Universidade de Brasília
Faculdade de Tecnologia

A framework for distributed simulated experiments in multi-robot systems

Vicente Romeiro de Moraes

PROJETO FINAL DE CURSO

ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Brasília

2023

Universidade de Brasília
Faculdade de Tecnologia

Um arcabouço para experimentos simulados distribuídos em sistemas multi-robô

Vicente Romeiro de Moraes

Projeto Final de Curso submetido como requisito parcial para obtenção do grau de Engenheiro de Controle e Automação

Orientadora: Profa. Dra. Genaina Nunes Rodrigues

Brasília
2023

Romeiro de Moraes, Vicente.

R763a A framework for distributed simulated experiments in multi-robot systems / Vicente Romeiro de Moraes; orientadora Genaina Nunes Rodrigues. -- Brasília, 2023.
75 p.

Projeto Final de Curso (Engenharia de Controle e Automação) -- Universidade de Brasília, 2023.

1. Sistema multi-robô. 2. Simulação robótica. 3. Computação Distribuída. 4. Conteinerização. I. Nunes Rodrigues, Genaina, orient. II. Título

Universidade de Brasília
Faculdade de Tecnologia

A framework for distributed simulated experiments in multi-robot systems

Vicente Romeiro de Moraes

Projeto Final de Curso submetido como requisito parcial para obtenção do grau de Engenheiro de Controle e Automação

Trabalho aprovado. Brasília, 24 de julho de 2023:

Brasília
2023

Abstract

Uncertainties about the environment are an inherent aspect of multi-robot systems. It is essential to have a stable, reproducible and ideally simple simulation environment so one can be sure of their robots' ability to perform their mission before deploying on a physical system. However, robotics simulators are often unreliable and too unstable for any kind of large-scale testing in more complex systems. Virtual distributed computer environments offer a way to provide a reliable and reproducible simulation without modifying the underlying simulation software, with also the added bonus of distributed computing for more resource-intensive experiments. For this project, we developed a programmable automation tool for containerised simulation environments in Python to ease the development process of robotic experiments with ROS 2 in a multi-robot setting. Using Docker containers we can isolate both the development and the deployment of an application thus providing a modular microservices approach to robotics simulation. By creating Python class wrappers around the Docker API we then provide a DSL (Domain Specific Language) for creating and running simulations in distributed Docker containers across multiple hosts. By these means, developers can run their multi-robot simulations in a more simplified, integrated and computationally distributed environment.

Keywords: Multi-robot System. Robotics Simulation. Distributed Computing. Containerisation.

Resumo

Incertezas sobre o ambiente de execução são um aspecto interente à sistemas multi-robô, é essencial ter um ambiente de simulação estável, reprodutível e idealmente simples para que tenhamos certeza da capacidade dos robôs de executar sua missão antes de serem feitos testes em campo. Entretanto, simuladores em robótica são comumente muito instáveis e não confiáveis para testes em larga escala em sistemas mais complexos. Ambientes de computação virtualizada e distribuída oferecem uma forma de prover uma simulação confiável e reprodutível sem modificar a estrutura interna do software de simulação, com um bônus adicional de computação distribuída para experimentos que são muito taxantes computacionalmente. Para esse projeto desenvolvemos uma ferramenta programável para automação de ambientes de simulção containerizados em Python para faciliar o desenvolvimento e execução de experimentos em robótica, utilizando o ROS2 no contexto de multi-robô. Nossa abordagem utiliza contêineres Docker para isolar tanto o ambiente de desenvolvimento quanto o ambiente de implementação provendo então uma abordagem de microsserviços modular para simulações em robótica. Criando wrappers ao redor da API do Docker com classes em Python, provemos uma DSL (Domain Specific Language) para construir e executar simulações em contêineres Docker distribuídos. Com isso, desenvolvedores conseguem criar suas simulações multi-robos de uma forma mais simplificada, integrada e computacionalmente distribuída em um único ambiente.

Palavras-chave: Sistema multi-robô. Simulação robótica. Computação Distribuída. Conteinerização.

List of Figures

List of Tables

List of abbreviations and acronyms

API	Application Programming Interface	15
BT	Behaviour Tree	17
DBT	Dynamic Behaviour Tree	17
DSL	Domain Specific Language	12
FSM	Finite State Machine	17
GUI	Graphical User Interface	25
HFSM	Hierarchical Finite State Machine	17
HMRS	Heterogeneous Multi-Robot System	29
NPC	Non-Playable Character	17
RMF	Robot Middleware Framework	25
ROS	Robot Operating System	13
SDK	Software Development Kit	15
STDOUT	Standard Output Stream in Linux	33
TB3	Turtlebot3	24
TB4	Turtlebot4	24
UUID	Universally Unique IDentifier	45

Contents

1 Introduction

The development of robotic applications can be quite expensive and mistakes on a production environment can be incredibly costly. In multi-robot systems, simulators are a must in developing and verifying an application in order to remove errors and minimise costs. However, experimentation and testing in robotics can also be quite challenging (AFZAL; GOUES, et al., 2020) given the complex nature of cyber-physical robotic systems. Simulators fill that niche where one can test their execution scenario against physical and logical constraints.

Nevertheless current robotics simulators are often not scalable enough to run hundreds or thousands of execution scenarios. In (AFZAL; KATZ, et al., 2020) they conduct a survey where robotics researchers answer questions about the biggest challenges facing robotics simulations. Among those challenges, it is worth noting that simulator environments are non-reproducible and unstable. Setting up an environment takes too long and executing the simulation is error prone and resource intensive. Additionally, most simulators yield non-deterministic results, which require a proper experimentation setup for replication. Therefore, frameworks that leverage reproducibility and performance of robotics simulations are a must.

In previous work (RODRIGUES et al., 2022), we provided a distributed simulation with Docker using the eight machines available in the lab. In total, we were able to run 1296 simulated trials using those eight machines in less than 24 hours which was quite an achievement. However there were several problems in orchestrating complex experiments with docker-compose (GUILLAUME LOURS, 2023), particularly regarding the management and automation of the startup and shutdown processes of containers involving multiple scripts in bash and python (GABRIEL ARAÚJO, 2021). Additionally, timing issues between the robot launch and the simulation environment happened at times. Last but not least, upgrading our software environment, such as the Morse simulator and ROS1, to more up to date versions with Gazebo and ROS2 required a cumbersome rework.

1.1 Objectives

Our objective in this work is to contribute a flexible, reliable and stable framework for robotics simulation in a distributed environment. Therefore, this project aims to explore virtual distributed computer environments for robotic simulations by providing a programmable system aimed at multi-robot systems implemented in the ROS middleware.

Our aim in this work is to ease the development of experiments in robotics, such as the experiment perform in MissionControl (RODRIGUES et al., 2022).

1.2 Contributions

To achieve those objectives, the contributions of this work are as follows:

i A simple DSL (Domain Specific Language) for running robotics experiments – We created a DSL in Python to configure, build and execute containerised simulation environments that is accessible and intuitive to use. The goal is to ease the deployment and execution of robotic applications with a configurable environment that's moldable to the researcher's needs.

ii A distributed runtime environment – We created a plugin architecture where docker containers are available as plugins for the developer to include in his simulation files. Therefore, with a single line of code it is possible to add robots with specific capabilities in a containerised environment in a way that is simpler and more scalable than a *docker-compose* system (GUILLAUME LOURS, 2023). We provide both prebuilt classes for many common programs as well as more general meta classes if a developer needs to make their own plugins.

iii An experimentation setup for reproducibility of experiments – We created a containerised simulation plugin that allows modularity and reproducibility to robotics system by isolating the execution environment. Analogous to a microservices architecture, we can think of different components in a robotics simulation (e.g. robots, sensors, actuators) as modular pieces of a larger system. By these means, developers can configure their simulation through a Python file where each component is represented within a class.

1.3 Document Structure

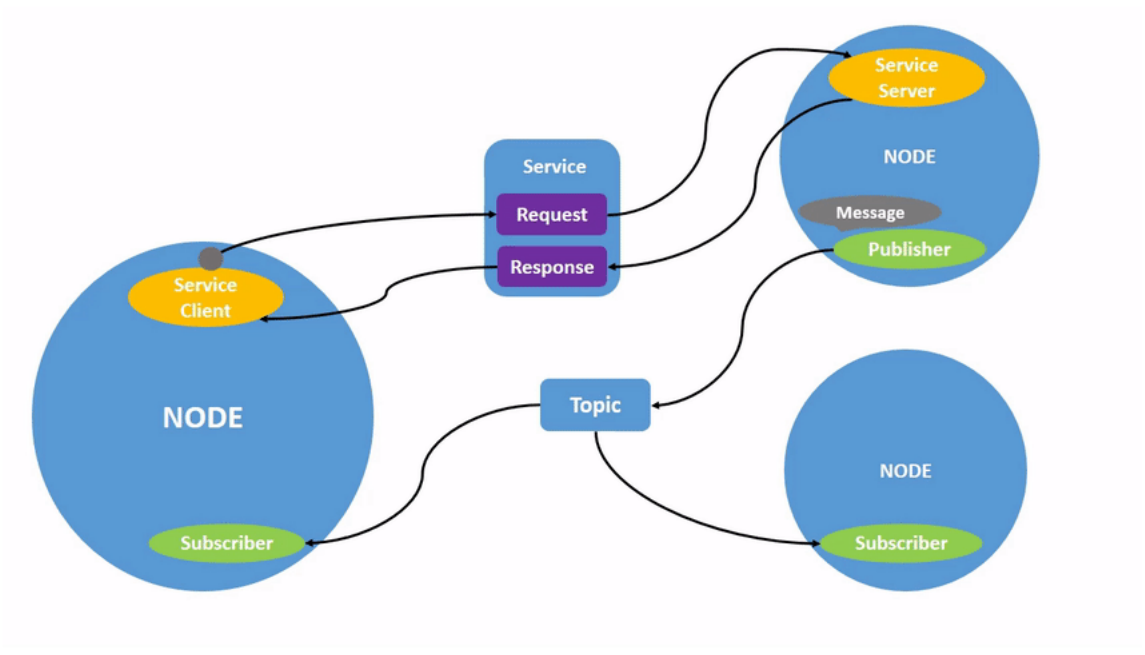
The remaining chapters are organized as follows. In Chapter 2 we present the technical background required for this work. In Chapter 3 we detail the architecture behind our framework and in Chapter 4 we show a couple examples of what's achievable with our framework and perform the MissionControl experiment.

2 Technical Background

In this chapter we introduce concepts related to the underlying robotics simulation, how robots perform tasks in the environment, how robots navigate, how different robotic components communicate, etc. We will also discuss how the docker API works and provide an explanation on how we use docker to create distributed multi-robot experiments.

2.1 ROS

Figure 2.1 – Communication between multiple nodes in ROS2.



Source: (UNDERSTANDING. . . , 2020)

ROS (QUIGLEY et al., 2009) is a middleware that aids software development for robotic applications particularly for communication between sensors, actuators, controllers and many other components for robots. In this project we will be using ROS2 to establish a communication network between different components and modules in our simulation.

Generally a simulation based on ROS is structured around nodes such that each node is responsible for a single unique purpose. This inherently provides ROS with good modularity and indeed one of its great advantages is allowing roboticists to create essentially "universal algorithms" for robotics without worrying about specific hardware capabilities and relying on the manufacturers to develop drivers to integrate ROS to a control device.

ROS works through a publisher-subscriber architecture (Fig 2.1). Wherein publisher nodes create topics where they publish information which may or may not be read by a subscriber node which follows the topic. Unlike traditional server-client architecture, the publisher is not aware of its subscribers.

It is also possible to have a server-client architecture using ROS services which can be called and interacted with through a different protocol, as well as ROS action servers where a client can request an action from a server and receive feedback from the action being executed.

One can think of many examples for such applications. Navigational systems, for instance, are usually represented by action clients as the feedback may be used to improve the planning and routing algorithms (GUIMARÃES et al., 2016). One can also think of classic control structures with a feedback loop, which can be more closely modeled by the ROS2 control framework (ROS-INDUSTRIAL, 2023).

ROS facilitates software development by standardising digital communication using topics and standard message libraries for publishing on topics and services. It also offers a vast collection of rich and mature packages, tools and frameworks, many of which have been in development since the inception of ROS in 2007.

ROS2 (MACENSKI; FOOTE, et al., 2022) is the second generation of ROS and is the leading robotics middleware, widely used in both academia and industry. ROS2 is built on top of the DDS (Data Distribution Service) communication standard which provides robots and their components safe and reliable communication without a dedicated network setup, offering much more robustness and dependability for industrial and critical applications (MARUYAMA; KATO; AZUMI, 2016).

We aim to use the new ROS2 software over the old version as ROS2 streamlines and simplifies a lot of the development of ros packages, as well as offering a less intrusive terminal interface and more up to date packages. The first version of ROS is also set to be deprecated in 2025 with the last release of ROS Noetic.

In this project we will also use the ROS2 framework nav2 to handle autonomous navigation for the robots, as well as different ros packages for simulation, modelling and deploying robotic systems.

2.2 Docker

Docker (MERKEL, 2014) is one of the main software used in containerisation of applications. In the context of robotic applications it offers a way to virtually isolate an environment for programming and deploying software for cyber-physical systems.

Containerisation means we can essentially save a snapshot of our environment and reproduce it in any machine. Containers are a type of lightweight virtual machines that separate an environment from its host machine, they offer a lot of the advantages of virtualisation while requiring less computational resources.

In this project docker is instrumental in the design of a modular programmable architecture for robotic simulations where different components can be assigned via python and containerised for a distributed execution environment. With this we can guarantee reproducibility of the simulations in any linux system running docker and we can offer potential robotics developers a jump-start on modularising their applications.

Our modular container approach for the design of robotic applications has more customisation options when compared to the traditional docker-compose program. In order to change an attribute, usually represented by an environment variable in docker, the user could have to navigate through a large and potentially quite complex *yaml* file, which could be very cumbersome for a programmer particularly if one is inexperienced with docker.

By using the python docker sdk (DOCKER, 2023) we can send build or run commands directly to the container process in the docker API, so we can write wrappers that add or modify behaviour and perform more complex functionality than what is possible using a *yaml* file, thus giving the user more freedom on how to customise his application.

2.2.1 Docker API and SDK

The docker API works by providing what is called a docker workflow, through which a docker client can make HTTP requests to the api server and then perform an action. Namely a workflow is a series of actions taken on registries, images or containers (API..., 2016).

The docker client requires either a remote or a local daemon listening to the API calls, remote daemons can be given actions through ssh or other ways to proxy web traffic. By using the python sdk we can control the client directly and programmatically create our own API calls and workflow.

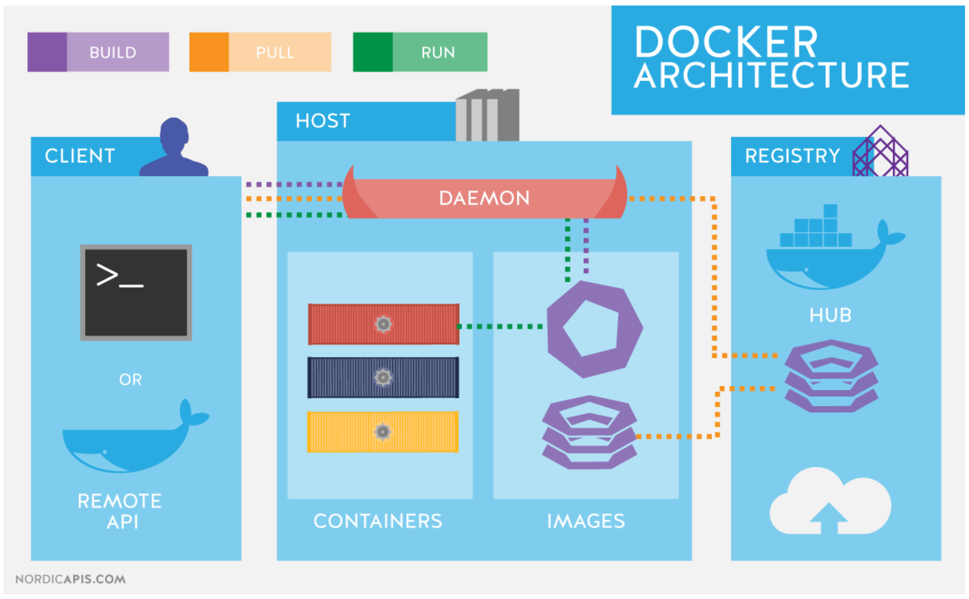
Through the workflow, one can create images, which are templates from which lightweight virtual machines called containers can be modelled after. Containers ideally offer the bare minimum environment necessary to run an application and therefore are much simpler and faster than having an entire virtual computer. Through those containers we can achieve an isolated environment and in our case achieve a reproducible simulation.

Inside those "*mini*"-virtual machines we can also provide additional functionality in the form of environment variables set up inside the container as well as volumes which are memory blocks which can be mounted from the host computer into a container or between two different containers. That way we can share files inside our application.

Traditionally simple systems can be implemented in a single Dockerfile that represents the docker workflow for the entire application, while more complex and modular systems can use a docker-compose (GUILLAUME LOURS, 2023) yaml file which builds and manages multiple containers simultaneously.

As mentioned previously the python sdk is yet another abstraction over the usual docker client and allows us to orchestrate our own API calls. In the context of a robotics simulation this is quite useful as it allows us to customise our operation in runtime as well as simplifying the development, deployment and execution of the simulation.

Figure 2.2 – Docker API flow diagram.



Source: (API..., 2016)

2.3 Robot Skills and Capabilities

For this project we define robotic capabilities as actions or tasks able to be performed by actuators present in the robot. This includes but not limited to:

- Interacting with other robots or components through ROS;
- Interacting with humans through a display panel or through synthesised speech;
- Displaying sensor data to another machine or to a human;
- Grabbing or otherwise handling physical objects through a specific component, such as a robotic arm, or by any other means;
- Navigating through its environment;

- Opening/Closing doors and cabinets;
- Operating an elevator.

It’s also important to note that although the robot may have the capability for human interaction it should be able to perform all of its innate capabilities autonomously, i.e. a human can order a moving robot to navigate to a specific room, but the robot should be able to navigate to said room regardless if ordered by a human or another machine.

For this projected skills are implemented with software abstractions through which we control the capabilities of each robot. Skills may be implemented either as contextually aware behaviour tree models using the BehaviorTree.CPP framework (BEHAVIORTREE. . . , 2020) or a simpler implementation using python functions directly.

Using skills we can observe and decompose tasks into simpler skills. Then with skill decomposition and sequencing turn them into achievable steps the robot must take to complete its missions. Skills can be executed sequentially or in parallel and more complex skills can be reduced into other operations.

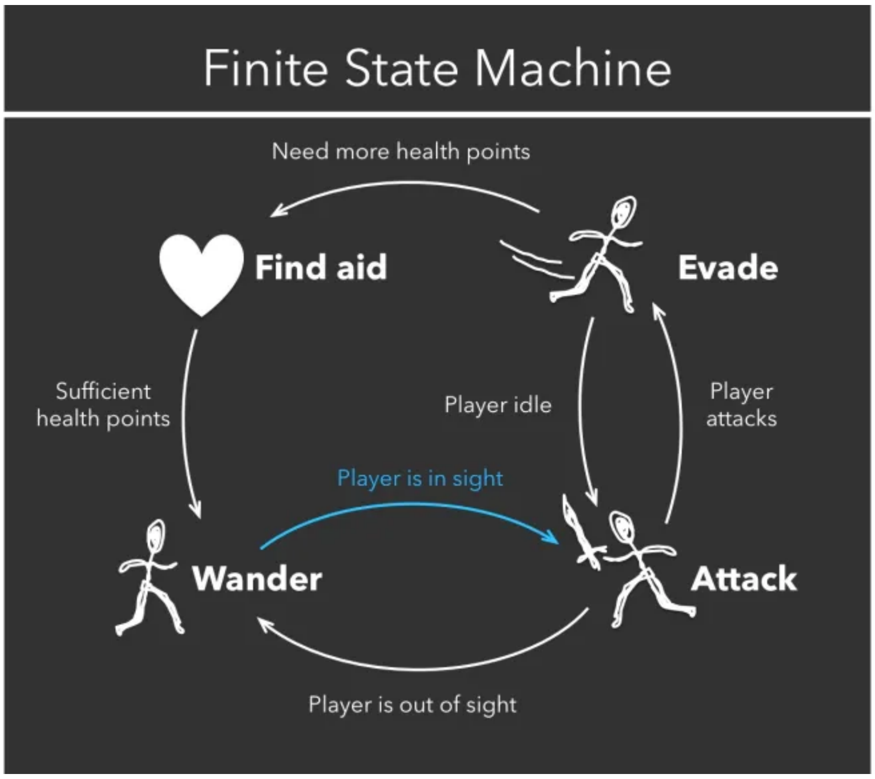
2.3.1 Behaviour Trees

Behavior Trees in Robotics and AI (COLLEDANCHISE; ÖGREN, 2018) defines Behaviour Trees as "a way to structure the switching between different tasks in an autonomous agent, such as a robot or a virtual entity in a computer game". BTs occupy a similar place to FSMs (Finite State Machines) as a control structure. However as we will discuss, they offer much more in terms of robustness and modularity to an application.

Behaviour Trees or HFSMs (Hierarchical State Machines) were initially created for controlling game AI and NPCs (ISLA, 2005) as a way of better modelling complex behaviour and creating more interesting scenarios and interactions for the player. More generally, BTs can be used to express both modelling and implementation characteristics for any autonomous agent system, while traditional FSMs struggle with implementation particularly in terms of properly expressing state transitions.

Traditionally FSMs are built as a connected graph of nodes representing states and links representing state transitions. Each link is associated with a certain trigger condition which changes the current state of the system. We can already spot one of the major issues with implementing FSMs which is in order to change a single state we would have to change all of its links and quite often that would also imply changing the entire control structure.

Figure 2.3 – Finite state machine for a simple shooting video game.



Source: (LOU, 2017)

In the figure 2.3, we can also infer that FSMs are inherently not scalable. Even simply adding different attack or wander patterns would result in multiple different state transitions with potentially many of them being redundant or repetitive. Now imagine the same issue, but in more complex systems with thousands of states and transitions and the whole structure becomes unreasonably expensive to maintain.

BTs fundamentally differ from FSMs in that they have an inherent hierarchical structure which implies a certain order of operation. They are represented as a tree connected graph where nodes can be either roots, i.e. they have no parent nodes above them, or leafs or non-leafs, a leaf node has no child nodes beneath them. Henceforth we will also be using the BT notation used in (COLLEDANCHISE; ÖGREN, 2018) which is also used for the BehaviorTree.CPP (COLLEDANCHISE; ÖGREN, 2017) framework to be discussed later in this section.

Leaf nodes usually imply a certain behaviour which is an action taken by the agent, such as picking up an object or interacting with the environment. When executing said action they will return a Tick to its parent indicating its current state. Ticks are what causes state transitions inside a BT and they can either be Running, as in the child node is currently executing its action, or Success or Failure implying the result of the action being completed.

Leaf nodes are an example of an execution node which are typically classified as either Action or Condition nodes, where Condition nodes return Success if a certain predicate holds and Action nodes are self-explanatory.

There is also the case of Reactive versus Non-Reactive BTs, where the non-reactive kind does not have a Running tick and only cares about the result of an action. Non-reactivity has several drawbacks (MILLINGTON; FUNGE, 2009) and are not useful in the context of this project in implementing BTs as skills for robotics, as reacting to feedback received from actions is essential in control systems.

Non-leaf nodes are subdivided into 4 different types and mostly imply making a certain decision about its child nodes. Specifically according to (COLLEDANCHISE; ÖGREN, 2018) those are:

- Sequence Nodes will execute its children nodes from left to right until all of them return a Success Tick. If any child returns a Failure or Running tick the node will propagate the same tick upwards. Therefore it returns Success iff all its children return Success.
- Parallel Nodes routes Ticks to all its children and returns Success if a certain number M of its N children returns Success, $M \leq N$. It returns Failure if N-M+1 children return Failure and otherwise it returns Running.
- Fallback Nodes will execute its children nodes from left to right until any child returns Success or Running then it will return Success or Running. Therefore it only returns Failure iff all its children return Failure.
- Decorator Nodes have a single child and alter the Tick of its child according to a user specified rule. For instance one could have an inverting decorator that only returns Success if its child returned Failure.

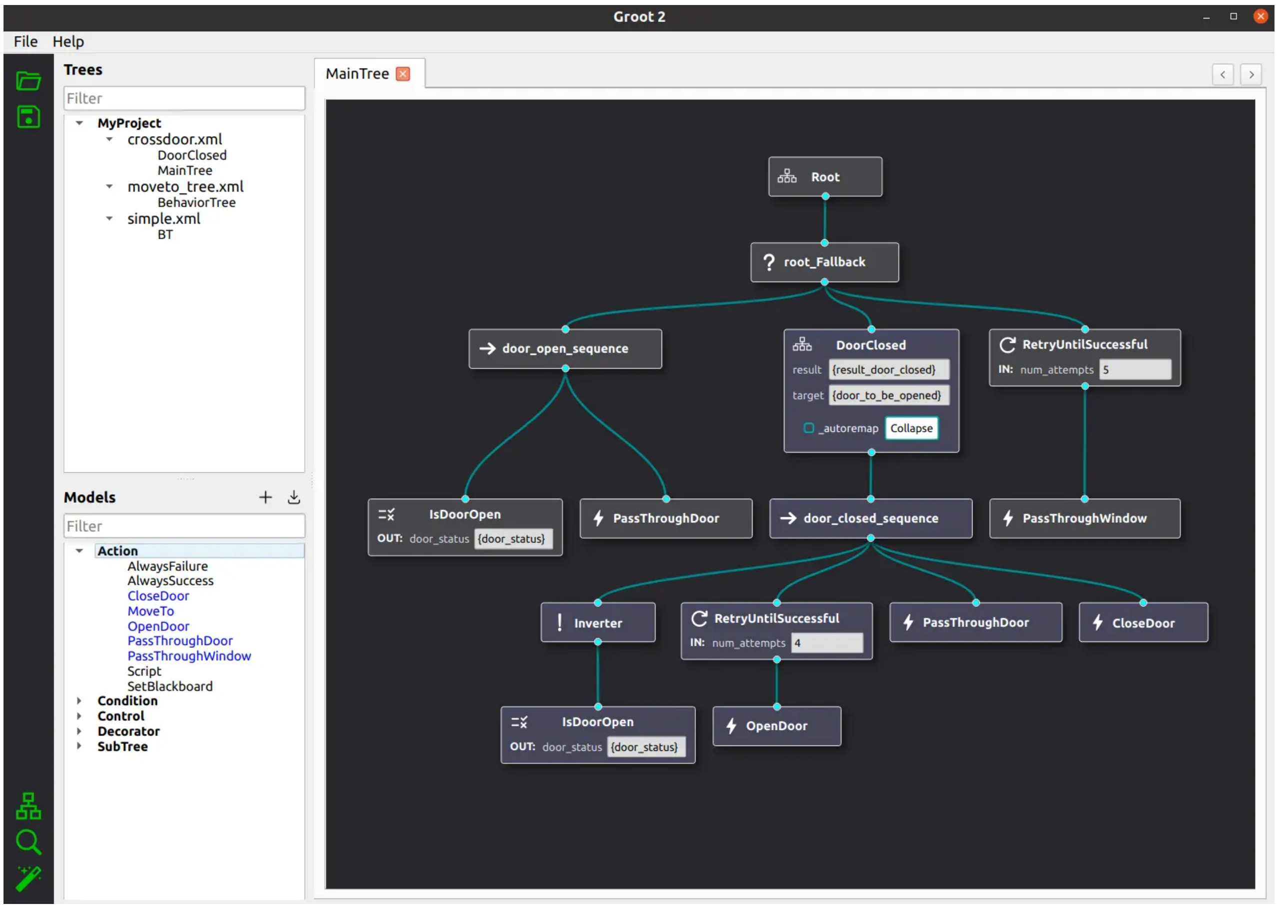
Finally, as with many AI systems, BTs have a type of memory in the form of a Blackboard which is a series of variables, essentially a dynamic database which is accessible by any node inside of the tree.

In the context of robotics BTs have also been widely used (ÖGREN; SPRAGUE, 2022) precisely because what makes BTs useful for autonomous agents are also extremely useful when incorporated into robotics systems. That is, robotic systems have classically been built on top of modular feedback control structures which naturally lend themselves quite well to a BT behaviour. It's also important to note that a hierarchy of tasks is fairly common in robotics as complex tasks can often be decomposed into a sequence of simpler tasks which also aligns quite well with the philosophy behind BT development.

Previously this project used Pytrees (SPLINTERED-REALITY, 2023) as the chosen BT implementation for python. However, we found that BT.CPP (COLLEDANCHISE; ÖGREN, 2017) was the better of the two in terms of documentation as well as being a companion of the excellent book *Behavior Trees in Robotics and AI* (COLLEDANCHISE; ÖGREN, 2018).

In general, BT.CPP has better integration with ROS2, especially with the nav2 navigation framework. Developing behaviours with the graphical tool *Groot* (FUNDACIO EURECAT, 2023) is also much more intuitive and easier than Pytrees. In figure 2.4 below we can see an example of a behaviour tree made with Groot.

Figure 2.4 – Behaviour tree for a robot opening a door.



Source: (FUNDACIO EURECAT, 2023)

2.3.2 Navigation

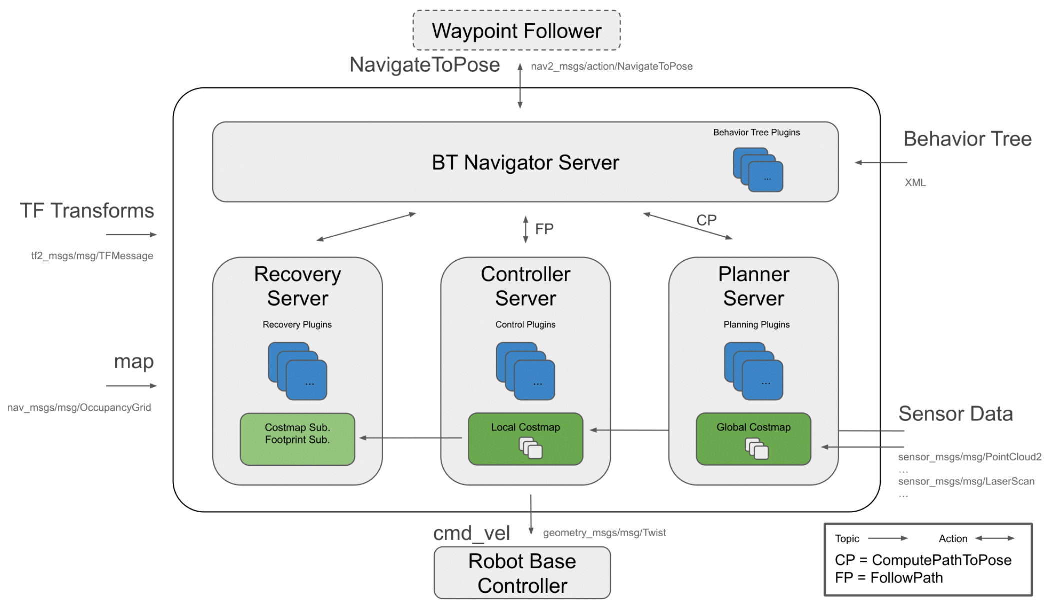
Navigation is one of the main aspects we aim to control with robotic skills. Nav2 (MACENSKI; MARTÍN, et al., 2020) is the most used navigation framework for ROS2 and designed to be able to work generally for autonomous navigation with a large variety of robots in many environments.

To do that they employ a new feature in ROS2 called lifecycle nodes which essentially comprise dynamic processes with complete and clear state transitions, meaning from instantiation through destruction, that allows the navigation server to be a reliable FSM in conjunction with another BT server that provides the navigation behaviour. That way we have the modularity of a BT architecture that's also highly reliable alongside ROS2 internal DDS communication.

Therefore navigation is entirely expressed within BTs internally in nav2 and that means that as a user we have complete customisability over the entire navigation stack which is subdivided into 3 different servers: Control, Planning and Recovery. User defined BTs that implement navigation behaviour occupy the highest level and have control over all layers of the stack.

The Control and Planning Layer work in tandem to compute navigation paths. The global planner computes the shortest path from the initial robot position to its goal while the control planner manages local short path estimations and controls the robot motion based on sensor data, usually LIDAR.

Figure 2.5 – Nav2 framework overview.



Source: (STEVE MACENSKI, 2023)

We can think of robotics navigation by using an analogy to how humans navigate. Suppose a person wants to go to their local grocery store. The first step would be to use their knowledge on their area and plot an optimal path from their home to the grocery store. This implies that the person is at least roughly aware of their street layout and knows which way to move in order to reach their destination efficiently. That is the equivalent of our robot's global planner, it analyses the known map data and plots a route through to its destination.

However once we plot our destinations we are not aware of every obstacle in the path we have chosen. Once we are navigating through the streets we may encounter a multitude of obstacles, such as cars, trees, lamp posts that we must avoid using our sensory information. And that's what the local planner treats in our navigation stack, it uses the laser sensor information to establish a costmap on nearby objects. It then assigns a weight based on the footprint or the rough size of our robot, and makes tiny changes to the global path in order to avoid collisions with nearby objects.

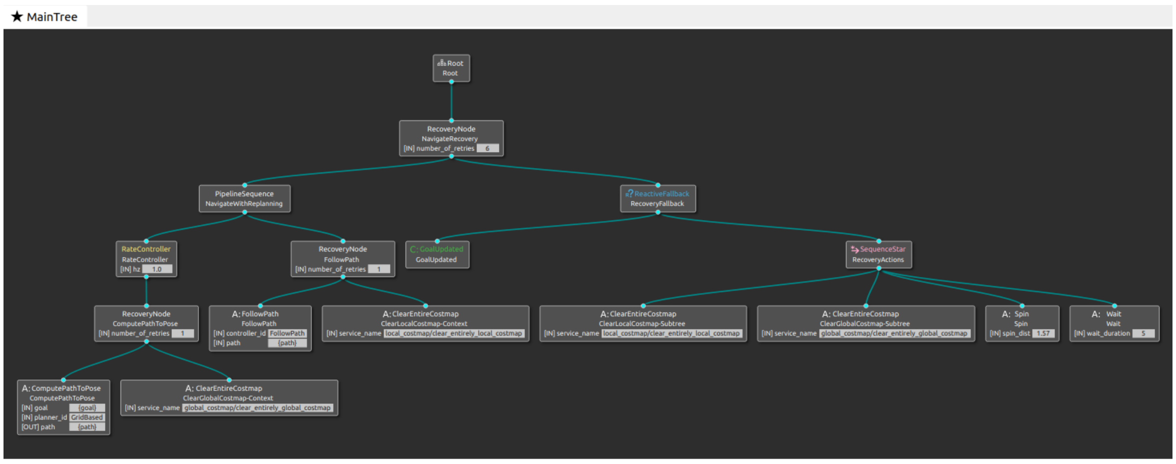
Another important aspect of navigating a robot is localisation. Through ROS we represent the rigid body dynamics of our robot as a tree of transformation coordinates. Each joint and moving part of our robot is represented by a coordinate system, alongside its proper moment and inertia as defined in the robot descriptor file. And then movement of our robot is represented by transformations applied through the tree.

In order to achieve robot localisation we start with a fixed coordinate, which is our map frame, then the odometry, that is the relative displacement of our robot in relation to the fixed map frame, is computed as the robot moves and is propagated to the transform tree. The navigation stack uses a monte carlo simulation called amcl to simulate the current position of our robot and align it with the transform tree. That way we can know roughly where our robot is and its sensors can know where they are located in relation to the robot so we can have accurate sensor data.

Finally there's the Recovery server which interacts with either global or local navigation in order to avoid a complete failure. Recovery behaviours implement a certain action to be taken usually after a Failure Tick is returned by a navigation node or as part of a Fallback node. An example of a local recovery behaviour would be the robot spinning in place to try to free itself, while a global recovery behaviour can be recalculating the global route or the costmap function.

Below in figure 2.6 we can see an example of the default running nav2 behaviour tree.

Figure 2.6 – Nav2 behaviour tree with replanning and recovery behaviours. Shown in groot



Source: (STEVE MACENSKI, 2023)

2.3.3 DepthAI and OpenCV

Though it is not explicitly the focus of this project we also provide skills related to computer vision in the form of a bridge between ros messages and OpenCV as well as skills related to the Depth AI platform.

OpenCV (BRADSKI, 2000) is one of the most widely used libraries for real-time computer vision and provides numerous functions to analyse and operate on image data. If a robot has a camera, which is true for both TB3 and TB4 robots, then it possible to use the real or simulated camera data with openCV using the vision opencv package (KENJI BRAMELD, 2023).

DepthAI (LUXONIS, 2023) is a more recent SpatialAI tool developed by Luxonis which also has many uses relating to depth perception and image segmentation, such as identifying a specific object in the screen region or estimating an object spatial coordinates. DepthAI is only available for their own custom developed cameras the OAK brand so it's usable on the TB4 robot which has an OAK-D Lite camera.

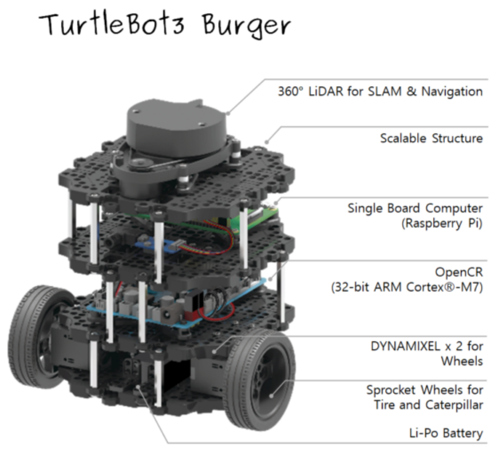
For this project in particular, we use OpenCV to help identify objects which are close to the camera based on the size of their silhouettes and we also use DepthAI to help identify the pose of specific objects in the screen like a person.

2.4 Robots and Simulators

We aimed to support the most used robots and simulators for open source robotics, but we also provide generic plugin templates for robots and simulators.

2.4.1 Turtlebot3 and Turtlebot4

Figure 2.7 – Illustrative pictures of TB3 and TB4


TurtleBot3 Burger
360° LiDAR for SLAM & Navigation
Scalable Structure
Single Board Computer (Raspberry Pi)
OpenCR (32-bit ARM Cortex®-M7)
DYNAMIXEL x 2 for Wheels
Sprocket Wheels for Tire and Caterpillar
Li-Po Battery

Turtlebot3 Burger Turtlebot4 Lite

Source: (ROBOTIS, CLEARPATH ROBOTICS, 2023)

Both Turtlebots (ROBOTIS, CLEARPATH ROBOTICS, 2023) are two wheeled mobile robots that offer a depth camera, a 360° LIDAR sensor and an internal raspberry pi. The TB3 is the older version of the turtlebot platform, but it's as of writing more stable and better supported than the TB4.

However the TB4 is more integrated with the ROS2 environment with official launch packages for navigation, simulation, image manipulation and more. Nonetheless both versions are supported.

2.4.2 Gazebo Classic and Gazebo Ignition

We also support both versions of the Gazebo (KOENIG; HOWARD, 2004) simulator. Gazebo (previously known as Gazebo Ignition) is the new version of the simulator and improves on a lot of functionality over the previous version particularly in terms of performance.

Gazebo Ignition is also more ROS agnostic and implements an internal communication protocol that in many ways supersedes ROS functionality. Ignition provides an internal structure for topics, services and actions so it's necessary to use a bridge to translate messages from ROS to the equivalent Ignition message. For our TB4 simulation a namespaced bridge is already provided.

2.5 Other Middlewares

2.5.1 RMF

RMF (Robot Middleware Framework) (OSRF, 2023) is another alternative framework for organising and planning robotic tasks. It provides an online interface for interacting with tasks and assigning different missions to multiple fleets of robots.

It aims to assist the interoperability of heterogeneous fleets of robots and provide a stable interface for service robot missions. For that end they provide multiple tools for traffic management which are particularly useful for our simulation in general.

We provide a general template to run RMF demos and tools, optionally with a GUI (Graphical User Interface), through the docker system.

In this chapter we discussed how the simulation works internally with ROS and robotic skills used to perform a mission or task in a experiment scenario. Next we discuss our framework and how we construct an architecture to user containerisation for distributed experiment execution.

3 Our Framework

As per the design goals, we aim in this work to create a framework that is useful for simulations in a research context. To that end, this work proposes a framework with a plugin-based architecture to implement a DSL in Python for reproducible experiments in robotics.

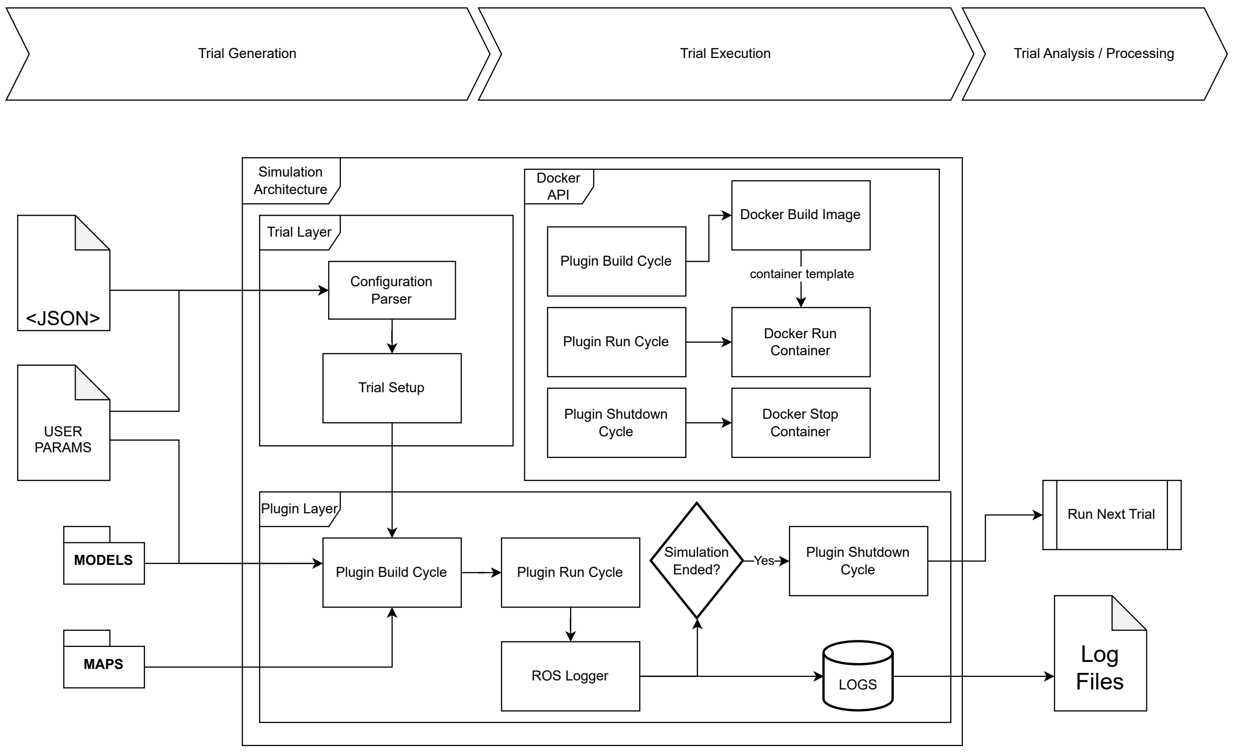
Before delving further into our work, we provide the definition of the following terms used throughout our framework:

1. Component: A component is the most abstract form of an object. It defines an object in the environment internal UUID (Unique Universal Identifier) as well as an object's parent and child nodes in a component tree (section 3.3.1).
2. Life cycle: A component with a life cycle has methods defining its build, run and shutdown stages during execution.
3. Module: A module implements a component with a life cycle in the simulation.
4. Plugin: A module whose life cycle is defined as docker workflow (section 2.2.1). Usually represents a docker container.
5. Trial: A collection of plugins and/or modules with a defined order of execution. A trial represents a single test executed in an experiment.
6. Experiment: A series of trials.

3.1 The Distributed Experiment Pipeline

Experiments are designed as a series of trials to be executed independently. From a dynamic perspective, the execution of an experiment in our framework has three stages: the trial generation, the trial execution and the trial analysis. A trial is then a collection of such three stages. Figure 3.8 depicts the process.

Figure 3.8 – Distributed Experiment Pipeline.



Source: Made by the author (2023).

Firstly, the user creates a Trial Class in Python and provides their own user parameters as well as a JSON configuration file. The Trial class then parses the json and instantiates each plugin in order according to its component tree and initiates the build cycle for a plugin/module. Alternatively, the user can execute plugins directly without the need for a Trial Structure (appendix C).

The Trial Execution stage starts with the Plugin Build Cycle calling the docker API with a build image command which creates a docker image based on user provided parameters and (optionally) a Dockerfile. The image serves as a template to then run a docker container with a specific user command. Then, the Run Cycle specifies the command to run inside the container and starts the execution of the trial. During the execution, the ROS Logger writes the corresponding log file and awaits for a "WARN" flag to trigger with the simulation result (see Table 4.4).

Finally, the Logger triggers the shutdown cycle which stops all running Plugins on the trial. For each docker container, it then calls the API with a stop command.

3.2 The Architecture

Our architecture is divided into three separate layers:

i Trial Layer – The Trial Layer implements classes to build and execute Trials based on user provided configurations, trials are executed in a series as part of a larger experiment. Trial execution corresponds to parsing the description provided by the user and instantiating each listed component, i.e. robots, simulators, etc.

ii Plugins Layer – The Plugins Layer implements the life cycle of a component in the simulation. Plugins represent components needed for a simulation, robots, simulators, ROS components (nav2, rviz, etc.) and so on. A plugins' life cycle relates to its building stage with user provided models, maps and configurations to its execution stage where our system must launch each module so that it's functional and testable and finally its shutdown stage where our system must recognise the end of a trial and stop all plugins associated with that trial.

iii Docker API Layer – Lastly, the docker API layer implements abstractions over the docker API using the python docker SDK. A Plugin that implements a docker container can have it's life cycle defined by the life cycle of a microservice in docker. That way, creating a plugin means creating an image, executing a plugin means running a container based on that image and shutting down a plugin means stopping the container.

With this containerised modular approach to robotics simulation we can leverage reproducibility and scalability of our environment without changing the underlying simulation software. It also allows us to provide a more scalable solution to trial execution as both trials and their subcomponents can be distributed across machines according to the user needs.

In the next sections, we present further details of each layer of our concrete architecture.

3.3 Trial Layer

A hierarchy of modules naturally emerges in a robotics simulation, navigation modules depend on robots which then depend on simulators which, in turn, depend on ROS and so on. So a tree structure is an intuitive way to model the relationship between components.

Specifically for trials, we propose that they can be better understood through a hierarchical view of its sub-components.

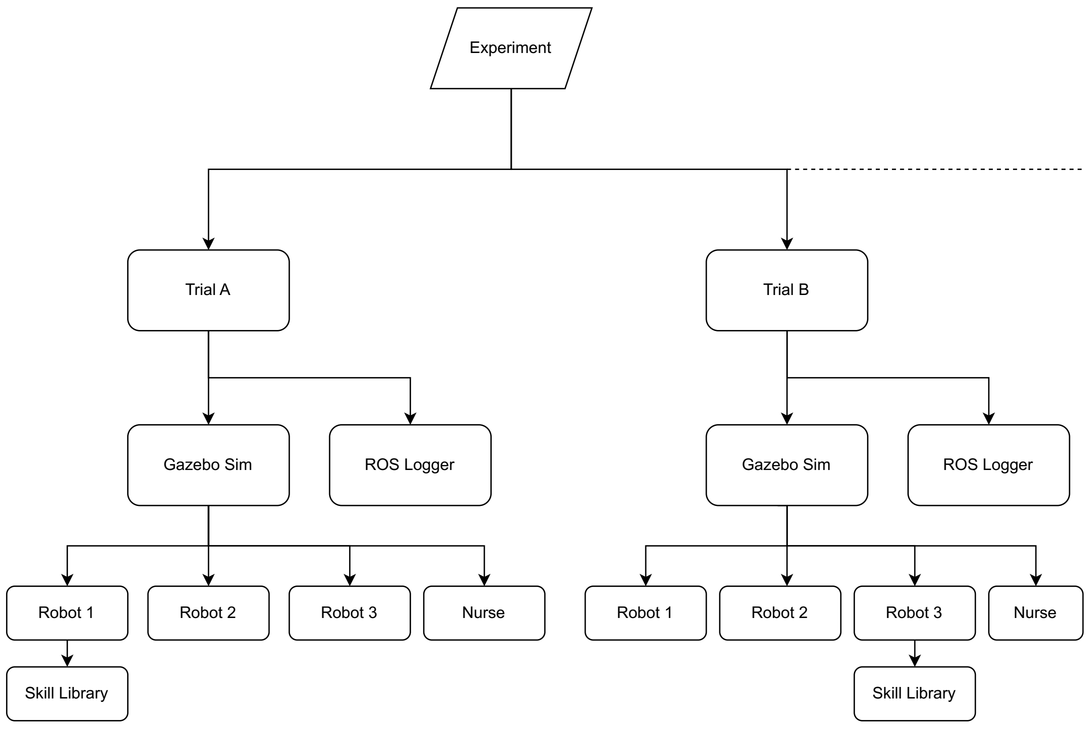
Components can have other components as build dependencies, but they may also have runtime dependencies in their life cycle. A navigation module needs another robot module in order to be built, while a robot module may then be built to work with multiple simulators through ROS. However, at runtime the robot always needs to be instantiated after the simulator, so we say that the robot has a run dependency on the simulator even though it may not have a build dependency.

With a tree of components we can model our trials and also use that hierarchical relationship in our module life cycle by propagating events through the tree (Section 3.3.2).

3.3.1 Component Tree

We model an experiment as a series of trials where each trial is represented by a component tree. Each node represents a component, module or plugin which can then have another parent or child node.

Figure 3.9 – Component tree for an experiment with 2 trials



Source: Made by the author (2023).

In Figure 3.9 we see an example of a usual HMRS (Heterogeneous Multi-Robot System) experiment. Each trial has 3 robots and a nurse which are assigned to a gazebo simulation. In each trial only one of the robots was assigned a mission thus the corresponding robot has the skill library module.

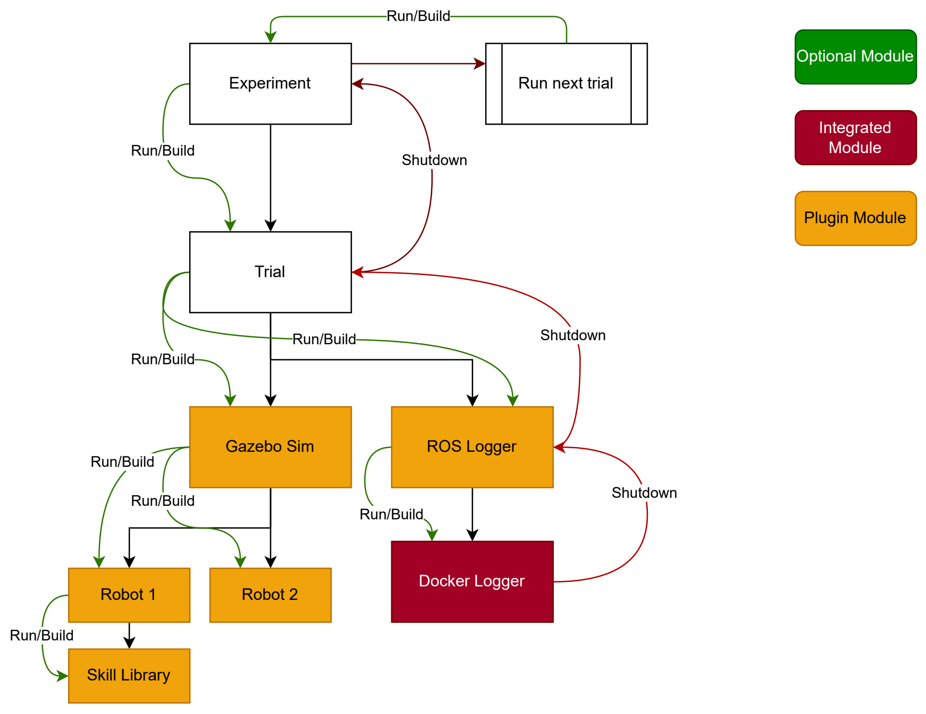
3.3.2 Event Callbacks

Each step of the life cycle of a module is performed by an event callback propagated in the component tree. Nodes receive triggers to run an event and then propagate it downwards or upwards in the tree.

A user can implement their own custom events, but we provide the following events natively:

- Run: Runs module and all of its children.
- Build: Builds module and all of its children.
- Stops: Stops module and all of its children.
- Shutdown: Signals Trial shutdown and propagates it to the node's parent.
- Restart: Signals Trial shutdown and then signals the Trial to be rerun.

Figure 3.10 – Event callbacks and module life cycles.



Source: Made by the author (2023).

Events are sent in the form of simple string messages which are parsed with a switch case. In Figure 3.10 we have the detailed life cycle of modules in an experiment. During the build and run cycles the corresponding event is propagated downwards as each module is built. While during the shutdown cycle the Docker Logger triggers the shutdown event which is propagated upwards.

When an Experiment class receives a shutdown from a trial it then triggers the run next trial method which restarts the cycle by calling the build event on a new trial in the queue (if one is available).

Using this callback system allows us to operate asynchronously on modules which is essential for a distributed simulation. That way we can have the user running multiple trials or experiments simultaneously on the same machine or on different hosts and not worry about any conflicts. This also allows the simulation to be more readily scalable.

3.4 Plugins Layer

Modules can also utilise other modules internally. We then classify those modules as either Optional or Integrated. An optional module can be chosen during the build stage while integrated modules are automatically launched alongside the main module.

For each of the named classes below we provide a standard superclasses that provides useful functionality for that type of structure, such as the initial pose (position and orientation) for a robot.

3.4.1 Simulators

The Simulator Superclass provides bindings to setup a GUI inside the container if the user passes the argument *headless* as False. It's also possible to have a GUI command and a different headless command to be run in the container for each case.

We provide 3 simulators by default:

- RMF Simulator: Launches An RMF Office Demo in Gazebo Classic
- Gazebo Simulator: Launches Gazebo Classic
- Gazebo Ignition: Launches Gazebo (Ignition)

For each simulator we provide methods to add directories with maps or models as volumes to the container as well as setting up environment variables for each simulator. It's also important to note that Ignition is built on top of OpenGL which has hard requirements for an X Server so nvidia or intel GPU docker runtime drivers are needed to execute the graphical interface.

3.4.2 Robots

The Robot Superclass sets up the following environment variables for general robot modules:

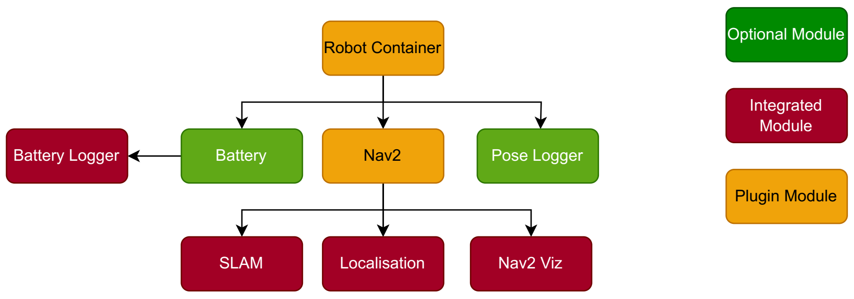
- ROBOT NAME: Name for the robot node in ROS2.
- ROBOT NAMESPACE: Namespace for the robot node.
- CONFIG: Json Config (section 4.2.1.3) for the robot.
- ROBOT POSE: Initial pose for the robot.
- ROBOT SDF: SDF model to launch in gazebo.

We provide 4 different robot modules:

- Human: Human model used for the Nurse in the simulation.
- Turtlebot3: Turtlebot3 robot without navigation.
- Turtlebot3 with Nav2: Turtlebot3 robot with nav2 navigation and simulated linear battery.
- Turtlebot4: Turtlebot4 robot based on the official model and simulator package (KREININ, 2023).

Below is a diagram representing the general structure of a mobile robot module:

Figure 3.11 – Robot module and submodules.



Source: Made by the author (2023).

3.4.3 Loggers

The basic form of the logger is built around the Docker Logger plugin which calls the docker API on a timer to access its parent's container's log info. That means the docker logger can be added onto any other plugin and log the STDOUT for the container. By default the timer will update the logs every 10 seconds.

A docker logger can also be passed a target string so it can launch an event or stop itself when the target string is present in the logs.

Built on top of the docker logger, the ROS Logger writes any messages published to a certain topic to a file, by default it will log any messages sent to the "/log" topic. In executing the simulation we use the ROS Logger to log the robot's battery, pose and communication status.

When the simulation ends a "WARN" message will be published to the "/log" topic which then triggers a shutdown event inside the component tree.

3.4.4 Miscellaneous

Finally we also provide miscellaneous plugins which offer additional functionality in ROS. Those are:

- RVIZ: Runs rviz2 in a GUI, also requires a nvidia or intel docker runtime driver as rviz2 is built on OpenGL.
- Nav2: Runs an independent nav2 instance.
- Skill Library: Models and executes robotic skills as python functions or BTs.

3.5 Docker API Layer

The docker SDK allows us to interact with the API for the docker daemon through python, it essentially lets us do anything that the *docker* command does in the terminal.

The Docker Engine API works by providing an interface between a docker client, either a local or remote host, and the docker daemon which performs the action.

Through the Plugin class we provide wrappers for most of the SDK methods and classes so the user can feasibly use our library without interacting with the SDK directly, we provide bindings for Volumes, Mounts, Environments, Networks, Containers and Images.

If needed the user can also access the internal SDK class of each object and use it manually that way.

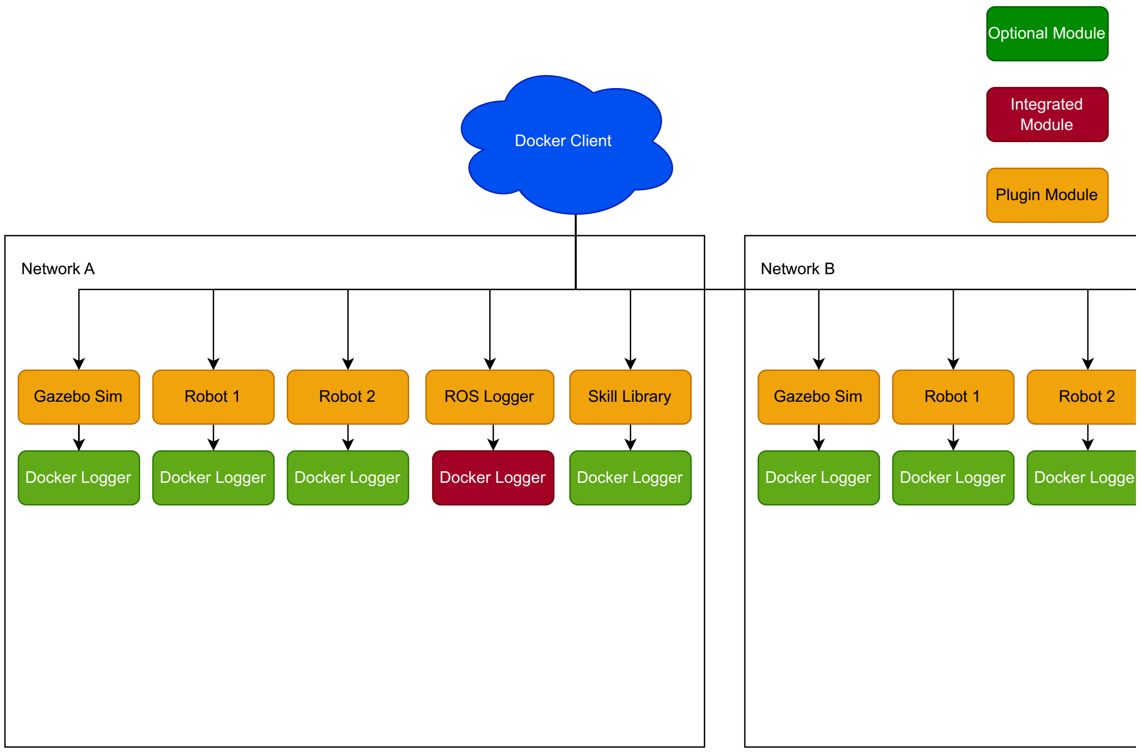
3.5.1 Runtime Priority

The Component Tree (section 3.3.1) represents our dependency graph for the build stage while the runtime queue represents the runtime execution order for a Trial.

Each Module is associated with a given priority from 0 to 5 and once added to a Trial are executed through a priority queue (lowest priority first). This gives us a clear separation between the build and run stages so modules have separate build and run dependencies.

In the Figure below (3.12) we have a diagram showing an example of a runtime queue for a multi-trial experiment.

Figure 3.12 – Docker runtime diagram



Source: Made by the author (2023).

Each trial is isolated with it's own network in the docker daemon, but all plugins share the same client. Each Plugin represents a python class wrapper around a docker container so you can still interact with it normally, i.e. check logs, execute commands, inspect it, etc., even with the *docker container* command in the terminal.

The Docker Logger submodule has higher priority than the main module so it's executed afterwards.

3.5.2 Distributed Simulations

For this project we wanted both Plugins as well as Trials and Experiments to have their own client variables, that way we can provide the user with a way to micromanage his simulation choosing specific modules to be executed in different hosts or simultaneously executing multiple Trials as a package of Plugins in different hosts.

This allows the simulation to be more scalable both in macro terms, meaning running more trials with more computer simultaneously, and micro terms by assigning each robot to a specific client thus allowing more robots and a larger simulation capacity.

We planned the distributed simulation to have a central host that delegates experiments, trials and/or plugins to other hosts through ssh. That means the central host acts as the docker client while the remote hosts' docker daemon receives the API calls through ssh.

Internally the docker SDK uses the paramiko ssh library (GAYNOR, 2023) for python and bridges all API communication through ssh. One of the main limitations of paramiko is you can't easily transfer directories like you can with scp. For our use case that means that with the docker SDK mounted volumes don't actually transfer files through ssh so volumes and mounts can only reference files already in the remote host. The user can circumvent that by transferring those files manually or with scp.

An important thing to note is that the gazebo server tends to be the largest bottleneck in the simulation, often simulating the robot's LIDAR tends to be quite computationally expensive for both CPU and GPU so the simulation can become unmanageable with many robots.

3.6 Currently Available Modules and Plugins

In order to allow the ready use of our framework, we have implemented a set of modules and plugins as presented in Table 3.1. The modules are classified according to their types.

We should also note that the code for our framework is publicly available at (MORAES, 2023a) and all the robotics skills created are available in (MORAES, 2023b). Further examples of how to setup individual modules as well as how to generate one's own trial are available in the appendices.

Table 3.1 – Available modules.

Module	Type	Dependencies	Description
Docker Logger	Logger	Network and Parent Container	Logs stdout of parent docker container
ROS Logger	Logger	Network and Parent ROS Container	Logs ros logs and ros messages of parent container
ROS 2 Network	Network	-	Instantiates ROS 2 environment and network
Human	Robot	Gazebo Simulator	Spawns human ragdoll model and optionally nurse behaviour node
Turtlebot3	Robot	Gazebo Simulator	Spawns static turtlebot3 robot model and publishes tf tree
Turtlebot3 & Nav2	Robot	Gazebo Simulator	Spawns turtlebot3 robot with nav2 navigation stack
Turtlebot4	Robot	Gazebo Ignition Simulator	Spawns turtlebot4 robot with the official simulator package (optionally can also spawn charging dock)
Nav2	Ros2 Component	Robot	Launches nav2 navigation stack for a given robot with a state publisher
Rviz2	Ros2 Component	ROS2 Network and nvidia docker drivers (needed for opengl)	Launches GUI visualizer for ros
Pytrees	Ros2 Component	Robot	Launches behavior trees for robot skillset and skill library
Skill Library	Ros2 Component	Robot	Executes robot's skills as either BTs or python functions
RMF Gazebo	Simulator	ROS2 Network	Launches RMF simulation
Gazebo	Simulator	ROS2 Network	Launches Gazebo (GUI optional)
Gazebo Ignition	Simulator	ROS2 Network and nvidia docker drivers (needed for opengl)	Launches Gazebo Ignition (GUI optional)

Source: Made by the author (2023).

4 Framework Evaluation

In this chapter, we present the evaluation of our framework and our thought process on exercising our experimentation pipeline in a distributed computing environment.

4.1 One-off Simulation Scenarios

In order to provide a bird's eye view of the feasibility of our work, we provide one-off simulation scenarios implemented in Python. Using the Pytest library (KREKEL et al., 2004) we created simulation scenarios for the build and run stages for every plugin and trial. In this section, we outline a couple of one-off simulation snippets as examples and give our thought process on their design.

For these simulations robots are instatiated without a specific objective

4.1.1 Plugins

Here is a snippet on how to setup a simple simulation with one robot inside a gazebo classic simulation.

Code 4.1 – Run a gazebo simulation with TB3

```python
import docker
from robotics_sim.plugins.simulators.gazebo import Gazebo
from robotics_sim.plugins.robots.turtlebot3 import Turtlebot3
from robotics_sim.plugins.networks.ros2_network import ROS2Network


client = docker.from_env()
network = ROS2Network(docker_client=client, name="ros2")
sim = Gazebo(docker_client=client, headless=True,
    auto_remove=True, network=network)
robot = Turtlebot3(docker_client=client, tag="robot1",
    auto_remove=True, network=network)

network.build()
sim.build()
robot.build()

sim.run()
robot.run()
```

The docker client is instantiated by the docker.from_env function which creates a client based on the user level environment variables. A user can also directly their own environment variables by setting the *ENVIRONMENT* argument when creating their client class.

Next we create the network with ROS so both robot and simulator can communicate and we name it ros2. We then create the simulator and robot plugins connected to the same network. Both plugins have the auto_remove argument set to True so the underlying docker container will be removed after it is done executing.

Then since we are not using a trial, we manually build and run the plugins. And that is a simple setup for what would have been a much more difficult and involved process with docker-compose.

Now we have a slightly more complex example where we create a simulation with two different robots with the nav2 framework:

Code 4.2 – Run gazebo simulation with 2 robots with nav2 and logging

```python
import docker
from robotics_sim.plugins.simulators.gazebo import Gazebo
from robotics_sim.plugins.robots.turtlebot3_nav2 import
    Turtlebot3withNav2
from robotics_sim.plugins.networks.ros2_network import ROS2Network
from robotics_sim.core import pose


client = docker.from_env()
network = ROS2Network(client, name="ros2")
sim = Gazebo(client, headless=False, auto_remove=True,
    network=network, path_to_world=
"/opt/ros/humble/share/turtlebot3_gazebo/
worlds/turtlebot3_world.world")
sim.add_logger(write_to_file=True, filename='sim.log')

ps = pose.Pose()
ps.position.x = -2
ps.position.y = -0.5
ps.position.z = 0.1
robot = Turtlebot3withNav2(client, robot_name="turtlebot",
    robot_namespace="turtlebot", auto_remove=True, network=network,
    initial_pose=ps, use_rviz=True)
robot.add_logger(write_to_file=True, filename='robot.log')
sim.add_model_path(container=robot,
    path="/opt/ros/humble/share/turtlebot3_gazebo")

ps = pose.Pose()
ps.position.x = 2
ps.position.y = -1
ps.position.z = 0.1
robot2 = Turtlebot3withNav2(client, robot_name="turtlebot2",
    robot_namespace="turtlebot2", container_name="turtlebot2",
    auto_remove=True, network=network, initial_pose=ps,
   use_rviz=True)
robot2.add_logger(write_to_file=True, filename='robot2.log')

network.build()
sim.build()
robot.build()
robot2.build()

sim.run()
robot.run()
robot2.run()
```

As before, we start by creating the simulator and network. But this time, we add the path inside the container to the world sdf file, we are be using the TB3 test world map for this example. One can then add a docker logger to any plugin. In this case, we will be writing the logs to the file 'sim.log'.

Next, we create a pose for the robot position as well as the first robot plugin also setting a custom name and a namespace. We also set the use_rviz argument to True so we can visualize the navigation topics in a separate rviz window each.

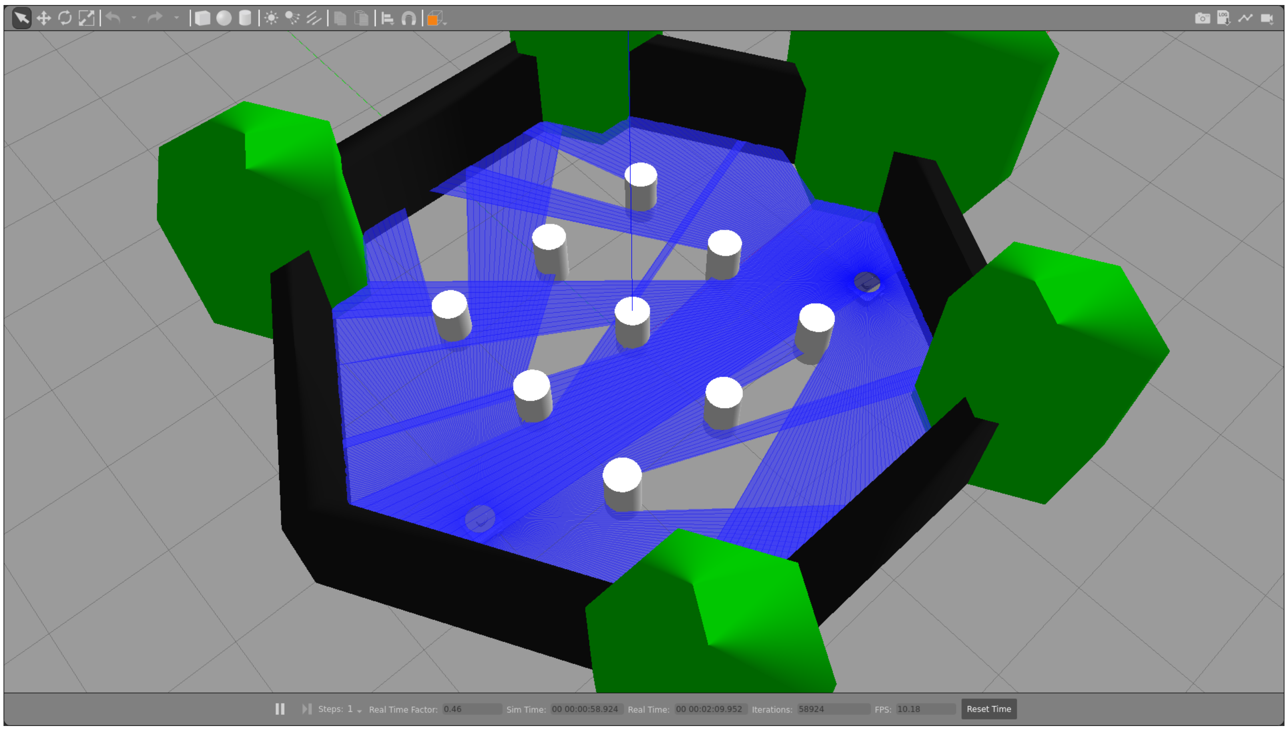
Then we add the TB3 models to the simulation so we can actually see the robots. We do this by creating a volume that is shared between the gazebo and turtlebot containers.

Finally, we create a separate robot for a multi-robot simulation.

In Figure 4.13 we show two robots in the turtlebot3 test map and in Figure 4.14 we show the *rviz* visualisation of both robots navigating simultaneously.

A video of the simulation is available in: `https://youtu.be/UFHcQ_ZXLOg`.

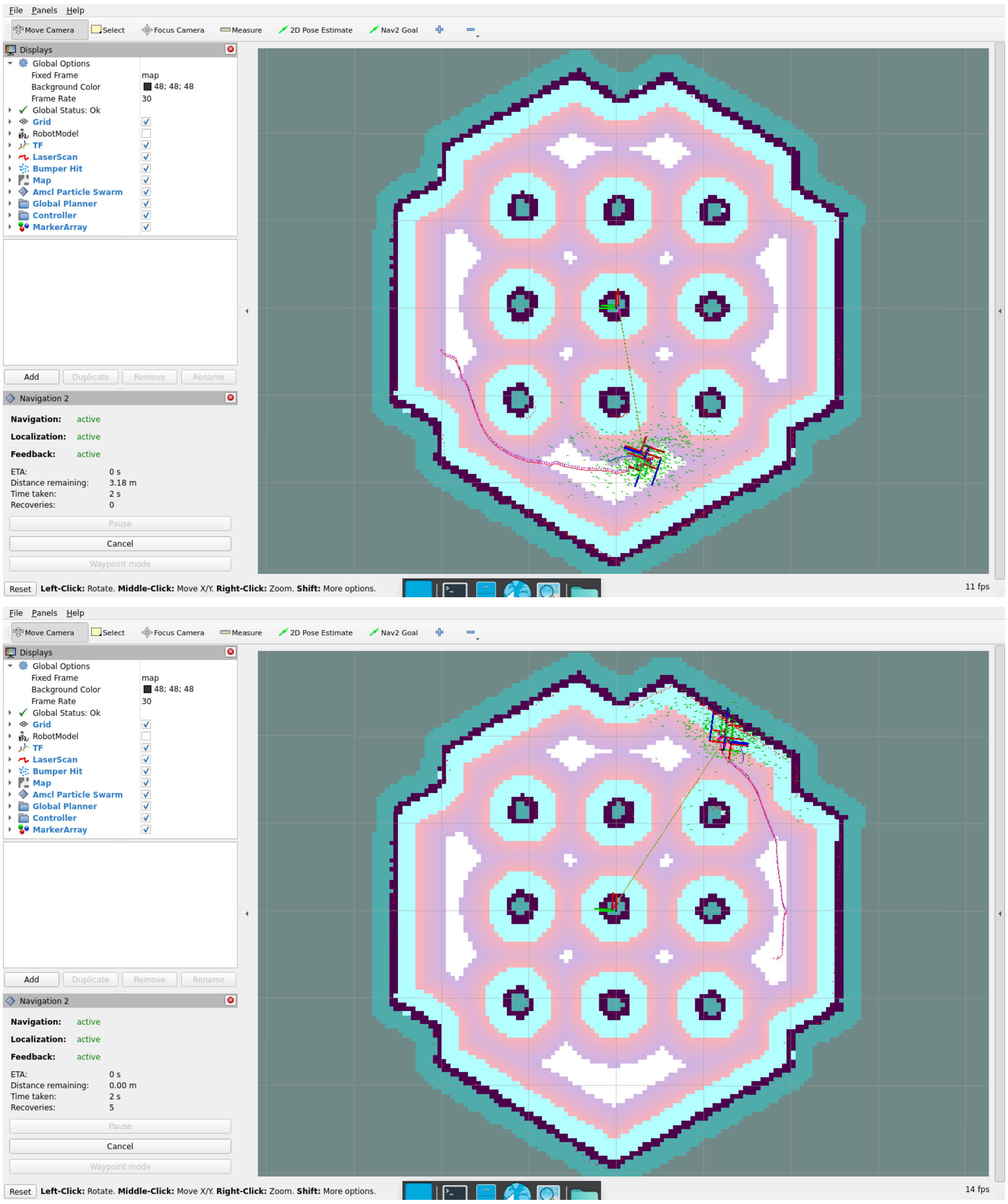
Figure 4.13 – Gazebo simulation with two TB3s.



Steps: 1 Real Time Factor: 0.46 Sim Time: 00 00:00:58.924 Real Time: 00 00:02:09.952 Iterations: 58924 FPS: 10.18

Source: Made by author.

Figure 4.14 – Robots navigating in rviz



```
Global Options
  Fixed Frame          map
  Background Color     48; 48; 48
  Frame Rate           30
Global Status: Ok
Grid                   ✓
RobotModel
TF                     ✓
LaserScan              ✓
Bumper Hit             ✓
Map                    ✓
Amcl Particle Swarm    ✓
Global Planner         ✓
Controller             ✓
MarkerArray            ✓

Navigation:    active
Localization:  active
Feedback:      active

ETA:                  0 s
Distance remaining:   3.18 m
Time taken:           2 s
Recoveries:           0
```

```
Navigation:    active
Localization:  active
Feedback:      active

ETA:                  0 s
Distance remaining:   0.00 m
Time taken:           2 s
Recoveries:           5
```

Source: Made by the author (2023)

4.1.2 Trials

Here is a more complex example where we setup a trial. We are running a trial from the Lab Samples mission (see Section 4.2).

Code 4.3 – Run Multi Robot Trial from Mission Control

```python
import docker
from robotics_sim.plugins.hmrs.load_experiment import parse_config
from robotics_sim.trials.hmrs_trial import HMRSTrial


docker_client = docker.from_env()
path = str(ProjectPath/
"tests/hmrs/old_hospital_map/experiment/trials.json")
config = parse_config(path)[0]

trial = HMRSTrial(docker_client, config, config['id'],
    config['code'], headless=False,
    path_to_world="/workdir/map/hospital.world", use_rviz=True)

trial.sim.add_mount(source=str(ProjectPath/
"tests/hmrs/old_hospital_map/param/map"), target="/workdir/map")

trial.setup_robots(param_path=str(ProjectPath/
"tests/hmrs/old_hospital_map/param"),
    map_yaml='/workdir/param/map/map.yaml', use_pose_logger=True,
    use_battery=True)
trial.setup_nurse()

trial.build()
trial.run()
```

First we create our client and parse the config json with the experiment description. The parse_config function will return a list of trial descriptions and since we are only running a single trial we will be using only the first entry.

Then we create our multi-robot trial and pass the arguments so internally it can setup the network and simulator. Next since we are using a custom map we set a mount so we can copy our local volume containing the map to the simulator container.

We then call the setup_robots function and pass the path to the parameters and map yaml files, we also set the robots to use a battery and a pose logger. Finally we setup a nurse and in just 12 lines of code we have successfully launched a trial. Seen in Figure 4.15

Lastly we show how to setup an experiment.

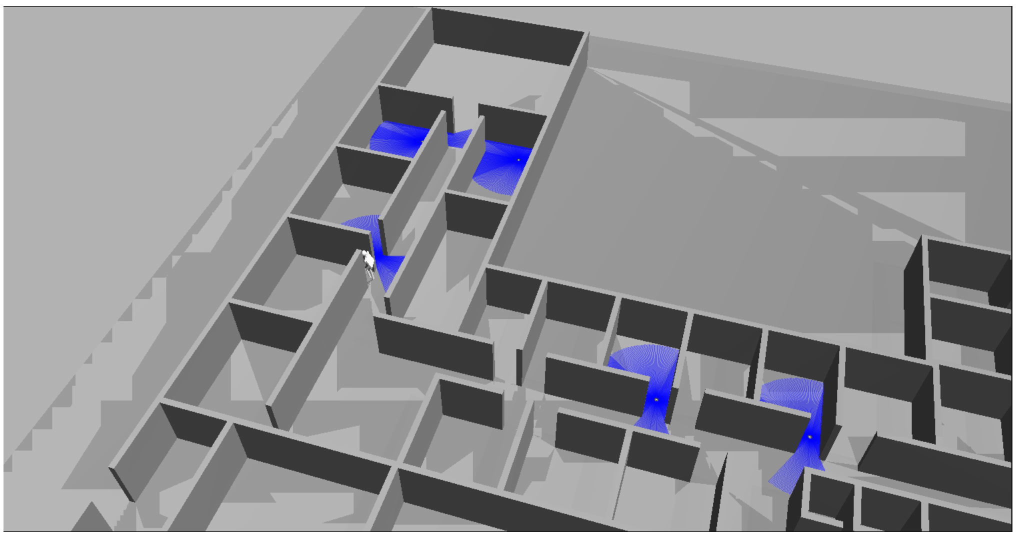
Code 4.4 – Run Multi Robot Experiment from Mission Control

```
import docker
from robotics_sim.plugins.hmrs.load_experiment import parse_config
from trials.experiment import Experiment
from core.components import ProjectPath

docker_client = docker.from_env()

path = str(ProjectPath/"tests/hmrs/old_hospital_map/
experiment/trials.json")
config = parse_config(path)

experiment = Experiment.from_config(docker_client, config=config,
    map_path=str(ProjectPath/"tests/hmrs/old_hospital_map/
param/map"),
    param_path=str(ProjectPath/"tests/hmrs/old_hospital_map/
param"), use_rviz=False,
    path_to_world="/workdir/map/hospital.world",
    dir="distributed_experiment/les-01", headless=True)

experiment.build()
experiment.run()
```

An experiment is essentially a list of trials so we can pass the config directly and let it instantiate each trial just as we did in the previous example.

We can also pass the trials_to_execute argument listing which trials to execute specifically as well as in the experiment run method we can pass the number of trials to be run simultaneously.

Figure 4.15 – Multi Robot experiment executed using our framework



Source: Made by the author (2023).

4.2 Mission Control Experiment Evaluation

For a more complex evalution of our work, we evaluate our framework in Mission-Control (RODRIGUES et al., 2022) for the Lab Sample robotic mission (ASKARPOUR; TSIGKANOS; MENGHI; CALINESCU; PELLICCIONE; GARCÍA, et al., 2021). To do so, we execute our framework in a distributed setting comprised of eight computers.

4.2.1 Experiment Setup

For the purposes of evaluating our framework in MissionControl, we will focus on the Trial Generation and the Trial Execution steps of our framework pipeline. Also, we provide logs so the user can use them with their own analysis tools, such as Microsoft Excel, Jupyter Notebook, R, etc.

4.2.1.1 Trial Generation

In this evaluation, trials perform a Lab Sample mission (ASKARPOUR; TSIGKANOS; MENGHI; CALINESCU; PELLICCIONE; GARCIA, et al., 2021) consisting of a single navigating robot in a hospital world that has to move to a room with a nurse on it. The robot then has to authenticate the nurse and receive a sample. Finally the robot has to deliver the sample to the laboratory room where a robotic arm will pick up the sample. The challenge comes in electing a robot that has the right capabilities to complete the mission among a group of 6 potential robots.

Robots are spawned in the simulation with parameters configured by specific factors described in the trial json file. A list of possible factors is available in Section 4.2.1.2.

We continued to follow the trial generation json format used in MissionControl (RODRIGUES et al., 2022), given as follows:

Table 4.2 – Trial generation json data fields.

Field	Type	Data
Trial	List	List of all trials to be executed
Code	String	String representing all factors (4.2.1.2) in the simulation
Factors	Dictionary	Dictionary relating factors to their respective codes
ID	Integer	Trial UUID
Nurses	List	List containing the room location and position of all nurses in a hospital scenario
Robots	List	List containing the robots in the simulation

Source: Made by the author (2023).

Each json file represents an experiment which is also a list of trials, each trial then has all of the other fields listed above (Table 4.2). The simulation then builds and runs every trial based on the file order. Although the user can also list specific trial ids to be executed when creating his experiment class.

You can see an example json trial in Annex B.

4.2.1.2 Factors

For this evaluation, we have 81 permutations of factors, which are each executed twice: one for the baseline random approach and one for the MissionControl coordinated approach.

Factors are coded from "a" to "c" and each one affects a different aspect in the simulation. Per (RODRIGUES et al., 2022) we have the following factors:

1. Average Speed - Speed fixed at 0.15m/s
2. Robot Battery Charge
3. Robot Battery Discharge Rate
4. Robots Location
5. Robot Skill Set (Skills)

Therefore we have 4 different modifiable factors, each with 3 possible values, totaling 3^4 or 81 permutations for possible trial generation.

Codes are given as 5-character strings, such as "acabb" meaning Average Speed "a", Battery Charge "c", Discharge Rate "a", etc. Each character is previously assigned a value related to its factor.

4.2.1.3 Robots

Robots have the following fields in the json file:

Table 4.3 – Robot json data fields.

Field	Type	Data
Average Speed	Float	Average Speed = 0.15m/s
Battery Charge	Float	Initial Battery Percentage
Battery Discharge Rate	Float	Battery Discharge rate per sim second
ID	Integer	Robot UUID
Local Plan	Dictionary	Dictionary of tasks assigned to the robot
Location	String	Room the robot starts in
Name	String	Robot name
Position	Float List	X, Y and Yaw for the robot's initial position
Skills	String List	List of skills the robot possesses

Source: Made by the author (2023).

4.2.1.4 Logs

Logs are updated every 10 seconds in the simulation clock and all messages sent to the ROS "/log" topic are written to a file named "id-code-p.log" or "id-code-b.log", for example the file "16-acabbp.log" means the trial with id 16, code "acabb" and planned approach.

By default the simulation logs the position and battery percentage of robots with a non-null local plan, i.e. robots capable of navigation. All communications performed by robots are also logged, such as the robot authenticating a nurse, opening a drawer or using an elevator.

4.2.2 Distributed Trials Execution

In the MissionControl experiment evaluation each computer in our distributed network executes 162 trials, over 8 machines which sum to 1296 trials executions. Trials executions can render the following outcomes:

Table 4.4 – Simulation end states.

State	Info
Success	All robots completed their local plan successfully
Skill Failure	Robot encountered a critical skill error or couldn't complete a navigation task in less than 600 seconds
No Skill	Robot doesn't have required skills to complete its local plan
Low Battery	Robot ran out of battery
Timeout	By default simulation times out after 15 minutes (wall time)

Source: Made by the author (2023).

Simulations can then be executed using the MissionControl runtime architecture for heterogeneous multi-robot mission coordination.

From a central computer, we delegate a different experiment instance to each host. Since our tool works asynchronously through the event callbacks we can build and run trials simultaneously on all machines. Below we provide the code used to setup the experiment:

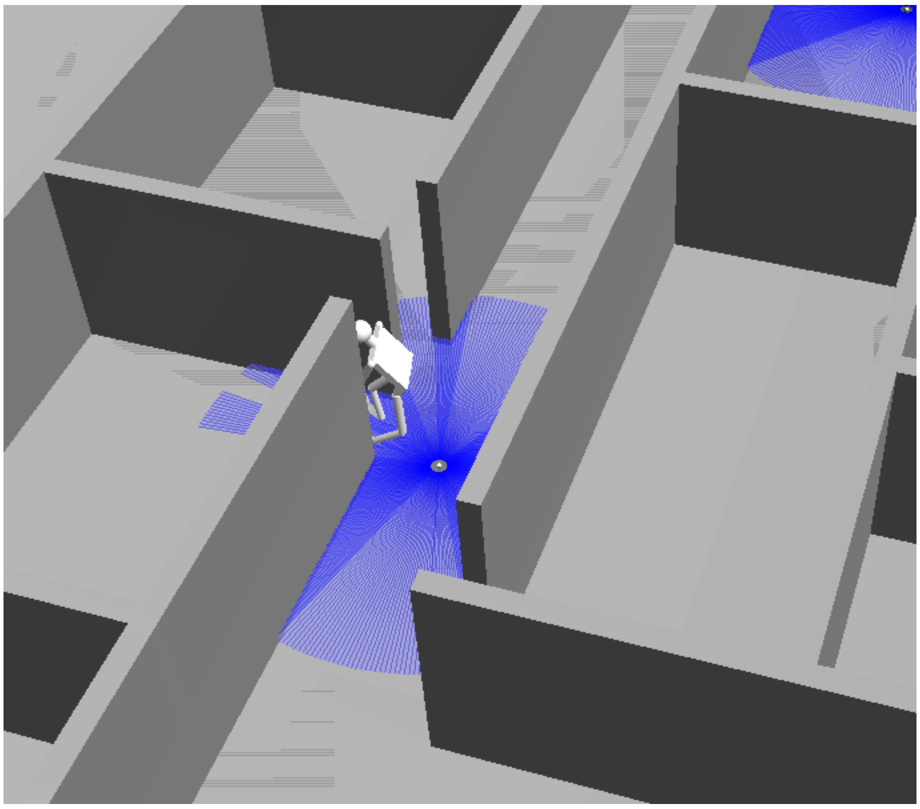
Code 4.5 – Distributed Experiment in 8 hosts

```python
import docker
from robotics_sim.plugins.hmrs.load_experiment import parse_config
from trials.experiment import Experiment
from core.components import ProjectPath


hosts = [f"les-0{i}" for i in range(1, 9)]
path = str(ProjectPath/"tests/hmrs/old_hospital_map/experiment/
trials.json")
config = parse_config(path)
experiments = []
docker_clients = []

for host in hosts:
    client = docker.DockerClient(base_url=f'ssh://lesunb@{host}')
    docker_clients.append(client)
    experiments.append(Experiment.from_config(client,
        config=config,
        map_path=str(ProjectPath/"tests/hmrs/old_hospital_map/
param/map"),
   param_path=str(ProjectPath/"tests/hmrs/old_hospital_map/
param"), use_rviz=False,
   path_to_world="/workdir/map/hospital.world",
   dir=f"distributed_experiment/{host}", headless=True,
   ssh_host=f"lesunb@{host}"))
for experiment in experiments:
    experiment.build()
for num, experiment in enumerate(experiments):
    print(f'Running Host: {hosts[num]}')
    experiment.run()
```

Before executing the trials, we have to copy the map, model and params files to each remote machine with scp since the docker sdk volumes cannot transfer files remotely. The setup is then exactly the same as the previous experiment example, but we point our docker client to use the server at a specific remote address over ssh.

For this case, since each skill is relatively simple, we will be using Python functions to model the behaviour instead of behavior trees (BTs). In Figure 4.16 we see a robot authenticating a nurse while performing its mission.

Figure 4.16 – Robot authenticating nurse



Source: Made by the author (2023).

4.2.3 Analysis

Our goal in redoing the experiment is to demonstrate that our framework is capable of performing experiments in a research context and not on corroborating the hypothesis presented in (RODRIGUES et al., 2022). So we will be comparing the results obtained in our experiment (Table 4.5) with the results obtained in the original paper (Figure 4.17).

Table 4.5 – Experiment simulation results obtained using our framework.

	Success	Low Battery	Timeout	No Skill	Skill-Failure
Planned	394	20	46	0	188
Baseline	228	22	19	120	259

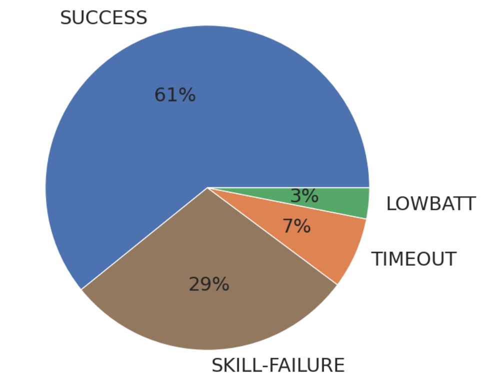
Source: Made by the author (2023).

Table 4.6 – Experiment simulation results in the original paper.

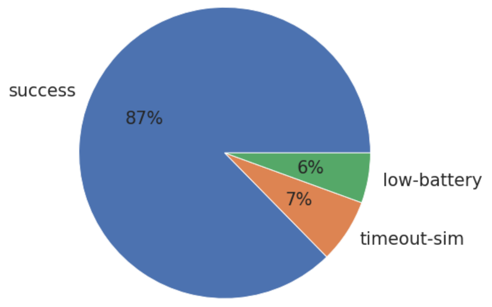
	Success	Low Battery	Timeout	No Skill	Skill-Failure
Planned	566	36	46	0	-
Baseline	354	97	22	175	-

Source: Made by the author (2023).

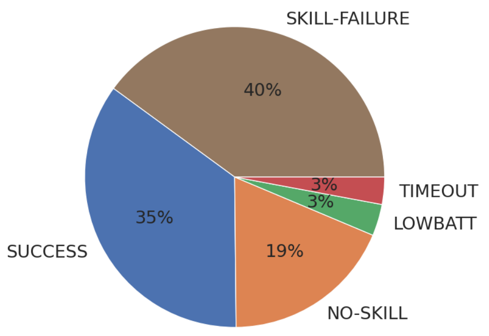
Figure 4.17 – Comparing simulation results between both experiments



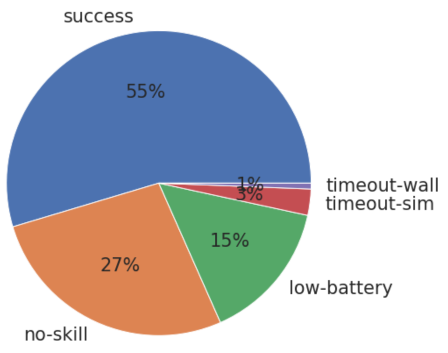
Planned Successes from the new execution



Planned Successes from previous execution

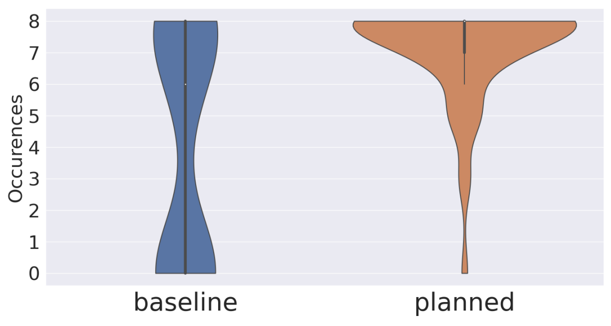


Baseline Successes from the new execution

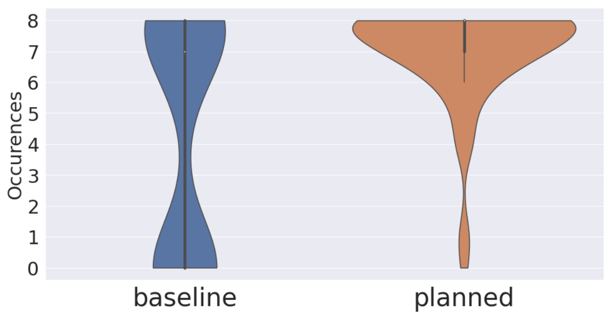


Baseline Successes from previous execution

Figure 4.18 – Violin plots for both executions



Violin plot for new execution



Violin plot for previous execution

Source: Made by the author (2023)

As expected the planned approach had more successes and had no failures due to not having the correct skill for a test.

For this scenario, the skill failure result occurs when the navigation skill fails to reach a target within 600 seconds (wall time), so either the robot fails to go to the nurse room or fails to deliver the sample within 600 seconds. Also as expected the baseline approach has more navigation failures than the MissionControl planned approach.

Overall the results were similar to the initial experiment, but navigation failures were a lot higher. This may indicate that we need to do more changes internally to the ROS simulation, such as adjusting the map scan, the waypoints passed to the navigation controller, the battery levels, etc. However, in figure 4.18 we can see the general shape of the success plot remains the same.

For a first attempt the results were promising and the ease of setting up the environment and reproducing it justifies using our framework over the initial evaluation simulation. As a comparison, in the previous evaluation (GABRIEL ARAÚJO, 2021) the simulation was launched with over 600 lines of python code and 3 separate bash files to automate startup tasks, while we had only 24 lines of code in python using our framework.

The full logs are available in `https://github.com/VicenteMoraes/robotics_sim/tree/master/logs/distributed_experiment`. You can also see a video of the experiment being executed at: `https://youtu.be/ZfpMZNr9LE8`

5 Conclusion

We implemented a framework aimed at automating tasks related to deploying and executing containers by providing a modular containerised microservices architecture to robotics simulations. Through the docker SDK we were also able to provide a simple and intuitive way to programmatically create those simulations.

Our modules were tested with automated testing with pytest where we developed 22 different tests for our project. In each test we simulate a use case for the plugin where the user instantiates a docker client and then creates a one-off simulation scenario.

Finally we test our project in a research context where we evaluate our framework in Mission-Control. Results were satisfactory, but the high amount of timeout results might indicate a problem with the navigation or interaction between gazebo and ROS inside the simulation. For future experiments a further look into the map scan for the hospital environment, or the waypoints setup might provide less timeout failures.

This work is also closely related to the problem of container orchestration, as defined by Red Hat:

> Container orchestration automates the deployment, management, scaling, and networking of containers. Enterprises that need to deploy and manage hundreds or thousands of Linux® containers and hosts can benefit from container orchestration.
>
> Container orchestration can be used in any environment where you use containers. It can help you to deploy the same application across different environments without needing to redesign it. And microservices in containers make it easier to orchestrate services, including storage, networking, and security.
>
> (REDHAT, 2022)

Modern orchestration tools such as Kubernetes or Docker Swarm are very mature projects and are apt at performing runtime orchestration tasks and container automation. In the literature there are lots of works related to the state of the art in container orchestration (CASALICCHIO; IANNUCCI, 2020) and performance comparisons between orchestration tools (JAWARNEH et al., 2019).

In relation to other orchestration projects our framework offers a more limited scope, but for our focus in multi-robot systems we offer easier integration with maps, models, robot skills planning and execution and in general ROS development.

5.1 Related Works

ExpRunA (SILVA et al., 2020) develops a DSL to execute technology oriented experiments. Using an online platform the user specifies which programs to execute and the system then runs those programs in containers according to user parameters.

In robotics, CS::Apex (BUCK; ZELL, 2019) develops flow state machines called Activity Flow Graphs to model and then execute processes in ROS for experimentation scenarios.

Both works offer similar functionality to ours, but neither has a focus on distributed experimentation and a more specific approach on providing support for gazebo and turtlebot simulations.

5.2 Future Work and Improvements

Future improvements for this work could be taken in the form of extending our BT library with more skill implementations and better BT integration and sequencing for skills. At the same time, developers can contribute with additional plugins and support for more types of trials and experiments.

Fundamentally the project would benefit from having more general and more abstracted versions of the plugins, for instance currently each robot is tied to a specific simulator, but ideally a TB3 plugin should be adaptable to be used in any simulator. More general versions of trials and experiments to be used in more multi robot situations are also interesting additions.

Lastly, further insight into the MissionControl evaluation would be useful for anyone attempting to reproduce its experiments using our framework.

References

AFZAL, A.; GOUES, C. L.; HILTON, M.; TIMPERLEY, C. S. A Study on Challenges of Testing Robotic Systems. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). 2020. P. 96–107. DOI: 10.1109/ICST46399.2020.00020. Cit. on p. 11.

AFZAL, A.; KATZ, D. S.; GOUES, C. L.; TIMPERLEY, C. S. **A Study on the Challenges of Using Robotics Simulators for Testing**. 2020. arXiv: 2004.07368 [cs.RO]. Cit. on pp. 11, 60.

API Driven DeVOps Spotlight on Docker. 2016. https://nordicapis.com/api-driven-devops-spotlight-on-docker/. Accessed: 2023-06-19. Cit. on pp. 15, 16.

ASKARPOUR, M.; TSIGKANOS, C.; MENGHI, C.; CALINESCU, R.; PELLICCIONE, P.; GARCIA, S.; J. VON OERTZEN, T.; WIMMER, M.; BERARDINELLI, L.; ROSSI, M.; BERSANI, M. M.; RODRIGUES, G. S. RoboMAX: Robotic Mission Adaptation eXemplars. In: SEAMS. **Proceedings of the 16th International Symposium on Software Engineering for Adaptive and Self-Managing Systems**. Madrid, Spain: ACM, May 2021. Cit. on p. 45.

ASKARPOUR, M.; TSIGKANOS, C.; MENGHI, C.; CALINESCU, R.; PELLICCIONE, P.; GARCÍA, S.; CALDAS, R.; OERTZEN, T. J. von; WIMMER, M.; BERARDINELLI, L.; ROSSI, M.; BERSANI, M. M.; RODRIGUES, G. S. RoboMAX: Robotic Mission Adaptation eXemplars. In: 2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). 2021. P. 245–251. DOI: 10.1109/SEAMS51251.2021.00040. Cit. on p. 45.

BEHAVIORTREE.CPP Documentation. 2020. https://www.behaviortree.dev/. Accessed: 2023-06-19. Cit. on p. 17.

BRADSKI, G. The OpenCV Library. **Dr. Dobb's Journal of Software Tools**, 2000. Cit. on p. 23.

BUCK, S.; ZELL, A. CS::APEX: A Framework for Algorithm Prototyping and Experimentation with Robotic Systems. eng. **Journal of intelligent robotic systems**, Springer Nature B.V, Dordrecht, v. 94, n. 2, p. 371–387, 2019. ISSN 0921-0296. Cit. on p. 53.

CASALICCHIO, E.; IANNUCCI, S. The state-of-the-art in container technologies: Application, orchestration and security. eng. **Concurrency and computation**, Wiley Subscription Services, Inc, Hoboken, v. 32, n. 17, n/a, 2020. ISSN 1532-0626. Cit. on p. 52.

COLLEDANCHISE, M.; ÖGREN, P. How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees. **IEEE Transactions on Robotics**, v. 33, n. 2, p. 372–389, Apr. 2017. ISSN 1552-3098. DOI: 10.1109/TRO.2016.2633567. Cit. on pp. 18, 20.

COLLEDANCHISE, M.; ÖGREN, P. **Behavior Trees in Robotics and AI**. CRC Press, July 2018. DOI: 10.1201/9780429489105. Available from: <https://doi.org/10.1201%2F9780429489105>. Cit. on pp. 17–20.

DOCKER. **dockerpy**. 12 July 2023. Available from: <https://docker-py.readthedocs.io/en/stable/index.html>. Cit. on p. 15.

ECHEVERRIA, G.; LASSABE, N.; DEGROOTE, A.; LEMAIGNAN, S. Modular open robots simulation engine: MORSE. In: 2011 IEEE International Conference on Robotics and Automation. 2011. P. 46–51. DOI: 10.1109/ICRA.2011.5980252. Cit. on p. 60.

FUNDACIO EURECAT. **Groot**. 20 June 2023. Available from: <https://github.com/BehaviorTree/Groot>. Cit. on p. 20.

GABRIEL ARAÚJO, V. M. **Morse Simulation**. 26 Aug. 2021. Available from: <https://github.com/Gastd/morse_simulation>. Cit. on pp. 11, 51.

GAYNOR, A. **Paramiko**. 25 June 2023. Available from: <https://www.paramiko.org/>. Cit. on p. 36.

GUILLAUME LOURS, D. T. **Docker-Compose V2**. 21 June 2023. Available from: <https://docs.docker.com/compose/>. Cit. on pp. 11, 12, 16.

GUIMARÃES, R. L.; OLIVEIRA, A. S. de; FABRO, J. A.; BECKER, T.; BRENNER, V. A. ROS Navigation: Concepts and Tutorial. In: **Robot Operating System (ROS): The Complete Reference (Volume 1)**. Ed. by Anis Koubaa. Cham: Springer International Publishing, 2016. P. 121–160. ISBN 978-3-319-26054-9. DOI: 10.1007/978-3-319-26054-9_6. Available from: <https://doi.org/10.1007/978-3-319-26054-9_6>. Cit. on p. 14.

ISLA, D. Handling Complexity in the Halo 2 AI. In: GAME Developers Conference. 2005. Cit. on p. 17.

JAWARNEH, I. M. A.; BELLAVISTA, P.; BOSI, F.; FOSCHINI, L.; MARTUSCELLI, G.; MONTANARI, R.; PALOPOLI, A. Container Orchestration Engines: A Thorough Functional and Performance Comparison. In: ICC 2019 - 2019 IEEE International Conference on Communications (ICC). 2019. P. 1–6. DOI: 10.1109/ICC.2019.8762053. Cit. on p. 52.

KENJI BRAMELD. **ROS Perception Vision**$_{O}$*pencv*. 20 June 2023. Available from: <https://github.com/ros-perception/vision_opencv>. Cit. on p. 23.

KOENIG, N.; HOWARD, A. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In: 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566). 2004. v. 3, 2149–2154 vol.3. DOI: 10.1109/IROS.2004.1389727. Cit. on pp. 24, 60.

KREININ, R. **Turtlebot4 Simulator**. 25 June 2023. Available from: <https://github.com/turtlebot/turtlebot4_simulator>. Cit. on p. 33.

KREKEL, H.; OLIVEIRA, B.; PFANNSCHMIDT, R.; BRUYNOOGHE, F.; LAUGHER, B.; BRUHIN, F. **pytest 7.4**. 2004. Available from: <https://github.com/pytest-dev/pytest>. Cit. on p. 38.

LOU, H. AI in Video Games: Toward a More Intelligent Game. **Science in the News, Harvard**, v. 15, 2017. Cit. on p. 18.

LUXONIS. **DepthAI**. 20 June 2023. Available from: <https://docs.luxonis.com/en/latest/>. Cit. on p. 23.

MACENSKI, S.; MARTÍN, F.; WHITE, R.; GINÉS CLAVERO, J. The Marathon 2: A Navigation System. In: 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). 2020. Available from: <https://github.com/ros-planning/navigation2>. Cit. on p. 20.

MACENSKI, S.; FOOTE, T.; GERKEY, B.; LALANCETTE, C.; WOODALL, W. Robot Operating System 2: Design, architecture, and uses in the wild. **Science Robotics**, v. 7, n. 66, eabm6074, 2022. DOI: 10.1126/scirobotics.abm6074. eprint: https://www.science.org/doi/pdf/10.1126/scirobotics.abm6074. Available from: <https://www.science.org/doi/abs/10.1126/scirobotics.abm6074>. Cit. on p. 14.

MALAVOLTA, I.; LEWIS, G. A.; SCHMERL, B.; LAGO, P.; GARLAN, D. Mining guidelines for architecting robotics software. **Journal of Systems and Software**, v. 178, p. 110969, 2021. ISSN 0164-1212. DOI: https://doi.org/10.1016/j.jss.2021.110969. Available from: <https://www.sciencedirect.com/science/article/pii/S0164121221000662>. Cit. on p. 61.

MARUYAMA, Y.; KATO, S.; AZUMI, T. Exploring the performance of ROS2. In: 2016 International Conference on Embedded Software (EMSOFT). 2016. P. 1–10. DOI: 10.1145/2968478.2968502. Cit. on p. 14.

MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. **Linux journal**, v. 2014, n. 239, p. 2, 2014. Cit. on p. 14.

MILLINGTON, I.; FUNGE, J. **Artificial Intelligence for Games**. CRC Press, 2009. ISBN 9780080885032. Available from: <https://books.google.com.br/books?id=4CLOBgAAQBAJ>. Cit. on p. 19.

MORAES, V. **Robotics Sim**. 25 June 2023a. Available from: <https://github.com/VicenteMoraes/robotics_sim>. Cit. on p. 36.

MORAES, V. **Skill Library**. 25 June 2023b. Available from: <https://github.com/VicenteMoraes/skill_library>. Cit. on p. 36.

ÖGREN, P.; SPRAGUE, C. I. Behavior Trees in Robot Control Systems. **Annual Review of Control, Robotics, and Autonomous Systems**, Annual Reviews, v. 5, n. 1, p. 81–107, May 2022. DOI: 10.1146/annurev-control-042920-095314. Available from: <https://doi.org/10.1146%2Fannurev-control-042920-095314>. Cit. on p. 19.

OSRF. **Robot Middleware Framework**. 20 June 2023. Available from: <https://osrf.github.io/ros2multirobotbook/>. Cit. on p. 25.

QUIGLEY, M.; CONLEY, K.; GERKEY, B.; FAUST, J.; FOOTE, T.; LEIBS, J.; WHEELER, R.; NG, A. ROS: an open-source Robot Operating System. In: v. 3. Cit. on p. 13.

REDHAT. **What is container orchestration?** 2022. https://www.redhat.com/en/topics/containers/what-is-container-orchestration. Accessed: 2023-06-20. Cit. on p. 52.

ROBOTIS, CLEARPATH ROBOTICS. **Turtlebot3**. 20 June 2023. Available from: <https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/>. Cit. on p. 24.

RODRIGUES, G.; CALDAS, R.; ARAUJO, G.; DE MORAES, V.; RODRIGUES, G.; PELLICCIONE, P. An architecture for mission coordination of heterogeneous robots. **Journal of Systems and Software**, v. 191, p. 111363, 2022. ISSN 0164-1212. DOI: https://doi.org/10.1016/j.jss.2022.111363. Available from: <https://www.sciencedirect.com/science/article/pii/S0164121222000929>. Cit. on pp. 11, 12, 45, 46, 49, 61.

ROS-INDUSTRIAL. **ROS2 Control Library**. 20 June 2023. Available from: <https://github.com/ros-controls/ros2_control>. Cit. on p. 14.

SILVA, E.; LEITE, A.; ALVES, V.; APEL, S. ExpRunA : a domain-specific approach for technology-oriented experiments. eng. **Software and systems modeling**, Springer Berlin Heidelberg, Berlin/Heidelberg, v. 19, n. 2, p. 493–526, 2020. ISSN 1619-1366. Cit. on p. 53.

SPLINTERED-REALITY. **Py Trees**. 20 June 2023. Available from: <https://github.com/splintered-reality/py_trees>. Cit. on p. 20.

STEVE MACENSKI. **Nav2**. 20 June 2023. Available from: <https://github.com/ros-planning/navigation2>. Cit. on pp. 21, 23.

UNDERSTANDING ROS2 Nodes. 2020. https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html. Accessed: 2023-06-19. Cit. on p. 13.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M.; REGNELL, B.; WESSLÉN, A. **Experimentation in Software Engineering**. Springer Berlin Heidelberg, 2012. (Computer Science). ISBN 9783642290442. Cit. on p. 60.

Appendix

Appendix A – Resumo Estendido em Língua Portuguesa

Experimentação é uma parte inerente de qualquer prática de ciência e engenharia. Através de experimentos controlados, pesquisadores podem obter dados para análise e verificação de suas hipóteses. Na Engenharia de Software, a experimentação surge como uma forma de validar o funcionamento de aplicações, assim como de se observar possíveis melhorias para uma aplicação (WOHLIN et al., 2012). Wohlin et al. (WOHLIN et al., 2012) definem experimentação em Engenharia de Software da seguinte forma: *Experiment (or controlled experiment) in software engineering is an empirical enquiry that manipulates one factor or variable of the studied setting.* [...] *In Human-oriented experiments, humans apply different treatments to objects, while in technology-oriented experiments, different technical treatments are applied to different objects.*

Para aplicações em robótica, técnicas de experimentação em engenharia de software surgem para suprir a necessidade de sistemas robustos e design de robôs com capacidades específicas. Mas como realizar experimentos em robótica? O projeto de robôs pode ser bastante caro do ponto de vista computacional, onde erros críticos podem ser fatais para o sistema.

Dessa forma, simuladores em robótica como o Gazebo e Morse (KOENIG; HOWARD, 2004; ECHEVERRIA et al., 2011) são essenciais no desenvolvimento de experimentos, a fim de eliminar falhas, principalmente aquelas de natureza crítica, e minimizar custos em uma aplicação. Entretanto, simuladores robóticos demandam bastante poder computacional para execução de experimentos em larga escala e não provêem meios de garantir reprodutibilidade em experimentos científicos.

Afzal et al. (AFZAL; KATZ, et al., 2020) realizaram uma pesquisa de opinião com robotistas sobre os principais desafios do uso de simuladores de robótica para testes. Os desenvolvedores alegam que é muito trabalhoso criar simulação e ambiente de execução, e há um custo computacional muito grande necessário para simulação de ambientes complexos.

Diante desses desafios, este projeto de pesquisa tem como objetivo: (i) construir um arcabouço para execução distribuída de simulações multi-robô e (ii) viabilizar a realização de experimentos científicos reprodutíveis de simulações multi-robô.

Experimentos em Engenharia de Software podem ser complexos e consistem de várias etapas que devem ser bem definidas. Segundo Wohlin et al. (WOHLIN et al., 2012), essas etapas são: (i) Definição de objetivos (*Goal Definition*); (ii) Design do experimento; (iii) Planejamento, execução e validação do experimento; (iv) Análise do experimento; e (v)

Apresentação dos resultados (*Packaging*).

O desenvolvimento de experimentos de software em robótica seguem as mesmas diretrizes, mas com um grande empecilho na simulação de modelos. É necessário criar mapas, fazer modelagem de robôs, ajuste de parâmetros, programação de controladores, etc., e ao final, o custo computacional para simular o processo pode impossibilitar a execução do experimento. No geral, simuladores multi-robô não são escaláveis o suficiente para experimentos em larga escala.

Ambientes distribuídos se mostram como uma forma de obter escalabilidade em experimentos sem a necessidade de modificar o software de simulação subjacente. Além disso, softwares desenvolvidos para robótica não são tão maduros como em outras áreas de engenharia de software (MALAVOLTA et al., 2021), e é comum que um projeto se torne inoperante após algum tempo devido à deprecação de suas dependências. A conteinerização garante a reprodutibilidade de um projeto mesmo que suas dependências estejam abandonadas/deprecadas.

Ambientes contêinerizados distribuídos já foram desenvolvidos para projetos de pesquisa em robótica no LES-UnB, o MissionControl (RODRIGUES et al., 2022), uma arquitetura desenvolvida para coordenação de missões envolvendo frotas de robôs heterogêneos, foi avaliado através de experimento distribuído com 8 máquinas do laboratório realizando 1296 cenários de execução.

Com esse projeto esperamos contribuir um ambiente de automação para execução de experimentos distribuídos tal que robotistas sejam capazes de facilmente usar princípios de Engenharia de Software experimental para criar experimentos e validar seus algoritmos e projetos.

Appendix B – Experiment Sample Json

Code B.1 – "Sample trial generation json"

```json
{
        "code": "acaccp",
        "factors": {
            "avg_speed": "a",
            "battery_charge": "c",
            "battery_discharge_rate": "a",
            "location": "c",
            "skills": "c"
        },
        "id": 63,
        "nurses": [
            {
                "location": "PC Room 3",
                "position": [
                    -28.5,
                    18.0,
                    -1.57
                ]
            }
        ],
        "robots": [
            {
                "avg_speed": 0.15,
                "battery_charge": 0.3020349930380309,
                "battery_discharge_rate": 0.00048000000000000007,
                "id": 1,
                "local_plan": null,
                "location": "IC Room 4",
                "name": "r1",
                "position": [
                    -39.44,
                    33.95,
                    0.0
                ],
                "skills": [
                    "approach_person",
                    "approach_robot",
                    "authenticate_person",
                    "navigation",
                    "operate_drawer"
                ]
            },
            {
                "avg_speed": 0.15,
                "battery_charge": 0.5779176354739738,
                "battery_discharge_rate": 0.00054,
                "id": 2,
                "local_plan": null,
                "location": "PC Room 5",
                "name": "r2",
                "position": [
                    -21.0,
                    18.0,
                    -1.57
                ],
                "skills": [
                    "approach_person",
                    "approach_robot",
                    "authenticate_person",
                    "navigation",
                    "operate_drawer"
                ]
            },
            {
                "avg_speed": 0.15,
                "battery_charge": 0.7232396784987936,
                "battery_discharge_rate": 0.00036,
                "id": 3,
                "local_plan": null,
                "location": "IC Room 2",
                "name": "r3",
                "position": [
                    -38.0,
                    21.5,
                    0.0
                ],
                "skills": [
                    "approach_person",
                    "approach_robot",
                    "authenticate_person",
                    "navigation",
                    "operate_drawer"
                ]
            },
            {
                "avg_speed": 0.15,
                "battery_charge": 0.4863040833267539,
                "battery_discharge_rate": 0.0006000000000000001,
                "id": 4,
                "local_plan": null,
                "location": "PC Room 7",
                "name": "r4",
                "position": [
                    -13.5,
                    18.0,
                    -1.57
                ],
                "skills": [
                    "approach_person",
                    "approach_robot",
                    "authenticate_person",
                    "navigation",
                    "operate_drawer"
                ]
            },
            {
                "avg_speed": 0.15,
                "battery_charge": 0.7258412726774711,
                "battery_discharge_rate": 0.0002,
                "id": 5,
                "local_plan": null,
                "location": "IC Room 6",
                "name": "r5",
                "position": [
                    -33.9,
                    18.93,
                    3.14
                ],
                "skills": [
                    "approach_person",
                    "approach_robot",
                    "authenticate_person",
                    "navigation",
                    "operate_drawer"
                ]
            },
            {
                "avg_speed": 0.15,
                "battery_charge": 0.9109794326728246,
                "battery_discharge_rate": 0.00072,
                "id": 6,
                "local_plan": [
                    [
                        "navigation",
                        [
                            "PC Room 3",
                            [
                                [
                                    -27.23,
                                    18.0,
                                    -1.57
                                ],
                                [
                                    -27.23,
                                    16.0
                                ],
                                [
                                    -28.5,
                                    16.0
                                ],
                                [
                                    -28.5,
                                    18.0,
                                    -1.57
                                ]
                            ]
                        ],
                        "navto_room"
                    ],
                    [
                        "approach_person",
                        [
                            "nurse"
                        ],
                        "approach_nurse"
                    ],
                    [
                        "authenticate_person",
                        [
                            "nurse"
                        ],
                        "authenticate_nurse"
                    ],
                    [
                        "operate_drawer",
                        [
                            "open"
                        ],
                        "open_drawer_for_nurse"
                    ],
                    [
                        "send_message",
                        [
                            "nurse"
                        ],
                        "notify_nurse_of_open_drawer_for_nurse_completed"
                    ],
                    [
                        "wait_message",
                        [
                            "nurse"
                        ],
                        "wait_nurse_to_complete_deposit"
                    ],
                    [
                        "operate_drawer",
                        [
                            "close"
                        ],
                        "close_drawer_nurse"
                    ],
                    [
                        "navigation",
                        [
                            "Laboratory",
                            [
                                [
                                    -28.5,
                                    18.0,
                                    -1.57
                                ],
                                [
                                    -28.5,
                                    16.0
                                ],
                                [
                                    -26.0,
                                    16.0
                                ],
                                [
                                    -26.0,
                                    13.0,
                                    1.57
                                ]
                            ]
                        ],
                        "navto_lab"
                    ],
                    [
                        "approach_robot",
                        [
                            "lab_arm"
                        ],
                        "approach_arm"
                    ],
                    [
                        "operate_drawer",
                        [
                            "open"
                        ],
                        "open_drawer_lab"
                    ],
                    [
                        "send_message",
                        [
                            "lab_arm"
                        ],
                        "notify_lab_arm_of_open_drawer_lab_completed"
                    ],
                    [
                        "wait_message",
                        [
                            "lab_arm"
                        ],
                        "wait_lab_arm_to_complete_pick_up_sample"
                    ],
                    [
                        "operate_drawer",
                        [
                            "close"
                        ],
                        "close_drawer_lab"
                    ]
                ],
                "location": "PC Room 4",
                "name": "r6",
                "position": [
                    -27.23,
                    18.0,
                    -1.57
                ],
                "skills": [
                    "approach_person",
                    "approach_robot",
                    "authenticate_person",
                    "navigation",
                    "operate_drawer"
                ]
            }
        ]
    }
```

Appendix C – Simulation Execution examples

Code C.1 – "Run a gazebo simulation with TB3"

```python
import docker
from robotics_sim.plugins.simulators.gazebo import Gazebo
from robotics_sim.plugins.robots.turtlebot3 import Turtlebot3
from robotics_sim.plugins.networks.ros2_network import ROS2Network


client = docker.from_env()
network = ROS2Network(docker_client=client, name="ros2")
sim = Gazebo(docker_client=client, headless=True,
    auto_remove=True, network=network)
robot = Turtlebot3(docker_client=client, tag="robot1",
    auto_remove=True, network=network)

network.build()
sim.build()
robot.build()

sim.run()
robot.run()
```

Code C.2 – "Run an RMF Gazebo Demo"

```python
import docker
from robotics_sim.plugins.simulators.rmf_gazebo_simulator import
    RMFGazebo

client = docker.from_env()
sim = RMFGazebo(client, headless=True)
sim.build()
sim.run()
```

Code C.3 – "Run TB4 simulation on a separate host with ssh."

```python

import docker
from robotics_sim.plugins.simulators.gazebo_ignition import
    GazeboIgnition
from robotics_sim.plugins.robots.turtlebot4 import Turtlebot4
from robotics_sim.plugins.networks.ros2_network import ROS2Network


client = docker.DockerClient(base_url="ssh://username@your_host")
```

```
network = ROS2Network(client, name="ros2")
sim = GazeboIgnition(docker_client=client, headless=False,
   network=network)
sim.add_logger(write_to_file=True, filename="ignition.log")

robot = Turtlebot4(docker_client=client, tag="tb4",
   auto_remove=True, network=network, use_rviz=True,
use_nav2=False, use_slam=False, use_localization=False)
robot.add_logger(write_to_file=True, filename="tb4.log")

network.build()
sim.build()
robot.build()

sim.run()
robot.run()
```

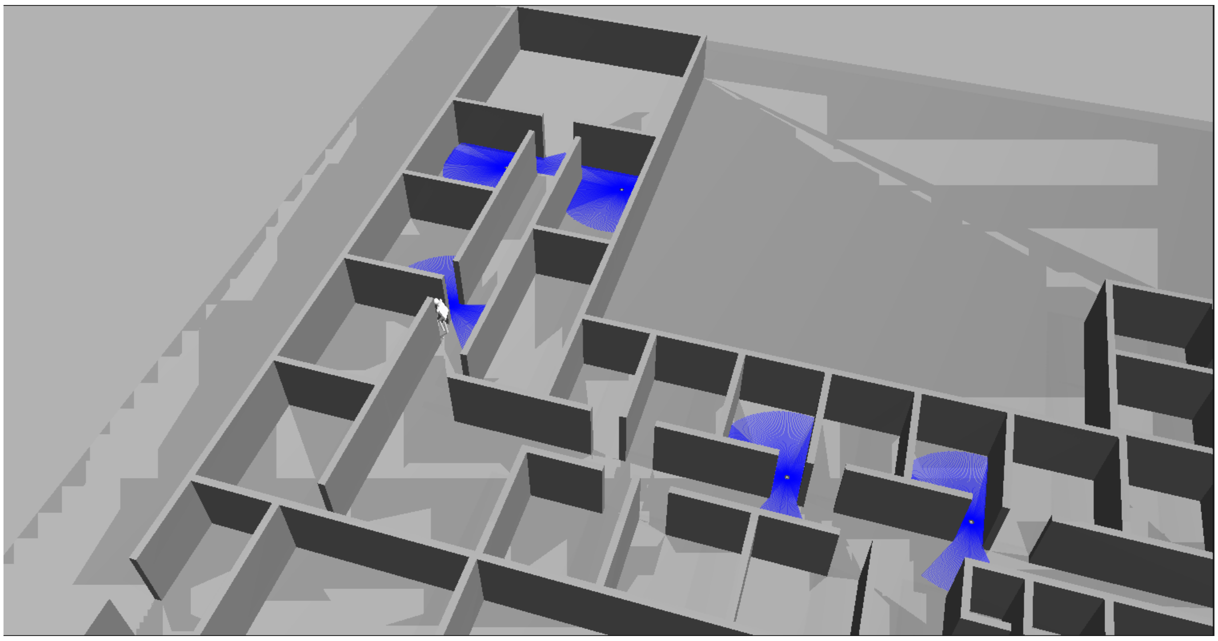
Code C.4 – "Run gazebo simulation with nav2 and logging"

```
import docker
from robotics_sim.plugins.simulators.gazebo import Gazebo
from robotics_sim.plugins.robots.turtlebot3_nav2 import
   Turtlebot3withNav2
from robotics_sim.plugins.networks.ros2_network import ROS2Network
from robotics_sim.core import pose


client = docker.from_env()
network = ROS2Network(client, name="ros2")
sim = Gazebo(client, headless=False, auto_remove=True,
   network=network, path_to_world=
"/opt/ros/humble/share/turtlebot3_gazebo/
worlds/turtlebot3_world.world")
sim.add_logger(write_to_file=True, filename='sim.log')

ps = pose.Pose()
ps.position.x = -2
ps.position.y = -0.5
ps.position.z = 0.1
robot = Turtlebot3withNav2(client, robot_name="turtlebot",
   robot_namespace="turtlebot", auto_remove=True, network=network,
   initial_pose=ps, use_rviz=True)
robot.add_logger(write_to_file=True, filename='robot.log')
sim.add_model_path(container=robot,
   path="/opt/ros/humble/share/turtlebot3_gazebo")

ps = pose.Pose()
ps.position.x = 2
ps.position.y = -1
ps.position.z = 0.1
robot2 = Turtlebot3withNav2(client, robot_name="turtlebot2",
   robot_namespace="turtlebot2", container_name="turtlebot2",
   auto_remove=True, network=network, initial_pose=ps,
   use_rviz=True)
robot2.add_logger(write_to_file=True, filename='robot2.log')

network.build()
sim.build()
robot.build()
robot2.build()

sim.run()
robot.run()
robot2.run()
```

Annex

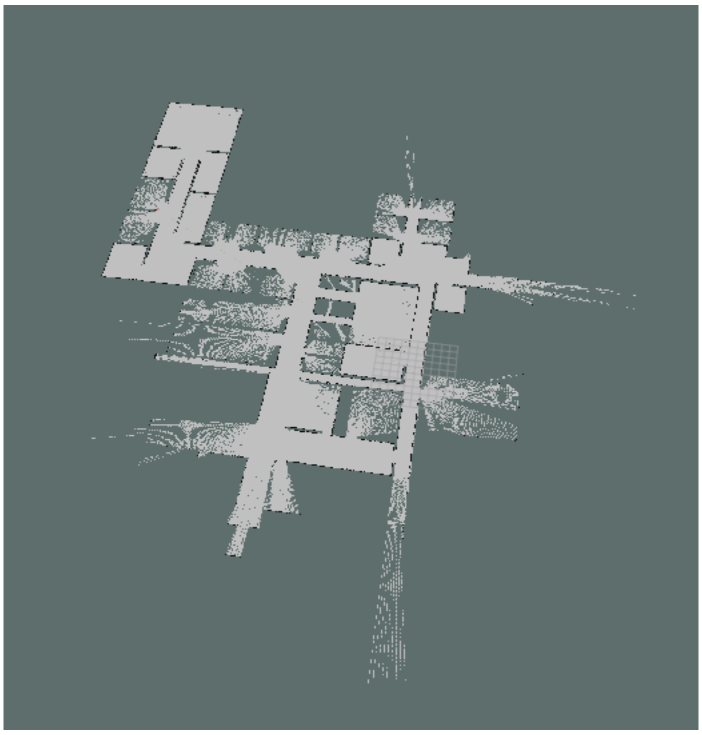
Annex A – Simulation Images

Figure A.19 – Running Simulation



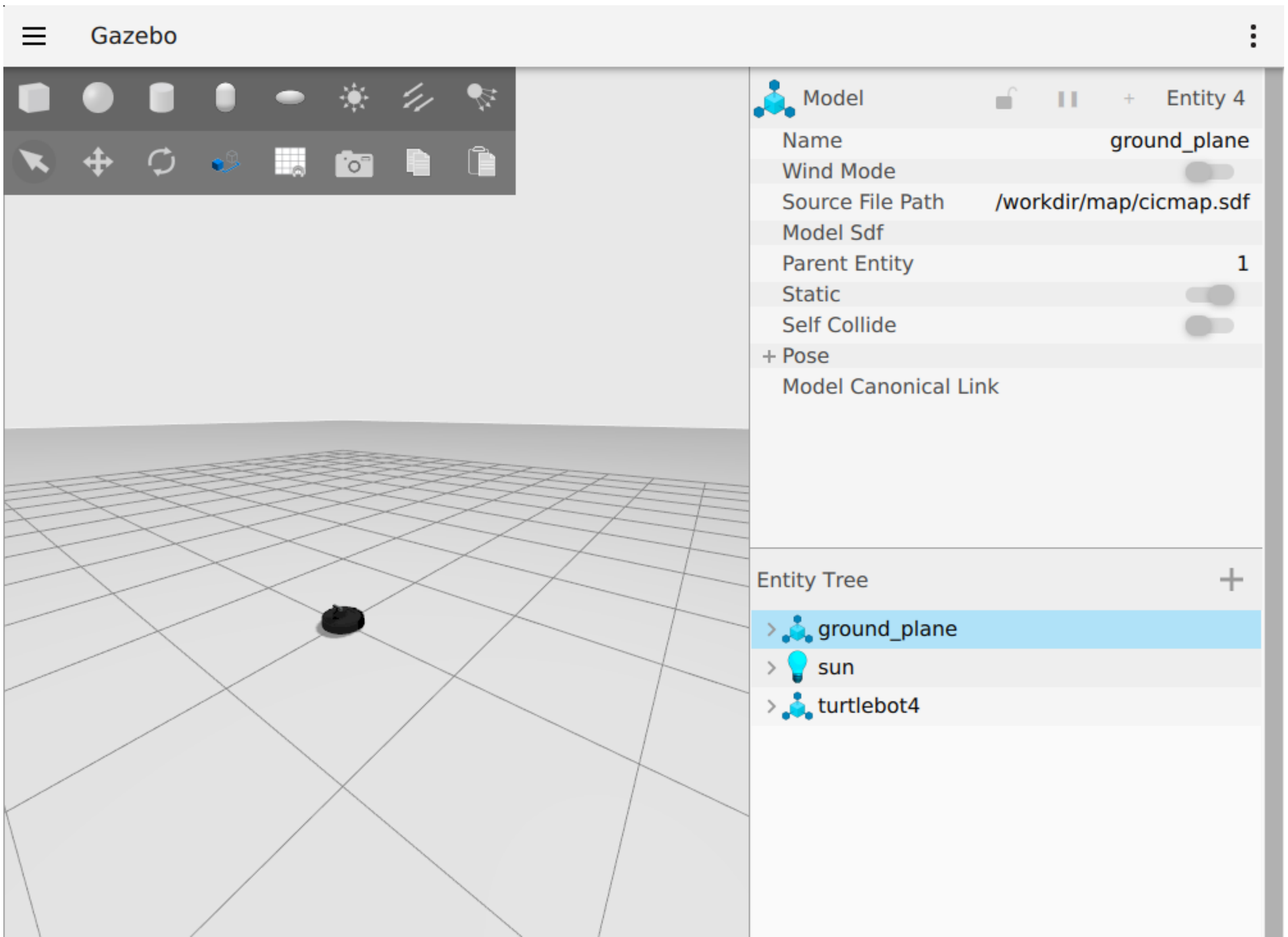
Made by the author (2023)

Figure A.20 – Map scan



Made by the author (2023)

Figure A.24 – Turtlebot4 in the Gazebo Ignition simulator.



Model Entity 4

Name ground_plane

Wind Mode

Source File Path /workdir/map/cicmap.sdf

Model Sdf

Parent Entity 1

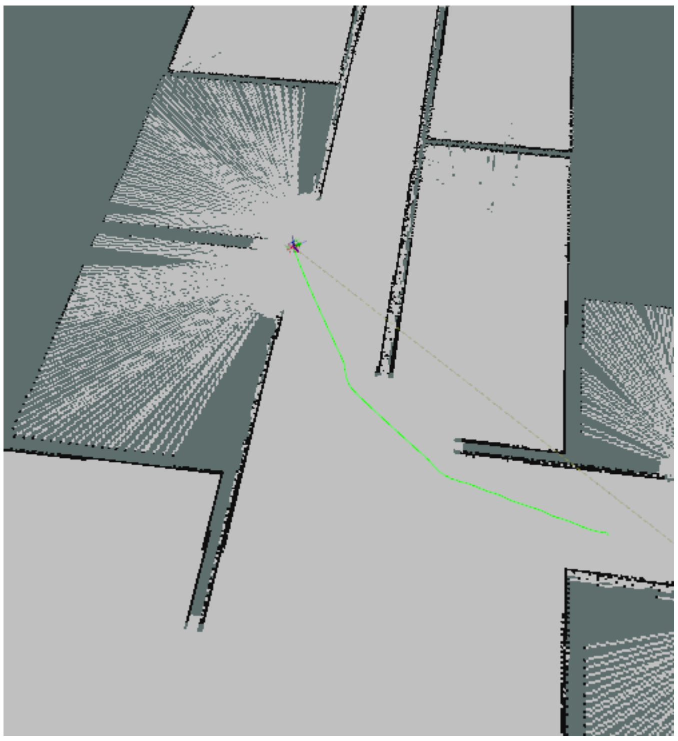
Static

Self Collide

+ Pose

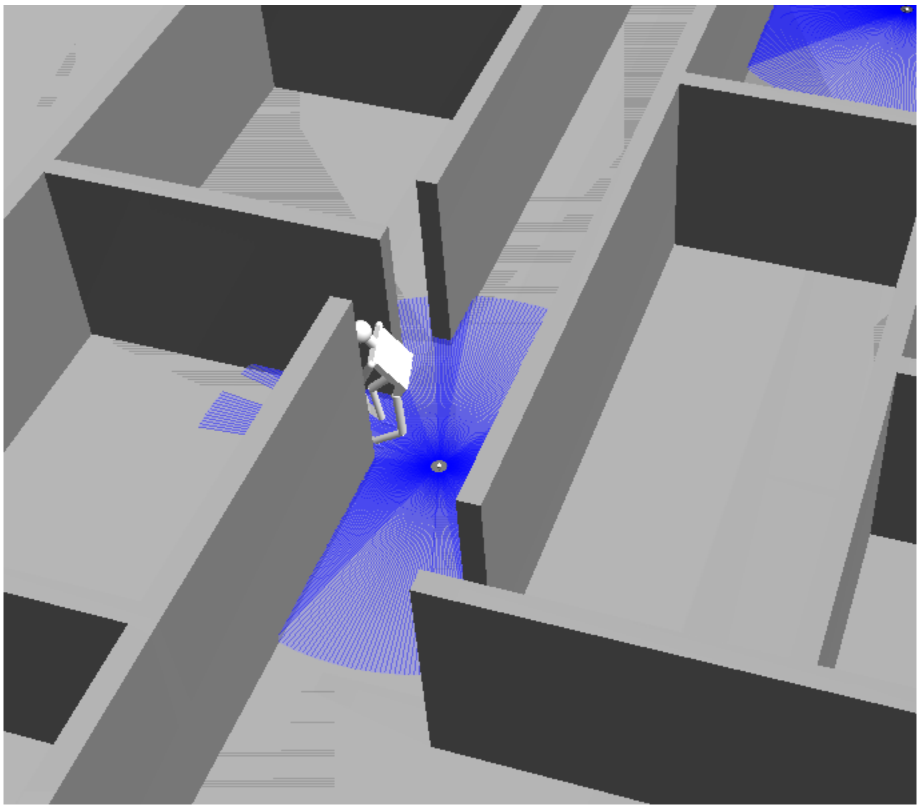
Model Canonical Link

Figure A.21 – Robot navigating through the map. Plotted path in green



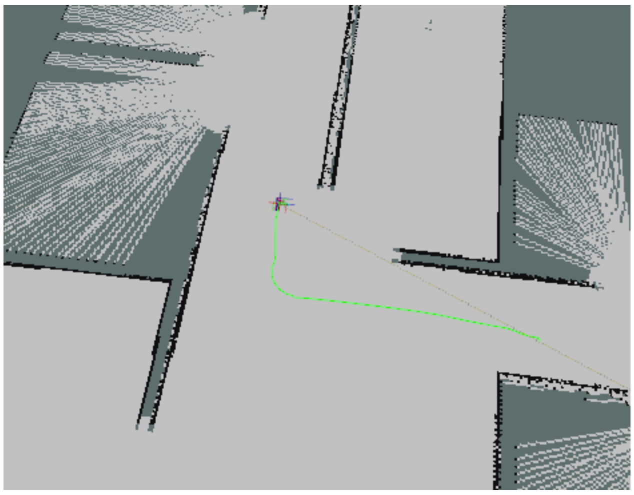
Made by the author (2023)

Figure A.22 – Robot authenticating nurse



Made by the author (2023)

Figure A.23 – Robot navigating through the map. Plotted path in green



Made by the author (2023)