already satisfactory.

4.3 Single-Floor Navigation Experiments

4.3.1 Experimental Setup

The first experiment evaluates the navigation layer on a single floor of the FEUP environment, using a mission in which the robot must move from an initial position A to a distant target position B along the main corridor. Figure 4.3 shows the corresponding route on the first-floor map, with A and B located in opposite wings of the building so that the mission exercises long-range corridor navigation, several doorways, and multiple turns while remaining on a single 2D map. This scenario provides a baseline against which the additional complexity of multi-floor and override experiments can be interpreted.

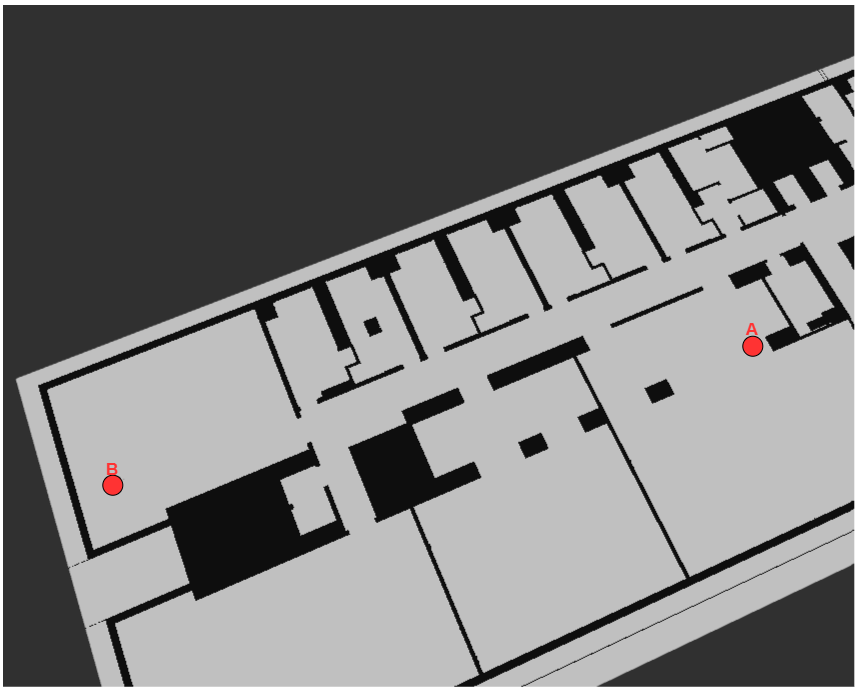


Figure 4.3: Single-floor experiment: initial pose A and goal B on the first-floor FEUP map used in simulation.

To issue repeatable missions, a small ROS node, `goal_publisher`, was implemented to publish navigation goals and, in later experiments, elevator teleport commands on the same interfaces that will be used by the ROS 2 decision layer. The node reads the desired goal coordinates and map name from private parameters `~goal_x`, `~goal_y`, and `~goal_map`, defaulting to $(126.0, 15.0)$ on map `feup_1_floor`, and provides a simple text interface in the terminal. When the user types `go`, the node constructs a `NavigationGoal` message containing the configured pose and map identifier and publishes it on the `/goal` topic; when the user types `tp`, it publishes an empty message on `/teleport_now`, which is only used in the subsequent multi-floor experiments to trigger simulated elevator teleports. This approach makes it possible to run several repetitions of the same mission without relying on RViz or higher-level components, while keeping the interface to the navigation layer identical to what will be used in the integrated robot.

For this first experiment, the robot is initialised at A with the correct pose on `feup_1_floor`, the costmaps are allowed to initialise, and then the `go` command is issued to send the mission goal B to the waypoint mission handler. The handler plans a waypoint route over the single-floor topological graph and feeds the resulting waypoint poses to `move_base` one at a time, as described in Chapter 3, while a `rosbag` recording logs odometry, commanded velocities, mission and waypoint states, and the goal messages themselves for offline analysis. No override episodes are triggered in this scenario, so the velocity override node remains in pass-through mode throughout all runs.

To ensure that the same analysis procedure is applied across all experiments, a Python script was developed to process every bag file in batch. For each bag in a `bags/` directory, the script first exports the relevant topics (`/odom`, `/cmd_vel`, and `/goal`) to CSV files using `rostopic echo -b -p`, storing the results under a dedicated subdirectory in `bag_analysis/`. It then computes, for each run, the mission duration as the time between the first goal timestamp and the last odometry sample, the travelled path length by integrating the odometry from the goal time onwards, and the average and maximum magnitudes of the commanded linear and angular velocities. Finally, it aggregates these metrics across all bags into a single table and a set of summary statistics, which are used directly in the quantitative results reported below.

4.3.2 Quantitative Results

Five runs of the A-to-B mission were recorded under identical conditions to obtain basic performance statistics for single-floor navigation. For each run, mission duration was measured from the timestamp of the first goal message on `/goal` until the last odometry sample considered by the analyser, and path length was computed by integrating the odometry between the same timestamps. All five missions succeeded on the first attempt, with modest run-to-run variation in duration and path length attributable mainly to small differences in local planner behaviour around doors and corners.

Table 4.1 summarises the aggregated metrics for this experiment. Mission durations cluster around one minute and forty-five seconds: the mean duration is 104.73 s with a standard deviation of 8.46 s, and values range from 98.72 s to 119.25 s. The corresponding travelled path length has a mean of 79.27 m with a standard deviation of 0.26 m, bounded between 78.88 m and 79.56 m. The tight spread in path length indicates that the robot follows a very similar route in each repetition, with only minor detours introduced by local re-planning.

The commanded velocity statistics show that the robot operates near its configured cruise speed in the main corridor while slowing appropriately near turns and doors. The average magnitude of the linear velocity $|v|$ across runs is 0.815 m/s (standard deviation 0.013 m/s), with a maximum $|v|$ of 1.0 m/s in every run. The average absolute angular velocity $|\omega|$ is 0.130 rad/s (standard deviation 0.0059 rad/s), and the maximum $|\omega|$ is 0.5 rad/s. These values reflect the intended behaviour of the TEB local planner on the wheelchair-based platform: brisk but not aggressive

motion along straight segments, and smooth, moderate turns at corridor junctions and doorways. [1]

Table 4.1: Summary of mission- and command-level metrics for the single-floor A-to-B navigation experiment (five runs).

Metric	Mean	Std. dev.	Min	Max
Mission duration [s]	104.73	8.46	98.72	119.25
Path length [m]	79.27	0.26	78.88	79.56
Avg. $\lvert v \rvert$ [m/s]	0.815	0.013	0.804	0.837
Max. $\lvert v \rvert$ [m/s]	1.000	0.000	1.000	1.000
Avg. $\lvert \omega \rvert$ [rad/s]	0.130	0.0059	0.120	0.136
Max. $\lvert \omega \rvert$ [rad/s]	0.500	0.000	0.500	0.500

4.3.3 Discussion

These results indicate that, on a single floor with a static map and no overrides, the navigation layer behaves as intended: all A-to-B missions complete successfully, without intervention, and with trajectories that are close to the ideal waypoint-graph distance. The very small variation in path length around 79 m and the consistent peak velocities across runs suggest that the underlying planners behave deterministically in the relatively simple corridor geometry, with only minor deviations introduced by timing and costmap updates.

The velocity statistics show that the chosen configuration for the TEB local planner and the wheelchair base yields motion that is neither excessively slow nor overly aggressive. Peak speeds reach the configured 1.0 m/s limit, but the average linear velocity remains lower, reflecting the time spent decelerating and accelerating near doors and junctions. The moderate average angular velocity indicates that rotations are executed smoothly rather than with abrupt changes, which is consistent with the comfort requirements of a host robot operating among people in narrow institutional corridors.

The absence of "no progress" events or mission-level recovery actions in any of the runs suggests that, in this baseline scenario, the main architectural features of the navigation layer—waypoint structuring over a single map and the integration of `move_base`—do not introduce unintended deadlocks or oscillations. Instead, they provide a stable framework within which the standard ROS navigation stack can operate. From a design perspective, the experiment also validates the decision to keep the waypoint graph relatively sparse and to delegate detailed motion between waypoints entirely to `move_base`. The fact that the robot can cover a long single-floor route with near-optimal distance and without mission-level intervention indicates that the chosen waypoint placement—at doors, junctions, and other passage points—is sufficient to guide global planning, while the local planner is capable of dealing with the remaining geometric detail.

[1] A video of the experiment can be seen here: https://youtu.be/bHArLtjUkes

4.4 Multi-Floor Navigation via Elevator

4.4.1 Experimental Setup

The second set of experiments evaluates the behaviour of the waypoint mission orchestrator and the MapSwitcher during missions that traverse two floors of the FEUP environment via an elevator. As in the single-floor case, missions are issued through the `goal_publisher` node, which publishes a high-level navigation goal (final destination pose and map identifier) on the same interface that the ROS 2 decision layer will use. From the navigation layer's perspective, the only difference relative to the first experiment is the structure of the waypoint route and the presence of an intermediate elevator segment; the way goals are injected and logged remains unchanged.

For these experiments, two separate 2D maps are used, each representing one floor of the building (Figures 4.1 and 4.2). Both maps are derived from FEUP floor plans using the same pipeline as in the single-floor experiment (floor-plan to bitmap, bitmap to vector, vector extrusion into a 3D Gazebo world), so the reader is referred to Section 4.2.1 for the details of this process. Here it suffices to note that each floor is represented by its own occupancy grid and that the corresponding Gazebo worlds are arranged side by side at the same height, so that the distinction between floors is logical (per-map) rather than geometric (physical height difference).

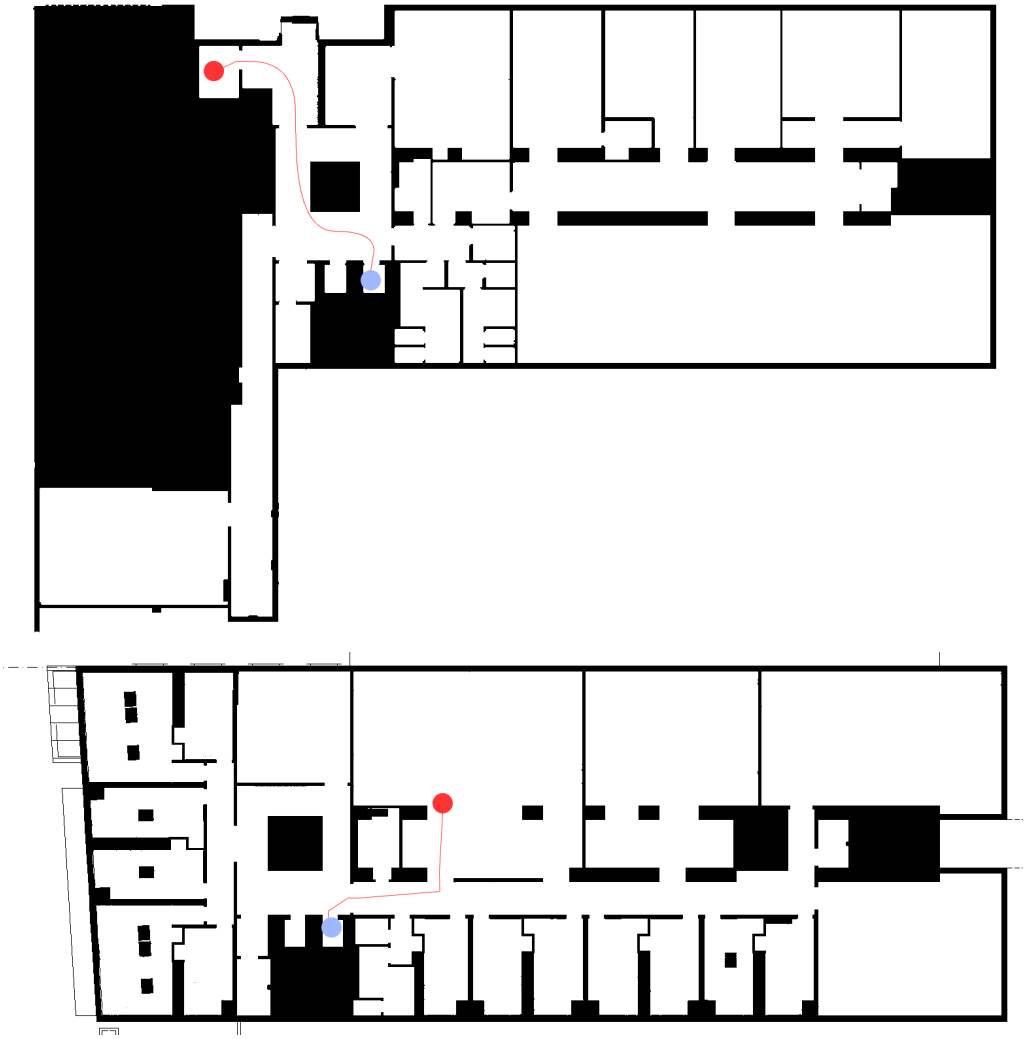


Figure 4.4: Multi-floor mission used in the elevator experiments. The upper map corresponds to the basement floor and the lower map to the first floor. Red markers indicate the initial and final mission waypoints, while blue markers show the elevator connection points used to teleport the robot between floors. A representation of the selected trajectory is shown with a red line

Figure 4.4 provides an overview of the full two-floor mission. The bottom half of the image shows the initial part of the route on the first-floor map: the robot starts from a pose analogous to point A in the single-floor experiment and moves along the main corridor towards the nearest elevator lobby. The top half then shows the continuation of the mission from the elevator lobby on the upper floor to the final target position. In both cases, waypoints are placed at corridor junctions, on each side of doors, and at the elevator interior and lobbies, so that the navigation layer can structure the mission as a sequence of semantically meaningful passage points across both maps.

Operationally, the elevator motion is simulated by a dedicated `elevator_teleporter` node running alongside Gazebo. The node subscribes to `/gazebo/model_states` to track the robot's pose and reads a list of elevator trigger regions and destinations from a YAML configuration file. Each elevator is specified by a circular trigger area on one floor and a destination

pose on the other. When the robot has remained inside a trigger area for at least a configurable dwell time (by default 3 s), the node issues a `/gazebo/set_model_state` request that instantaneously moves the robot from the elevator interior on one map to the corresponding lobby pose on the other.

To integrate this behaviour with the navigation layer and make the elevator motion appear as part of the mission, MapSwitcher was extended with a testing-only hook. Immediately after stopping the current localisation instance and before starting the new one, the node now publishes an empty message on a `/teleport_now` topic and waits briefly:

```
self.stop_localization()

# Testing only: trigger simulated elevator motion
rospy.loginfo("Requesting teleport...")
self.teleport_pub.publish(EmptyMsg())
rospy.sleep(0.5)

self.start_localization(localization)
```

The `elevator_teleporter` subscribes to `/teleport_now` and, upon receiving this message, checks whether the robot is currently within any elevator trigger area. If so, it performs the teleport to the configured destination pose on the other floor; if not, it logs a warning and leaves the robot in place. This explicit coupling between MapSwitcher and `elevator_teleporter` ensures that the simulated elevator ride is synchronised with the navigation-layer view of the world: the map is about to change, so the physical pose is changed in Gazebo at the same moment.

From the mission orchestrator's point of view, elevator use is encoded purely through waypoints. A mission that goes from the first to the second floor is defined as a sequence of waypoints that begins at the start position on the first floor, passes through an elevator interior waypoint on that map, continues from the corresponding elevator lobby waypoint on the second-floor map, and ends at the final destination on that floor. The orchestrator runs exactly the same state machine as in the single-floor scenario: it expands the high-level goal into a waypoint sequence, iterates through the sequence one waypoint at a time, and, before sending each goal to `move_base`, enters the CHECK_MAP state to verify whether the next waypoint lies on the current map or requires a call to `/switch_map`.

As before, all relevant topics are recorded to `rosbag` for offline analysis, including `/odom`, `/cmd_vel`, the waypoint and mission states, and the MapSwitcher and `elevator_teleporter` events. For the multi-floor experiments, additional attention is paid to the moments around floor transitions: the timestamps and outcomes of all calls to `/switch_map`, the activation of the appropriate localisation pipeline (indoor AMCL or `gps_amcl` when outdoor maps are present), and the behaviour of `move_base` immediately before and after each switch. The velocity override node remains in *PassThrough* mode throughout these runs so that any changes in behaviour can be attributed entirely to the waypoint graph, MapSwitcher, and elevator teleporter.

4.4.2 Quantitative Results

The multi-floor mission was executed five times under identical conditions to obtain statistically meaningful performance measures. For each run, the same mission-level metrics as in the single-floor experiment were computed from the recorded logs: mission duration from the time the goal was issued until the final waypoint was reached, and total path length from the integrated odometry after goal issuance. In addition, two metrics specific to floor transitions were monitored: the number of successful map switches per mission and the number of failed or timed-out calls to `/switch_map`. In all five runs, the expected number of floor transitions (two, corresponding to entering and leaving the elevator segment) were completed without manual intervention, and no failed or repeated map-switch attempts were observed.

Across the five repetitions, mission duration had a mean of approximately 131.4 s with a standard deviation of 9.1 s, and ranged from 124.5 s to 146.9 s. The corresponding travelled path lengths had a mean of about 160.2 m with a very small standard deviation of 0.8 m, and lay between 159.2 m and 161.3 m. The ratio between the travelled distance and the shortest path over the waypoint graph therefore remained very close to one in all runs, indicating that the robot followed the planned waypoint sequence without significant detours.

Velocity statistics derived from the `/cmd_vel` stream show that the robot's motion remains within the intended comfort envelope for a wheelchair-derived base. Over the five runs, the average magnitude of the commanded linear velocity $|v|$ was approximately $0.55\,\mathrm{m\,s^{-1}}$ (mean $0.552\,\mathrm{m\,s^{-1}}$, standard deviation $0.034\,\mathrm{m\,s^{-1}}$), with a consistent peak $|v|$ of $1.0\,\mathrm{m\,s^{-1}}$ in all runs, matching the configured cruise speed limit. The average magnitude of the angular velocity $|\omega|$ was around $0.248\,\mathrm{rad\,s^{-1}}$ (standard deviation $0.006\,\mathrm{rad\,s^{-1}}$), with a maximum $|\omega|$ of $0.5\,\mathrm{rad\,s^{-1}}$. These values reflect a pattern in which the robot travels at or near its configured linear cruise speed in open corridors, slows appropriately near doors, junctions, and the elevator lobbies, and uses moderate angular velocities to negotiate turns and reorientations.

At the waypoint level, all intermediate nodes on both floors were reached in the planned order in every run; no waypoints were skipped, reordered, or revisited, and no "no progress" events were triggered near the elevator approaches despite the tighter corridor geometry in those areas. Inspection of the logs around each floor transition shows a short, well-defined pause in the mission orchestrator's state machine: after detecting that the next waypoint lies on a different map, it enters CHECK_MAP, calls `/switch_map` with the target map identifier and an initial pose inside the elevator on the new floor, and only proceeds to send the next waypoint goal after the service returns success.

The MapSwitcher logs confirm that, for each requested transition, the correct map was loaded, the appropriate localisation launch file was restarted, an initial pose was published on `/initialpose`, and the `/nav/clear_costmaps` service was called on `move_base`. In parallel, the `elevator_teleporter` logs record a single teleport event per transition, triggered either automatically after the dwell time in the elevator trigger area or immediately in response to the `/teleport_now` message sent by MapSwitcher in the testing configuration. After each transi-

tion, both global and local costmaps resumed publishing normally on the new map, and `move_base` accepted and executed the subsequent waypoint goal without error. [2]

4.4.3 Discussion

Taken together, the multi-floor experiments provide strong evidence that the waypoint graph, MapSwitcher, and elevator teleporter mechanisms work together as intended. First, the fact that every mission completed successfully with the planned number of floor transitions, and that all intermediate waypoints on both floors were reached in sequence, shows that the waypoint mission orchestrator correctly expands high-level goals into building-wide routes and drip-feeds those waypoints to `move_base` one hop at a time. The orchestrator's division of labour with the navigation stack remains clear: it decides which waypoint comes next and when a map change is required, while `move_base` continues to handle the continuous motion between waypoints on whichever map is currently active.

Second, the absence of failed or repeated map-switch attempts, combined with the clean resumption of navigation after each transition, indicates that MapSwitcher consistently coordinates map loading, localisation restart, initial pose publication, and costmap clearing. From the navigation stack's viewpoint, each floor change appears as an atomic reconfiguration event: the map, localisation, and costmaps are updated in a tightly controlled sequence, and only then is a new goal issued. The short, clearly delimited pauses observed around floor transitions match the expected behaviour of the CHECK_MAP state: during these intervals the robot remains stationary while the navigation stack is reconfigured and teleported into the elevator lobby on the new floor, after which it immediately continues to the next waypoint.

Crucially, no spurious motion or costmap artefacts were observed during or after switches. The costmap reset performed at the end of the map-switch sequence prevents stale obstacle information from the previous map from leaking into the new one, and the odometry and pose estimates remain consistent across the teleports thanks to the controlled initial-pose publication. From a mission-level perspective, the resulting behaviour is that of a host robot that reliably follows a waypoint-defined route across floors, with elevator segments appearing as short, predictable interruptions rather than as sources of navigation failure or erratic behaviour.

These experiments also highlight a practical advantage of handling multi-floor navigation through map-aware waypoints and an explicit map-switching service, rather than through a single integrated 3D map. Because each floor is represented by its own 2D occupancy grid and each vertical transition is encoded as a pair of waypoints plus a call to `/switch_map`, the same machinery can be reused if additional floors or outdoor maps are introduced. The core navigation engine (`move_base`) remains oblivious to the existence of multiple maps, operating on whatever occupancy grid MapSwitcher has made current, while the mission layer retains full control over when and where transitions occur. In this sense, the evaluation presented here serves both as a

[2] A video of the experiment can be seen here: https://youtu.be/kyA7GCOumRw

proof-of-concept for the FEUP environment and as a template for extending the Host Robot to larger and more complex institutional sites in future work.

4.5 Outdoor Navigation with GNSS-Aided Localization

4.5.1 Experimental Setup

The final experiment evaluates outdoor navigation using the GNSS-aided localization configuration, `gps_amcl`. The route begins at the mechanical barrier located in the parking area in front of FEUP and ends at the Department of Electrical Engineering, as shown in Figure 4.5. This scenario evaluates the system over a substantially longer route than the previous indoor experiments and in an environment that does not provide the dense, structured wall geometry normally available for map-relative localization.

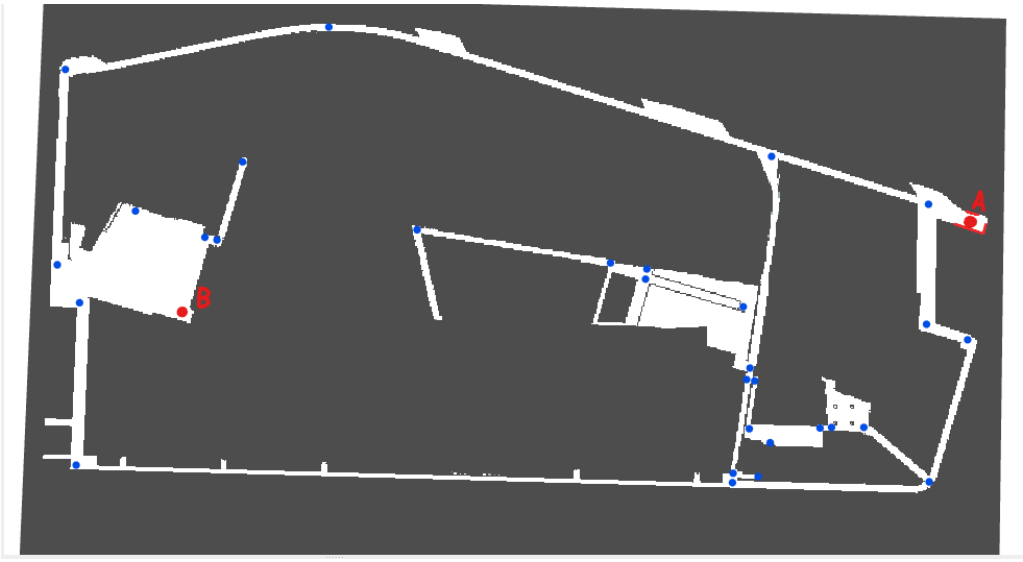


Figure 4.5: Outdoor FEUP environment used in the GNSS-aided navigation experiment. The start and end positions are shown in red, while the blue markers represent waypoints placed in the topological map for this area.

The route is represented as an ordered sequence of waypoints, indicated by the blue markers in Figure 4.6. These waypoints were positioned strategically rather than uniformly along the route. First, several are placed before or after narrow passages, bends, and constrained areas so that the robot approaches them with an appropriate orientation. This reduces the likelihood of an unsuitable approach angle and facilitates traversal through narrow sections of the environment. Second, the waypoints divide the overall route into shorter, predominantly straight navigation segments. This simplifies each individual planning task and allows the local planner to focus on short-range obstacle avoidance and trajectory execution between consecutive goals. Finally, some waypoints correspond to entrances to university buildings. Besides their geometric role in the route, these points can serve as semantic links between the outdoor map and the indoor maps

associated with each building, allowing a future mission to transition from outdoor navigation to indoor navigation through a known entrance location.

The mission was executed using the same waypoint-based framework used in the previous experiments. The mission layer provides the ordered waypoint sequence, while `move_base` performs local planning and generates velocity commands between consecutive points. Mission duration, travelled path length, and commanded linear and angular velocities were recorded for offline analysis.

Before adopting `gps_amcl`, the same outdoor route was tested using the conventional AMCL localization configuration. Although AMCL is effective in structured indoor environments, it was not sufficiently reliable for this scenario. The outdoor map covers a large area and contains relatively few distinctive, stable geometric features that can be matched consistently against the incoming range measurements. As a result, the pose estimate drifted during navigation, and this drift was observed repeatedly throughout the route rather than as an isolated event. The accumulated localization error could cause the estimated position to diverge from the robot's actual position, making long-range navigation with conventional AMCL unreliable in this environment.

The `gps_amcl` configuration was therefore used to provide an absolute-position reference for the outdoor segment. By incorporating GNSS-based position information, this configuration reduces the dependence on laser-scan matching against sparse outdoor map features and is better suited to the scale and open geometry of the route.

4.5.2 Quantitative Results

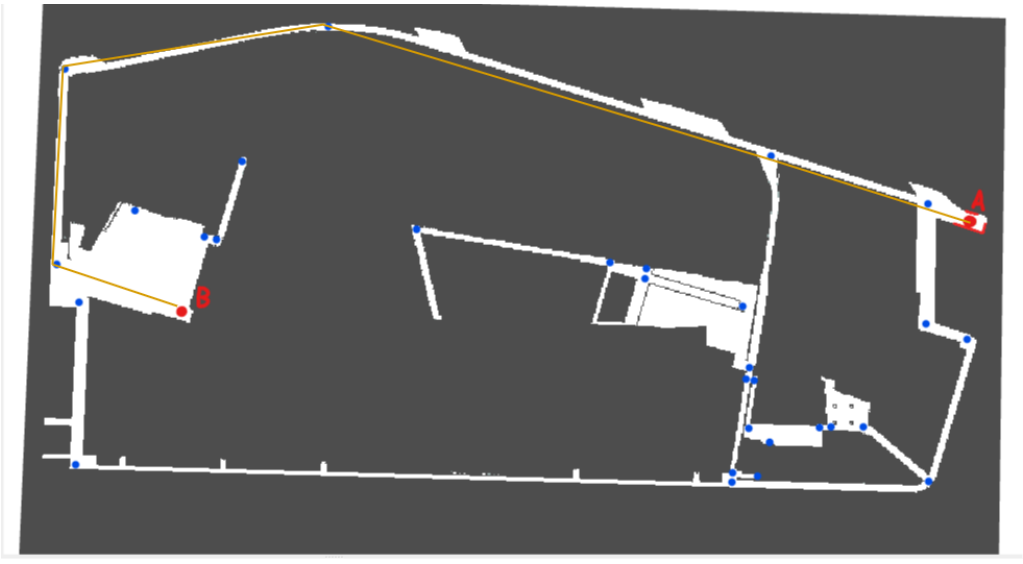


Figure 4.6: Outdoor FEUP route used in the GNSS-aided navigation experiment. The start and end positions are shown in red, while the blue markers represent the intermediate navigation waypoints. The orange line shows the trajectory selected and executed by the navigation system through these waypoints.

The orange trajectory in Figure 4.6 represents the route selected and executed by the navigation system through the predefined waypoints.

As can be seen in table 4.2, the outdoor mission had a mean duration of 676.435 s, corresponding to approximately 11.3 minutes. The standard deviation was 34.529 s, with recorded durations ranging from 647.303 s to 734.979 s. The travelled path length was highly consistent across the experiment, with a mean of 477.675 m, a standard deviation of 1.184 m, and values between 476.531 m and 479.345 m.

The commanded linear velocity had a mean magnitude of 0.708 m/s and a standard deviation of 0.033 m/s. Its values ranged from 0.652 m/s to 0.738 m/s, while the maximum linear velocity reached the configured limit of 1.0 m/s. The mean angular velocity magnitude was 0.220 rad/s, with a standard deviation of 0.006 rad/s and a range from 0.214 rad/s to 0.229 rad/s. The maximum angular velocity was consistently 0.5 rad/s. [3]

Table 4.2: Summary of mission- and command-level metrics for the outdoor navigation experiment using GNSS-aided localization.

Metric	Mean	Std. dev.	Min	Max
Mission duration [s]	676.435	34.529	647.303	734.979
Path length [m]	477.675	1.184	476.531	479.345
Avg. $\lvert v \rvert$ [m/s]	0.708	0.033	0.652	0.738
Max. $\lvert v \rvert$ [m/s]	1.000	0.000	1.000	1.000
Avg. $\lvert \omega \rvert$ [rad/s]	0.220	0.006	0.214	0.229
Max. $\lvert \omega \rvert$ [rad/s]	0.500	0.000	0.500	0.500

4.5.3 Discussion

The results show that the navigation system was able to execute the long outdoor route with a stable travelled distance and a consistent command profile. The 477.675 m mean path length varied by only 1.184 m across the recorded runs, indicating that the robot followed substantially the same waypoint-defined route in each execution. This small variation suggests that the combination of the selected waypoint layout, the local planner, and the GNSS-aided localization estimate limited unnecessary detours and route deviations.

Mission duration exhibited greater variation than path length, ranging from approximately 10.8 to 12.2 minutes. Since the geometric route remained consistent, this variation is more likely associated with differences in local motion behaviour, such as the time spent accelerating, decelerating, aligning with intermediate waypoint goals, and negotiating turns or constrained sections of the route. The waypoint structure contributes to predictable navigation by breaking the route into manageable segments, but each waypoint also introduces a local goal-approach and orientation-adjustment phase that can affect the total completion time.

The velocity results remain within the limits configured for the wheelchair-derived platform. The peak linear and angular velocities reached 1.0 m/s and 0.5 rad/s, respectively, in all runs,

[3] A video of this experiment can be seen here: part 1 https://youtu.be/i3w9H1fctxQ part 2 https://youtu.be/krCW-kIWX-A

whereas the mean linear velocity remained at 0.708 m/s. This difference reflects the expected behaviour of the local planner: the robot can travel at its maximum configured speed on sufficiently clear portions of the route, but reduces speed while turning, approaching a waypoint, or adapting its local trajectory. The mean angular velocity of 0.220 rad/s is also consistent with controlled directional adjustments rather than abrupt rotational behaviour.

The preliminary trials using standard AMCL demonstrate why the GNSS-aided configuration is required for this type of outdoor mission. AMCL relies on matching current range observations with known occupancy-map geometry. In the large outdoor FEUP map, however, relatively open areas and the scarcity of distinctive, persistent features provide weak constraints for that matching process. Consequently, localization drift became a recurrent problem during the route. This is particularly significant for a long mission, because small localization errors can accumulate until the robot's estimated pose no longer corresponds sufficiently closely to its physical position for reliable waypoint execution.

Using `gps_amcl` mitigates this limitation by adding an absolute-position source that is not solely dependent on local scan matching. The stable path lengths and successful execution of the approximately 478 m mission support the use of GNSS-aided localization for open-air sections of the proposed operational environment. More broadly, this experiment reinforces the design decision to select the localization strategy according to context: map-relative AMCL remains appropriate for feature-rich indoor areas, whereas `gps_amcl` is better suited to extended outdoor segments where GNSS is available and environmental geometry alone does not provide a sufficiently stable localization reference.

4.6 Override Module Validation

This experiment validates the velocity override node as an interface between the navigation stack and the lower guest-monitoring and execution layers. Its purpose is not to assess the path-planning performance of `move_base`, but to verify that a lower-layer request can temporarily replace the velocity commands reaching the base and that normal navigation can subsequently resume. The main intended application is guest reacquisition: when the guest being guided is no longer visible in the rear camera, the lower layer can stop the robot and request a controlled rotation towards the side where the guest was last detected.

4.6.1 Experimental Setup

The experiment uses the single-floor A-to-B mission on the first-floor FEUP map as a nominal navigation scenario. The robot is initialized at the same starting position used in the single-floor experiment, and the waypoint mission orchestrator sends the resulting waypoint sequence to `move_base`. The navigation stack therefore continues to compute its normal local plans and publish velocity commands while the mission is active.

The `override_node` is placed between `move_base` and the robot base command interface. It receives the navigation-generated `cmd_vel` commands and republishes the final command

stream on `output_cmd_vel`. A small helper node emulates the lower guest-monitoring layer by publishing discrete mode values on `override_mode`. These commands select one of the node's operating modes: *PassThrough*, *Stop*, *RotateLeft*, or *RotateRight*.

The validation consists of three intervention cases introduced while the robot is executing the otherwise unchanged navigation mission:

1. **Stop intervention.** The helper node activates *Stop* while the robot is moving. The expected behaviour is that navigation commands are no longer forwarded and the base receives repeated zero-velocity commands until the override mode changes.

2. **Guest-search rotation.** Following a stop, the helper node activates either *RotateLeft* or *RotateRight*. This emulates a guest-monitoring component determining the side towards which the robot should turn to search for the guest. The expected behaviour is that the base receives a zero-linear-velocity command with a constant angular velocity of the configured sign.

3. **Return to autonomous navigation.** Once the simulated guest-reacquired condition is reached, the helper node activates *PassThrough*. The expected behaviour is that the override node stops publishing replacement commands and resumes forwarding the commands generated by `move_base`.

During the experiment, the `cmd_vel`, `output_cmd_vel`, and `override_mode` topics are observed in RViz, the ROS console, or recorded in a rosbag. The purpose of recording these topics is to verify which command source is active during each override phase, rather than to derive aggregate navigation-performance statistics.

4.6.2 Observed Behaviour

During normal operation, with the node in *PassThrough* mode, the velocity commands generated by `move_base` are forwarded to the execution layer without modification. The robot therefore follows the route planned by the navigation stack in the same manner as in the single-floor baseline experiment.

When the *Stop* mode is activated, the override node ceases to forward incoming `move_base` commands and publishes zero-valued velocity commands at its configured timer rate. The robot stops even though the navigation stack may continue to publish commands internally. This confirms that the override node is positioned at the final command-selection point and that the lower layer can temporarily prevent autonomous motion from reaching the base.

When *RotateLeft* or *RotateRight* is requested, the output command contains no linear velocity and a constant angular velocity with the configured sign. The robot consequently turns in place, allowing the rear-facing perception system to scan in the direction selected by the lower layer. The override node does not itself determine where the guest is located or when the search has succeeded; it only implements the motion mode requested through `override_mode`.

When the lower layer returns the mode to *PassThrough*, the override-generated stop or rotation commands cease and navigation-generated commands are again forwarded to the base. The mission-level waypoint goal does not need to be reissued and the navigation stack does not need to be restarted or reconfigured. The robot can therefore continue towards the currently active `move_base` goal once the temporary guest-reacquisition manoeuvre has finished. [4]

4.6.3 Discussion

The experiment demonstrates the intended separation of responsibilities between autonomous navigation and guest-aware motion control. The waypoint mission orchestrator remains responsible for route structure and `move_base` remains responsible for generating local navigation commands. The lower guest-monitoring layer is responsible for deciding when the robot should interrupt normal motion and which rotation direction should be used to search for the guest. The override node acts only as the command arbiter between these layers.

This separation is valuable because the guest-reacquisition behaviour does not need to be embedded in either the global planner, the local planner, or the waypoint mission state machine. The lower layer can request a stop or rotation through a small, explicit interface, while the navigation stack remains configured for its normal task. Returning to *PassThrough* restores autonomous motion without requiring a new mission, a route re-plan, or a modification to the active `move_base` goal.

The override mechanism also has a secondary safety benefit. While an override is active, only the override node publishes the final command stream, so the robot does not receive simultaneous navigation and guest-search commands. However, its principal role in this work is to support interaction-aware guidance: it allows the robot to pause and turn when visual contact with the guest has been lost, then continue the planned route once the guest has been reacquired.

4.7 Cross-Scenario Analysis

Considering the three experimental scenarios together provides a consolidated view of how the navigation layer meets the requirements identified in Chapter 3. On a single floor, the system delivers repeatable missions whose duration and path length are close to the ideal waypoint-graph values, indicating that the combination of the waypoint structure and the `move_base` configuration is well matched to the FEUP-derived environment. When extended to multi-floor missions, the same architecture handles repeated map transitions without loss of localisation or planner failures, and the robot follows the waypoint-defined routes across floors in a way that is observable both in the pose traces and in the absence of mission- level errors.

The override experiments add a third dimension to this picture. They show that the navigation layer can coexist with a safety-oriented control path beneath it without either layer having to know

[4] A video of the experiment can be seen here: `https://youtu.be/DIW5YrEqLS0`

the internal details of the other. The navigation layer continues to think in terms of goals, waypoints, and maps, while the lower layer can, when necessary, take direct control of `cmd_vel` to enforce safety or guest-tracking policies. The clean hand-off observed in the experiments suggests that this division of labour is workable in practice, and that it should generalise to other kinds of low-level interventions beyond the simple stop and rotate behaviours tested here.

Together, these results support the claim that the navigation layer satisfies its core design goals—multi-map, multi-floor, and safety-aware navigation on a wheelchair-based platform—by composing standard ROS components with a small number of targeted extensions. They also indicate that the architecture has headroom for further extensions, such as more sophisticated localisation sources, richer mission-level logic, and tighter integration with building infrastructure, without requiring a fundamental redesign of the stack.

4.8 Summary

The experiments presented in this chapter show that the proposed navigation layer behaves robustly across the three scenarios for which it was designed. On a single floor, the waypoint-structured mission layer and the `move_base`-based navigation stack consistently executed long corridor missions without intervention: all five A-to-B runs succeeded, with mission durations around 105 s and path lengths tightly clustered around 79 m, and with average and peak velocities compatible with comfortable wheelchair motion. This baseline confirms that the sparse waypoint graph and the reuse of the Intellwheels configuration are sufficient to navigate the FEUP first-floor map efficiently and without mission-level recovery.

Extending the same architecture to multi-floor missions via the simulated elevator demonstrated that the MapSwitcher and waypoint mission orchestrator correctly handle repeated map transitions. In the elevator experiments, the robot followed a building-wide waypoint route across the two FEUP floors, as summarised in Figure 4.4, with all intermediate waypoints reached in order and every requested map switch completed successfully. Mission durations increased to around 131 s and path lengths to about 160 m, reflecting the additional distance and pauses at elevator lobbies, but the ratio between travelled distance and the shortest waypoint-graph path remained close to one, and no “no progress” events or localisation failures were observed at floor boundaries.

Finally, the override experiments confirmed that the safety-oriented velocity override path can temporarily supersede the navigation stack without destabilising missions. When stop or rotate modes were activated, the override node cleanly replaced `move_base`’s commands at the `cmd_vel` interface, after which control returned seamlessly to autonomous navigation and missions completed successfully. Across all scenarios, the combined evidence supports the claim that the navigation layer meets its core design goals: enabling waypoint-based, multi-map, multi-floor navigation with a safety-aware control path, while building incrementally on top of the existing Intellwheels ROS 1 baseline rather than replacing it.

Chapter 5

Conclusions and Future Work

This chapter summarises the work presented in this dissertation, reflects on how far the proposed navigation layer meets the requirements identified earlier, and outlines directions for future development. The focus is on the extent to which the system enables a host robot to execute long, multi-floor missions in a large institutional environment while building on an existing ROS1 navigation baseline rather than replacing it. In addition to revisiting the main contributions and experimental findings, the chapter discusses how the design choices made here relate to broader trends in service-robot navigation and what kinds of extensions would be most valuable for deployment in real institutions.

By structuring the discussion in this way, the chapter aims to make clear not only what has been achieved, but also where the proposed solution sits within the wider design space of indoor–outdoor navigation for service robots. The goal is to provide a balanced view: recognising that the implemented navigation layer already enables qualitatively new capabilities for the Host Robot platform, while also acknowledging the simplifying assumptions and controlled conditions under which it has been evaluated so far.

5.1 Summary of Contributions

The starting point for this work was the observation that many of the capabilities required by a host robot in a hospital- or campus-scale institution—robust 2D navigation, multi-floor route abstraction, GNSS-aided outdoor positioning, ROS1/ROS2 interoperability—are well studied in isolation but seldom integrated into a single, reusable navigation layer for a wheelchair-based platform. Building on the Intellwheels ROS1 baseline, the dissertation addressed this gap by designing and implementing a navigation layer that brings these ingredients together in a coherent and incremental way.

Concretely, the work contributes:

- **A waypoint-graph mission layer** that structures long missions as sequences of waypoints over a building-wide graph of meaningful locations, rather than as single long-distance goals on a monolithic map. This representation captures the structure of the institution in terms

that are intuitive for both robots and humans, and it allows long routes to be decomposed into smaller, well-defined hops that are passed to the underlying planners in turn.

- **A MapSwitcher mechanism for multi-map and multi-floor operation**, implemented as an explicit service that coordinates map loading, localisation mode, and costmap resets at each transition. By making map changes atomic from the mission layer's perspective, the MapSwitcher hides the complexity of synchronising map, localisation, and planning components while still exposing clear success or failure outcomes.

- **A unified localisation interface** that integrates indoor AMCL-based localisation and outdoor GNSS-aided localisation under a common estimator output, so that the rest of the navigation stack always sees a consistent pose estimate. Although the present evaluation focuses on indoor and elevator scenarios, the architecture prepares the ground for future mixed indoor–outdoor missions by treating changes of absolute position source as a normal part of navigation rather than as a special case.

- **A velocity override path** beneath the navigation stack, allowing the lower control layer to enforce stops and controlled rotations without modifying `move_base` or the waypoint logic. This provides a simple but effective way to integrate guest-tracking and safety behaviour implemented at lower layers with the higher-level navigation policies described in this dissertation.

These contributions are realised without discarding the tuned components of the Intellwheels baseline: the standard ROS1 2D Navigation Stack remains the core engine for point-to-point motion on a single map, and the new components are layered around it. The resulting design respects the constraints of the existing hardware and mixed ROS1/ROS2 software architecture while extending the robot's navigational reach from single-floor, single-map missions to building-scale guidance tasks.

Beyond their immediate value for the Host Robot project, these contributions suggest a more general pattern for extending legacy ROS1 systems. By treating the existing navigation stack as a black box that solves the "single-goal on a single map" problem reliably, and by adding mission-level and map-management layers around it, the dissertation shows how more complex behaviours—multi-floor routes, indoor–outdoor transitions, and safety overrides—can be achieved without a full migration to ROS2 or a complete redesign of the navigation pipeline.

5.2 Discussion of Results

The experiments in Chapter 3 and Chapter 4 demonstrate that the proposed navigation layer behaves as intended in the scenarios for which it was designed. On a single floor, the system executes long corridor missions reliably, with path lengths close to the nominal waypoint-graph distance and velocity profiles consistent with comfortable use on a wheelchair-derived base. This confirms that retaining `move_base` as the core navigation engine, supplemented by a waypoint mission

layer, is sufficient for routine indoor navigation on a known map and that the added mission logic does not introduce unnecessary overhead or instability.

In multi-floor missions, the combination of waypoint graph and MapSwitcher allows the robot to traverse elevator segments and resume navigation on another floor without manual intervention. Waypoints on both floors and inside the elevator are reached in the expected order, every requested map switch completes successfully, and the navigation stack recovers cleanly on the new map. From the robot's perspective, floor changes appear as brief, controlled pauses in which the environment representation and localisation are updated, rather than as disruptive events that risk losing the route. This behaviour is particularly encouraging because it matches what a human user would expect from a host robot: short interruptions at elevators, followed by smooth continuation of the route.

The override experiments show that adding a safety-oriented control path at the velocity level does not destabilise the navigation system. When override modes are active, the robot stops or rotates as commanded, and once the override is cleared, `move_base` regains control and the mission proceeds to completion without replanning or retuning. This confirms that guest-tracking and safety interventions can be implemented beneath the navigation layer, preserving a clear separation between high-level routing logic and low-level motion control. From a software-engineering standpoint, this separation of concerns is valuable: changes to safety or tracking policies can be made in the lower layer without forcing changes to the navigation layer, as long as the override interface is respected.

Taken together, these results indicate that the navigation layer satisfies the functional and non-functional requirements defined in Chapter 3 within the scope of the simulated FEUP environment. They also suggest that the chosen architectural pattern—waypoint-structured missions over multiple maps, an explicit map-switching service, and a pass-through override node—provides a practical template for extending other ROS1-based wheelchair platforms to similar institutional settings. While the experiments necessarily abstract away some real-world complications, they show that the navigation layer can coordinate multi-floor routing, localisation changes, and safety interventions without collapsing into brittle, hard-to-maintain logic.

5.3 Limitations

Despite these positive results, several limitations remain, both in terms of the implementation and in terms of the evaluation conditions.

First, all evaluation has been conducted in simulation, albeit using environments derived from real building floor plans. While this allows controlled and repeatable experiments, it does not capture the full variability of real-world conditions, such as imperfect sensing, unmodelled furniture, temporary signage, or human behaviour in crowded corridors. For example, real-world LiDAR data often includes spurious returns from glass, reflective surfaces, and moving people, and wheel

odometry is affected by wear, uneven floors, and mechanical tolerances. Transferring the navigation layer to the physical Robot and evaluating it in a live institutional environment is therefore an essential next step if the system is to be trusted in routine operation.

Second, the current multi-floor implementation relies on a simplified elevator model in which floor transitions are instantaneous and fully under the robot's control. Real elevators introduce additional complexity: variable waiting times depending on traffic, shared use with people who may delay or reroute the car, and interfaces that may require coordination with building infrastructure or external controllers. While the waypoint and MapSwitcher design can accommodate these aspects conceptually—for example by attaching time or availability costs to elevator-related waypoints—the present implementation does not yet integrate them, nor does it model time spent waiting in front of closed doors.

Third, the localisation and perception pipelines are deliberately conservative. Indoor localisation is map-relative and laser-based, outdoor localisation is GNSS-aided, and perception focuses on geometric obstacle detection via merged laser and depth data. This matches the needs of a host robot whose primary task is safe guidance, but it does not exploit richer semantic information about people, signage, or building semantics that might further improve comfort and robustness. In particular, the system does not distinguish between people and other dynamic obstacles at the navigation level, nor does it incorporate knowledge of room types or departmental regions into its route planning beyond what is encoded in the waypoint graph.

Fourth, the evaluation of the override mechanism is scoped to a small set of scripted interventions. In a full deployment, overrides may be triggered by a wider range of conditions, from sensor anomalies to explicit input from safety officers or clinical staff, and may need to be coordinated with human decisions about when to resume autonomous motion. Analysing how such interventions interact with longer missions, particularly under high traffic or mixed indoor–outdoor conditions, remains open work, as does integrating explicit feedback to the user when overrides are active.

Finally, the current system treats repeated local planner failures or persistent blockages primarily as reasons to abort at the mission level rather than as triggers for alternative routing. While this is acceptable in controlled experiments, it limits the robot's ability to adapt to real institutional conditions, where corridors may be closed temporarily, doors locked, or elevators out of service. Extending the mission layer to reason more flexibly about alternative paths and to communicate such adaptations clearly to the decision layer would be necessary for long-term autonomous deployment.

5.4 Future Work

Several immediate extensions follow naturally from the present work and would substantially increase the practical value of the navigation layer.

5.4.1 Deployment on the Physical Robot

The most important next step is to deploy the navigation layer on the actual Host Robot platform and validate it outside simulation. Although the Gazebo experiments provide a controlled way to test waypoint execution, map switching, and override behaviour, they do not fully capture real sensing noise, wheel slip, people moving unpredictably in corridors, or the variability of a live institutional environment. Running the full system on the physical robot would therefore allow the waypoint structure, planner configuration, and map-switching logic to be evaluated under the conditions for which they are ultimately intended.

This deployment phase should include both technical and operational validation. At the technical level, it would make it possible to tune localization, costmaps, and controller parameters against real sensor streams rather than simulated ones. At the operational level, it would allow the robot to be tested in realistic host-robot missions involving users, doors, elevators, and partially dynamic public spaces, providing a stronger basis for assessing robustness, safety, and comfort.

5.4.2 Additional Odometry Sources

A second natural extension is to improve the odometry and state-estimation pipeline by incorporating additional motion sources beyond the current fused baseline. In particular, the OAK–D cameras make visual or visual-inertial odometry an attractive option, since they already provide image and IMU data that could be used to estimate short-term motion more robustly in cases where wheel slip or uneven flooring degrades purely wheel-based estimates. This could be particularly useful in long corridors, near ramps, or during transitions between indoor and outdoor areas.

Likewise, adding wheel encoders to the platform would provide a more direct and conventional odometric source for the fusion layer. Encoder-based odometry would be especially valuable for low-speed manoeuvres, controlled turns, and local planner execution, where short-term motion smoothness matters as much as global accuracy. A future version of the system could therefore fuse wheel encoders, IMU data, and visual odometry within the same EKF or UKF-based pipeline, allowing the navigation layer to rely on a richer and more fault-tolerant motion estimate than any single source can provide on its own.

5.4.3 Expansion and Improvement of the Map Library

Another important direction for future work is the expansion and refinement of the map library used by the navigation layer. Although the current architecture supports multiple maps and transitions between them, the experimental evaluation only considers a limited subset of the institution. Extending the library to include all relevant buildings and floors would be necessary to support navigation throughout the institution, rather than being restricted to the areas currently represented.

In addition to expanding its coverage, the accuracy of the existing maps should also be improved. The maps developed for this work were derived from architectural blueprints that were not necessarily representative of the latest state of the buildings. This required some manual

adaptation during their conversion into occupancy grids. The issue was particularly evident in the outdoor environment, where the original representation had to be substantially simplified to obtain a map suitable for the simulation and navigation framework. Consequently, the resulting maps may not fully represent the current physical environment, potentially affecting localization accuracy and path planning when deployed on the actual robot.

A possible improvement would be to generate updated maps directly from sensor data collected in the target environments. In particular, Simultaneous Localization and Mapping (SLAM) techniques could be used to construct more accurate representations of indoor and outdoor areas, reducing the dependence on potentially outdated architectural documentation. These maps could then be processed into occupancy grids compatible with the existing navigation stack and integrated into the MapSwitcher framework.

Furthermore, a semantic library will also need to be developed to associate identifiable locations within the institution, such as rooms, laboratories, offices, and other facilities, with representative coordinates in the corresponding maps. This would allow further interfacing with the brain layer, allowing it to more easily give destinations to the navigation layer in the form of room names or semantically relevant places. The navigation layer could then resolve a requested destination into its associated map and coordinate, using the existing route-planning and map-switching mechanisms to reach it.

Combining a more comprehensive and accurate map library with a semantic representation of the institution would provide a stronger foundation for reliable and more intuitive navigation across the institution.

5.4.4 Adaptive and Persistent Mission-Level capabilities

A further extension would be to allow the mission layer to evolve its route selection over repeated journeys. The current waypoint mission orchestrator can respond to an immediate obstruction by waiting, retrying a waypoint, removing a temporarily blocked graph connection, or aborting the mission when progress cannot be restored. However, these decisions are currently made only within the context of the active mission. Once the mission ends, the system does not retain a long-term record of which corridors, doorways, elevator approaches, or other topological connections repeatedly caused delays, local-planner failures, or persistent blockages.

Future work could introduce a route-performance memory associated with the waypoint graph. Each graph connection could maintain statistics such as the number of successful traversals, the number and duration of blocked-passage events, the frequency of `move_base` failures or retries, the average traversal time, and the observed deviation from the nominal route. These measurements could be accumulated over several missions and used to estimate the practical reliability or traversal cost of each connection, rather than treating all graph edges as equally available and equally efficient.

When planning a new mission, the route planner could incorporate this experience into its edge-cost function. Corridors or passage points that are consistently obstructed, difficult to traverse, or associated with repeated navigation failures could receive a higher cost or be temporarily

marked as unavailable. Conversely, connections that are repeatedly completed quickly and reliably could remain preferred. The resulting route would therefore adapt over time to the operational characteristics of the institution, while preserving the existing topological representation and A*-based planning approach.

This capability would be particularly valuable in large public institutions, where the nominally shortest route may not be the most reliable route in practice. For example, a corridor may be frequently occupied by waiting people, temporary equipment, cleaning activities, or queues near a service area. Likewise, an entrance or elevator lobby may be consistently difficult to approach because of narrow doorways or congestion at particular times of day. In such cases, a slightly longer route may be preferable if it is more predictable and less prone to delays.

To avoid permanently penalising a connection after an isolated event, the recorded statistics could use time-decayed costs. A temporarily blocked edge would therefore become less attractive for a period, but would gradually return to its nominal cost once it could be traversed normally again. The planner should also retain fallback routes, so that a high-cost connection is avoided when possible but remains available if it is required to reach the destination.

Overall, this extension would complement the existing reactive obstruction handling with a lightweight form of long-term route adaptation. The mission layer would continue to replan around immediate blockages, while also learning which connections are consistently unreliable and incorporating that experience into future route selection.

References

[1] "Robots in healthcare: A solution or a problem?" European Parliamentary Research Service, Brussels, Tech. Rep., 2018, https://epthinktank.eu/2018/12/06/robots-in-healthcare.

[2] J. H. Lee, J. W. Kim, and S. H. Kim, "User perception of medical service robots in hospital wards: A cross-sectional study," *Journal of Yeungnam Medical Science*, vol. 39, no. 2, pp. 116–123, 2022, ISSN: 2801-8887. DOI: 10.12701/yujm.2021.01319

[3] M. Liu et al., "Campus guide: A lidar-based mobile robot," in *2019 European Conference on Mobile Robots (ECMR)*, IEEE, 2019, pp. 1–6. DOI: 10.1109/ECMR.2019.8870916

[4] T. Zhu, H. S. Ahn, E. Broadbent, B. A. MacDonald, and N. Gasteiger, "A deep dive into human-robot interaction in hospitals: Scoping review on the services provided, engagement behaviours and interaction outcomes," *International Journal of Social Robotics*, vol. 17, no. 8, pp. 1541–1561, 2025, ISSN: 1875-4791. DOI: 10.1007/s12369-025-01299-2

[5] K. Cai, C. Wang, J. Cheng, C. W. de Silva, and M. Q.-H. Meng, *Mobile robot path planning in dynamic environments: A survey*, arXiv:2006.14195, Accessed October 31, 2026, 2020. DOI: 10.48550/arXiv.2006.14195 [Online]. Available: https://arxiv.org/abs/2006.14195

[6] Y. Tang, M. A. Zakaria, and M. Younas, "Path planning trends for autonomous mobile robot navigation: A review," *Sensors*, vol. 25, no. 4, p. 1206, 2025, ISSN: 1424-8220. DOI: 10.3390/s25041206

[7] T. Kim, G. Kang, D. Lee, and D. H. Shim, "Development of an indoor delivery mobile robot for a multi-floor environment," *IEEE Access*, vol. 12, pp. 45 202–45 215, 2024, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2024.3381489

[8] W. Wen, X. Bai, and L.-T. Hsu, "3d vision aided GNSS real-time kinematic positioning for autonomous systems in urban canyons," *NAVIGATION: Journal of the Institute of Navigation*, vol. 70, no. 3, navi.590, 2023, ISSN: 0028-1522. DOI: 10.33012/navi.590

[9] ROS Navigation Stack Team, *Setup and configuration of the navigation stack on a robot*, http://wiki.ros.org/navigation/Tutorials/RobotSetup, Accessed July 30, 2026, 2020.

[10] ros1_bridge Authors, *Ros1_bridge: Bridge communication between ROS 1 and ROS 2*, https://docs.ros.org/en/foxy/Tutorials/Intermediaries/ros1_bridge.html, Accessed May 1, 2026, 2026.

[11] A. Elkady and T. Sobh, "Robotics middleware: A comprehensive literature survey and attribute-based bibliography," *Journal of Robotics*, vol. 2012, no. 1, p. 959 013, 2012. DOI: 10.1155/2012/959013

[12] R. Stricker et al., "Interactive mobile robots guiding visitors in a university building," in *Proceedings of the 21st IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, Paris, France: IEEE, 2012, pp. 695–700. DOI: 10.1109/ROMAN.2012.6343832

[13] J. Palacín, E. Rubies, R. Bitriá, and E. Clotet, "Path planning of a mobile delivery robot operating in a multi-story building based on a predefined navigation tree," *Sensors*, vol. 23, no. 21, p. 8795, Oct. 2023, ISSN: 1424-8220. DOI: 10.3390/s23218795

[14] A. Sifer and collaborators, *Intellwheels_nav: Navigation stack for the intellwheels wheelchair platform*, https://github.com/siferati/intellwheels_nav, Accessed July 30, 2026, 2016.

[15] R. Casimiro and R. Nunes, *Routing people and vehicles inside buildings*, Extended abstract, Accessed October 31, 2026, Oct. 2016. [Online]. Available: https://fenix.tecnico.ulisboa.pt/downloadFile/1407770020544652/ExtendedAbstract.pdf

[16] S. Wang, C. Martínez Almansa, J. Peña Queralta, Z. Zou, and T. Westerlund, "UWB-based localization for multi-UAV systems and collaborative heterogeneous multi-robot systems," *Procedia Computer Science*, vol. 175, pp. 357–364, 2020, ISSN: 1877-0509. DOI: 10.1016/j.procs.2020.07.051 [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050920317324

[17] H. J. Chang, C. S. G. Lee, Y. C. Hu, and Y.-H. Lu, "Multi-robot SLAM with topological/metric maps," in *Proceedings of the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2007, pp. 1467–1472. DOI: 10.1109/IROS.2007.4399142

[18] W. Liu, R. Li, and Z. Deng, "A review of topological map construction methods for indoor robot localization and navigation," in *Proceedings of the Work-in-Progress Papers at the 14th International Conference on Indoor Positioning and Indoor Navigation (IPIN-WiP 2024)*, CEUR Workshop Proceedings, Vol. 3919, Hong Kong, China: CEUR Workshop Proceedings, Oct. 2024. [Online]. Available: https://ceur-ws.org/Vol-3919/short26.pdf

[19] P. S. Madhukar and L. B. Prasad, "State estimation using extended kalman filter and unscented kalman filter," in *2020 International Conference on Emerging Trends in Communication, Control and Computing (ICONC3)*, Lakshmangarh, India: IEEE, 2020, pp. 1–4. DOI: 10.1109/ICONC345789.2020.9117536

[20] Luxonis, *Oak-d lite hardware specifications*, https://docs.luxonis.com/hardware/products/OAK-D%20Lite, Accessed: 2026-08-31, 2026.

[21] ROS Perception, *depthimage_to_laserscan: Converts a depth image to a laser scan for use with navigation and localization*, https://github.com/ros-perception/depthimage_to_laserscan, Accessed: 2026-08-31, 2021.

[22] Open Robotics, *Sensor_msgs/laserscan message*, https://docs.ros2.org/foxy/api/sensor_msgs/msg/LaserScan.html, Accessed: 2026-08-31, 2020.

[23] Midemig, *gps_amcl: AMCL ROS package reimplementation for using GNSS data*, https://github.com/midemig/gps_amcl, Accessed August 31, 2026, 2020.