Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Integração do Simulador HMR Sim com o Middleware ROS 2

Kálley Wilkerson Rodrigues Alexandre

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Genaina Nunes Rodrigues

Brasília
2023

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Integração do Simulador HMR Sim com o Middleware ROS 2

Kálley Wilkerson Rodrigues Alexandre

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Genaina Nunes Rodrigues (Orientador)
CIC/UnB

Prof. Dr. Carla Maria Chagas E Cavalcante Koike
Universidade de Brasília

Prof. Dr. Marcelo Antonio Marotta
Universidade de Brasília

Prof. Dr. João Luiz Azevedo De Carvalho
Coordenador do Curso de Engenharia da Computação

Brasília, 24 de fevereiro de 2023

Dedicatória

Eu dedico esta monografia aos entusiastas do mundo da robótica que sonham com o dia em que os robôs tomarão todos os postos de trabalho.

Agradecimentos

Gostaria de agradecer ao Gabriel que me auxiliou a entender o que esse projeto significa e os objetivos a serem alcançados e pela sua paciência desconcertante comigo.

Também agradecer ao meu colega Danilo que trabalhou neste projeto ao meu lado e que foi de grande valia para me ajudar a continuar trabalhando de maneira saudável nele, mesmo que trabalhando em pontos diferentes do projeto.

Agradecer também aos meus amigos do Núcleo de Vida Cristã (NVC) que me ajudaram em oração por semestres por este projeto, isso significou muito para mim.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

A robótica está evoluindo muito e robôs estão sendo cada vez mais utilizados para diferentes tarefas. Isso aumenta a necessidade por sistemas robóticos que controlam os robôs em um ambiente. O desenvolvimento deste tipo de sistema pode passar por uma fase de simulação que muitas vezes se prova essencial para descobrir e resolver problemas antes que o sistema seja testado com robôs reais.

Simuladores para sistemas robóticos, como Gazebo e Webots, podem muitas vezes gerar um alto gasto de recursos computacionais devido principalmente ao alto nível de detalhamento físico dos robôs simulados.

O HMR Sim é um simulador para sistemas robóticos que possui um baixo nível de detalhamento físico, com isso, ele permite uma simulação com uma maior quantidade de robôs a um menor custo computacional. Ele é uma proposta complementar a simuladores de alto detalhamento físico, permitindo validar comportamentos de alto-nível de forma mais eficiente.

No entanto o HMR Sim possui limitações no que tange à forma como os robôs são controlados. O controle dos agentes robóticos é feito através de uma sequência de comandos que fica dentro de um módulo do próprio simulador. Dessa forma, o controle dos robôs simulados fica muito dependente do HMR Sim, quando na verdade, é melhor que esta responsabilidade esteja unicamente com um software separado dele. Esta limitação faz com que fique muito oneroso fazer com que um software de controle de robôs que utiliza o simulador passe a controlar robôs reais.

Este trabalho tem como objetivo mitigar esta limitação através do desenvolvendo de uma camada de comunicação no HMR Sim que permitirá que outros softwares controlem os robôs sendo simulados. Esta camada de comunicação é feita integrando-se o HMR Sim ao ROS, que é um conjunto de bibliotecas e ferramentas que funciona como um protocolo de comunicação entre softwares. Uma vantagem do ROS é que ele já é muito conhecido na literatura e possui uma comunidade de milhões de desenvolvedores e usuários, aumentando as chances de um software controlador já está familiarizado com a ferramenta.

Adicionalmente, foi implementado um software para controlar os robôs simulados em Python utilizando *Behaviour Trees*, que é uma forma de controlar os comportamentos de

uma entidade utilizando conceitos de árvores.

Como resultado, a nova camada de comunicação implementada agora permite que um software separado do simulador possa controlar a navegação e garras dos robôs simulados utilizando as mesmas interfaces ROS que eles utilizariam com robôs reais.

Palavras-chave: ROS, simulador, behavior trees

Abstract

Robotics is evolving a lot and robots are being used more and more for different tasks. This fact increases the need for robotic systems that control robots in an environment. The development of systems like this can go through a simulation phase which often proves essential to find and solve problems before the system is tested with real robots.

Simulators for robotic systems, such as Gazebo and Webots, often generate a high usage of computational resources mainly due to the highly detailed physics of simulated robots.

The HMR Sim is a simulator for robotic systems that has low detailed robots, thus allowing a simulation with a greater number of robots at a lower computational cost.

However, HMR Sim has limitations regarding the way robots are controlled. The control of robos is done through a sequence of commands that is within a module of the simulator itself. Thus, the control of the simulated robots is very dependent on HMR Sim, however, it is better that this responsibility is with a software separated from it. This limitation makes it very costly to make a robot control software that uses the simulator start to control real robots.

This work aims to mitigate this limitation by developing a communication layer in HMR Sim that will allow other software to control the robots being simulated. This communication layer is done by integrating HMR Sim with ROS, which is a set of libraries and tools that works as a communication protocol between softwares. An advantage of ROS is that it is already well known in the literature and has a community of millions of developers and users, increasing the chances that a controller software is already familiar with the tool.

Additionally, software was implemented in python to control the simulated robots using *Behaviour Trees*, which is a way to control the behavior of an entity using tree concepts.

As a result, the new communication layer implemented now allows a software separated from the simulator to control the navigation and claws of the simulated robots using the same ROS interfaces that they would use with real robots.

Keywords: ROS, simulator, behavior trees

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

API Application programming interface.

callbacks processos que devem ser executados para tratar um comando que chegou via ROS.

CAPES Coordenação de Aperfeiçoamento de Pessoal de Nível Superior.

ECS Entity-Component-System.

FSM Finite State Machines.

HMR Sim Heterogeneous Multi-Robot Simulator.

model é uma forma de padronizar atributos/componentes de entidades a depender de atributos colocados nelas no diagrama original que gera o ambiente.

MORSE Modular Open Robots Simulation Engine.

Nav2 Navigation 2.

NPC non-player character.

NVC Núcleo de Vida Cristã.

RAM Random-access memory.

ROS Robot Operating System.

spin processo da biblioteca ROS que executará os callbacks conectados caso necessário.

Capítulo 1

Introdução

Muitos ambientes como escritórios, hospitais e escolas possuem uma diversidade de tarefas a serem realizadas que poderiam ser substituídas por robôs futuramente. É para casos como estes em que pode-se ser utilizado um sistema multi-robôs (sistema que controla mais que um único robô) [1].

Sistemas multi-robôs já são desenvolvidos atualmente e para ajudar na validação deles, os desenvolvedores muitas vezes utilizam simuladores de ambientes robóticos, o que contribui para a redução dos custos de produção do sistema descobrindo-se, por exemplo, falhas que poderiam ser detectadas já na simulação, sem precisar que o sistema tenha sido de fato implementado com robôs reais.

Atualmente, simuladores tais quais Gazebo [3] e Webots [4] suportam simulações multi-robôs mas a um alto custo computacional devido ao alto nível de detalhamento dos agentes robóticos sendo simulados. [1]

O HMR Sim [1] é uma proposta de simulador para controle a um nível mais abstrato de robôs, permitindo que ele possa simular mais robôs a um custo computacional menor. Ele portanto é útil em casos onde o controle dos robôs é feito a nível de missão e não de sensores e atuadores. Nele, os detalhes de implementação dos agentes do sistema são abstraídos em prol de um maior foco em suas habilidades e capacidades. Por exemplo, se um robô é capaz de levantar objetos pequenos de certo peso, as implementações específicas de controle de hardware para isso ser realizado são abstraídas e somente os efeitos desta ação são considerados no simulador (como o gasto de bateria resultante da operação). [1]

No entanto o HMR Sim possui algumas limitações. Em especial o controle dos robôs dentro do simulador atualmente é realizado através de uma sequência de comandos predefinidos acoplados ao próprio robô. Por exemplo, uma sequência de comandos para um robô poderia ser para ele ir a uma certa posição, pegar um objeto, levá-lo a uma outra posição e largar o objeto na nova posição. Esta forma de controle é limitada pois: **(i)** é difícil migrar os comandos de controle do simulador para robôs reais, pois os comandos

não são suportados em *middlewares* de programação de robôs amplamente utilizados na literatura; **(ii)** os sistemas simulados tem limitação quanto a autonomia de planejamento e tomada de decisão - uma vez que os robôs apenas realizam uma sequência de comandos previamente definidas, não sendo possível tomar decisões a partir da percepção do robô sobre o ambiente em tempo de execução.

O Robot Operating System (ROS) [5] é um middleware para sistemas robóticos popular na literatura e considerado o padrão de facto no desenvolvimento de software para controle de robôs. É um conjunto de bibliotecas e ferramentas que ajuda os desenvolvedores a criar aplicações robóticas. Possui uma comunidade com milhões de desenvolvedores e usuários e oferece uma plataforma de software padrão e pode ser utilizado para criação de uma nova camada no simulador responsável pela comunicação com sistemas externos. [6]

A utilização do ROS junto ao HMR Sim permitirá que sistemas externos possam controlar os agentes robóticos sendo simulados tal como já é feito nos ambientes de desenvolvimento de aplicações robóticas reais.

A limitação quanto à autonomia de planejamento e tomada de decisão poderia ser mitigada através de um sistema de controle que utilizasse *behaviour trees* e que fosse completamente desacoplado do simulador. *Behaviour tree* é uma forma de definir o comportamento de uma entidade através do conceito de árvores.

1.1 Objetivos Gerais

O objetivo deste trabalho é permitir a utilização do HMR Sim para exercitar software de controle de robôs que seja realista, isto é, capaz de controlar robôs reais.

Para tanto, pretende-se desenvolver uma camada de integração com o middleware ROS. A implementação desta camada será através de novos módulos dentro do simulador, cada um com a responsabilidade de adaptar os já existentes para enviar e receber comandos através do ROS.

A integração com o ROS, poderá diminuir a complexidade de adaptar uma aplicação desenvolvida em um ambiente simulado para passar a utilizar robôs reais e vice-versa. Além de deixar mais clara a separação entre o que é software controlador e o que é simulador, pois estarão sendo executados em processos separados.

Contribuições

De forma efetiva, as contribuições alcançadas neste trabalho são as seguintes:

- Prover uma camada de integração que permita o controle de robôs na simulação utilizando o ROS.

- Prover interfaces ROS para controle de navegação dos robôs
- Prover interfaces ROS para controle da capacidade de manipulação dos robôs (e.g., pegar e largar objetos)
- Permitir o controle de robôs por meio de *Behavior Trees*.

1.2 Organização do Trabalho

Primeiro será feita uma pequena introdução ao HMR Sim e ao propósito de sua existência no Capítulo 1. Junto à introdução terá uma explanação mais detalhada dos objetivos deste trabalho. Depois, no Capítulo 2, será apresentado um referencial teórico onde serão apresentados e explicados conceitos relevantes para fundamentar o restante deste trabalho como o próprio HMR Sim, conceitos relevantes para o trabalho do ROS e *Behaviour Trees*. No Capítulo 3 será detalhado o processo de implementação dos objetivos apresentados na Introdução. O Capítulo 4 mostrará dois cenários de execução das implementações realizadas no projeto. Por fim, no Capítulo 5 serão comentadas as conclusões deste trabalho.

Capítulo 2

Referencial Teórico

Neste capítulo serão abordados e explicados conceitos que serão relevantes para o bom entendimento da implementação do projeto. Na Seção 2.1 o simulador HMR Sim [1] é apresentado e são explicados conceitos relevantes sobre seu estado atual que serão utilizados posteriormente. A Seção 2.2 fala sobre o ROS e o que for relevante ser falado sobre ele para o bom entendimento do restante do projeto, como conceitos de Node, Action, etc. Na Seção 2.3 são referenciados outros simuladores que também implementam uma comunicação via ROS. A Seção 2.4 explica o conceito de *Behaviour trees*, que será utilizado na implementação dos objetivos deste projeto.

2.1 HMR Sim

O HMR Sim é um simulador desenvolvido sob a proposta de ser um simulador para controle mais abstrato de robôs, onde o controle é feito a nível de missão. Em casos como este, o controle de sensores e atuadores dos robôs é simulado dentro do próprio HMR Sim, o que permite que simulações com uma maior quantidade de robôs possa ser executada sem a necessidade de uma quantidade inacessível de recursos computacionais. Neste sentido, os robôs simulados são agentes que gerenciam o controle de seus próprios sensores e atuadores. O simulador então é útil para testes de algoritmos de coordenação de times de robôs. [1]

A Figura 2.1 mostra uma tela de execução do simulador no navegador. Mostra o título do cenário, os dados dos passos sendo executados e botões de controle da visualização da simulação. Mostra também um mapa com paredes em azul, robô em amarelo e um objeto que pode ser apanhado em vermelho.

O HMR Sim se propõe a trabalhar em conjunto com outros simuladores já presentes na literatura, em uma etapa anterior a utilização deles, onde o controle é a nível de missão, uma vez que em um sistema com muitos robôs pode-se não entrar nas especificidades de

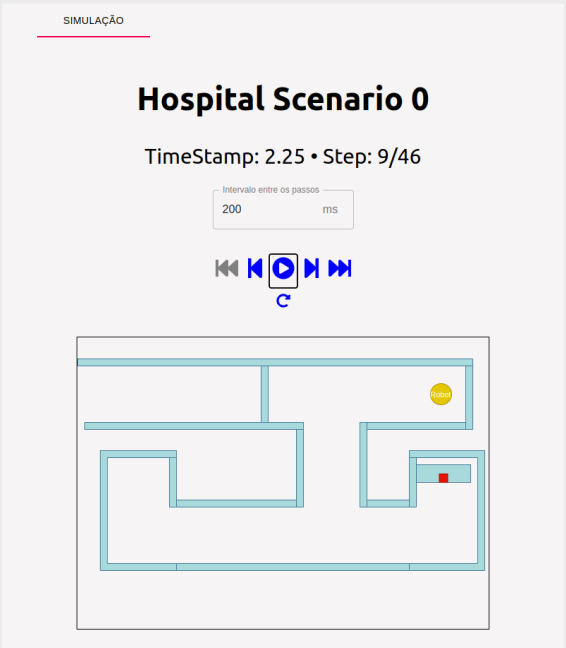

Figura 2.1: Exemplo de mapa utilizado em uma execução do HMR Sim. Fonte: De autoria própria.

cada robô o que torna mais útil uma simulação mais simplificada deles. Em uma outra etapa do desenvolvimento de um sistema robótico, poderá então ser utilizada a simulação com controle a um nível mais baixo dos robôs, a nível de sensores e atuadores.

2.1.1 Arquitetura

O simulador é feito com arquitetura Entity-Component-System (ECS), o que significa que conceitos como entidades, componentes e sistemas são conceitos chave no projeto [1]. A Figura 2.2 demonstra o funcionamento de uma arquitetura ECS, onde **entidades** possuem **componentes** e cada **sistema** na estrutura é responsável por gerenciar entidades que possuem componentes específicos. Por exemplo, um sistema de movimentação de robôs que gerencia entidades que possuem `Velocidade` e `Posição` como componentes e um sistema para gerenciar o gasto de energia que gerencia entidades com componente `Bateria`.

As **entidades** normalmente representam objetos físicos como robôs, pessoas e paredes e são gerenciadas pela biblioteca **esper** [7]. Os **componentes** de uma entidade indicam o que ela é capaz de fazer, por exemplo uma entidade possuir um componente de velocidade significa que ela pode se movimentar e o componente guarda a velocidade de seu movimento. E por fim os **sistemas** que são quem de fato implementam as capacidades e comportamentos das entidades e do simulador como um todo. Um exemplo de sistema pode ser um *sistema de movimentação*, que pode por exemplo acessar os componentes de

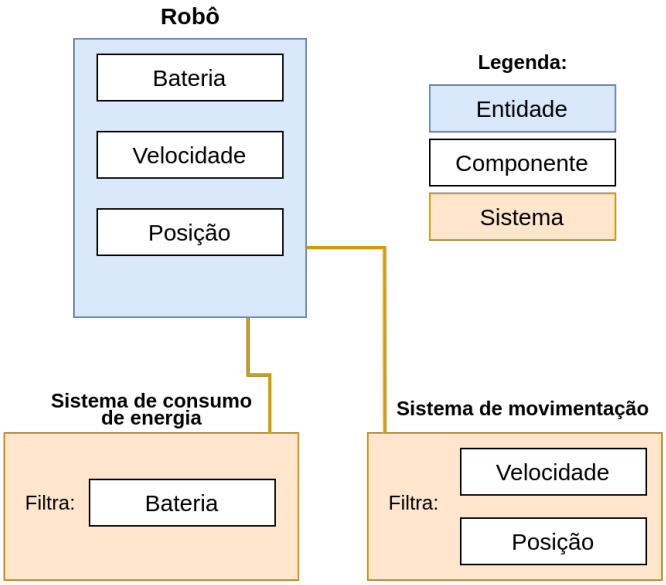

Figura 2.2: Demonstração de como funciona uma arquitetura ECS, exemplificando com o que é utilizado no HMR Sim. Fonte: De autoria própria, baseado em [1], página 12.

posição e velocidade de uma entidade e utilizar os dados desses componentes para mudar a entidade para um novo estado, e.g., novos valores de posição e velocidade.

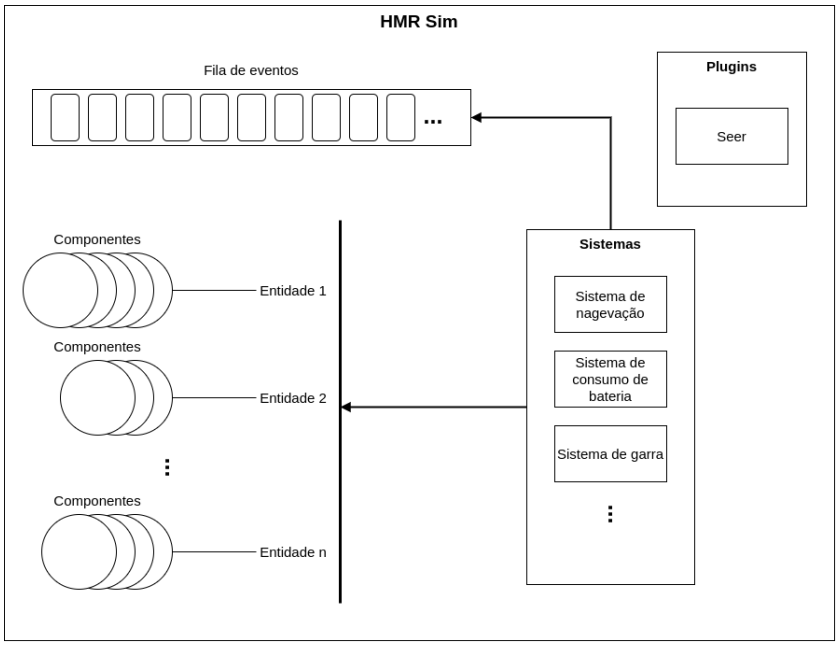

Figura 2.3: Diagrama representando a arquitetura do HMR Sim. Fonte: De autoria própria baseado em [1], página 35.

A Figura 2.3 mostra a arquitetura do HMR Sim, onde os sistemas dependem das entidades e da fila de eventos. A figura também mostra três dos sistemas já presentes no

simulador antes da implementação deste projeto.

O HMR Sim utiliza **eventos discretos** como uma técnica de simulação, onde uma **fila de eventos** é observada por sistemas, que são executados à medida que eventos relacionados são lançados. Os eventos dentro do HMR Sim funcionam como uma interface de comunicação entre os sistemas. Eles permitem que os sistemas possam se comunicar entre si ao mesmo tempo que não precisam conhecer a implementação um do outro. Neste documento, ao falar sobre sistema, refere-se a sistema nesta arquitetura ECS.

2.1.2 Sistemas do HMR Sim

Aqui serão tratados dois sistemas presentes no HMR Sim que serão relevantes para este projeto. Os sistemas tratados aqui não são parte deste projeto mas já existiam antes dele dentro do HMR Sim e seu conhecimento ajudará na compreensão dos objetivos do mesmo.

Como dito anteriormente, um dos objetivos deste projeto é a implementação de uma interface ROS para comunicação com o HMR Sim, ou seja, muitos sistemas já presentes dentro do HMR Sim não precisarão ser alterados, sendo portanto reaproveitados e, quando necessário, complementados com uma camada de comunicação com o ROS.

Sistema de Navegação

Este é um sistema responsável por realizar a navegação dos robôs dentro da simulação. Ele utiliza componentes como os de posição e velocidade para tal, alterando os valores destes componentes para simular a navegação dos agentes robóticos.

Neste projeto o objetivo é continuar utilizando este sistema de navegação. O que será feito é a criação de outro sistema responsável por receber os comandos via ROS que traduzirá os comandos recebidos para o sistema atual de navegação. Observe que a utilização deste método permite que o sistema atual de navegação possa ser alterado sem afetar diretamente a comunicação via ROS.

Sistema ClawProcessor

O sistema `ClawProcessor` é responsável pela execução de ações relativas a garra de robôs que possuem o componente de garra. São as ações de pegar e a de largar um objeto.

A garra no HMR Sim é um componente que permite que o agente robótico possa pegar e soltar objetos de um certo peso que estejam a uma certa distância. Dentro do próprio componente se define o peso máximo e a distância máxima que a garra pode carregar.

Este projeto irá reutilizar este sistema, criando um outro sistema que ficará responsável por receber os comandos via ROS e traduzi-los para que possam ser executados pelo

sistema atual de garra. Dentro da simulação terão objetos rotulados com nomes, esses nomes poderão ser recebidos via ROS para indicar qual objeto deve ser apanhado/solto.

2.2 *Robot Operating System* (ROS)

O Robot Operating System (ROS) é um sistema distribuído, um conjunto de bibliotecas e ferramentas *open-source* para criação de aplicações robóticas [2], funcionando como um *middleware* entre a aplicação e os robôs. Ele permite controlar dispositivos a baixo nível e implementa funcionalidades normalmente utilizadas, troca de mensagens entre processos e gerência de pacotes. Um dos principais conceitos do *ROS* é o de *Node* (ou Nó).

2.2.1 Node

Um **node** (ou nó) no *ROS* é um processo que faz computações. Nodes são combinados um com o outro em um **gráfico** e se comunicam entre si através de **tópicos** por exemplo. Um **gráfico** do ROS é uma rede de elementos do ROS 2 que processam dados juntos ao mesmo tempo. Um sistema de robôs normalmente terá vários nodes. Por exemplo, um node poderia estar controlando os motores das rodas do agente robótico, um implementando localização, outro fazendo planejamento da rota, um outro fazendo uma visualização gráfica do sistema, etc.

Uma vantagem em utilizar nodes é que eles permitem que exista uma tolerância adicional à falhas uma vez que uma falha fica isolada em apenas um node, a complexidade do código diminui em comparação com sistemas monolíticos, detalhes de implementação também ficam bem escondidos uma vez que só é exposta uma *API* mínima para comunicação com outros nodes. [8]

2.2.2 Tipos de Comunicação

O *ROS* define tipos de comunicação entre clientes e servidores. Serão citados aqui *Tópicos* (topics), *Serviços* (services) e *Actions*. Estas comunicações se diferenciam por terem ou não requisições e respostas.

Tópicos

Os tópicos são barramentos rotulados por onde os *nodes* trocam mensagens. Eles têm uma semântica *publish/subscribe*, o que separa a produção de uma informação de seu consumo. Em geral, os nodes não sabem com quem estão se comunicando, mas caso tenham interesse, apenas se "inscrevem" em um certo tópico, e outros que têm interesse

podem “publicar” nesse mesmo tópico. A figura 2.4 ilustra a estrutura de um tópico do ROS.

Mas os tópicos servem apenas para uma comunicação unidirecional, se algum node precisa receber uma resposta a uma requisição que ele tenha feito, vai precisar utilizar **serviços** ao invés de tópicos. [9]

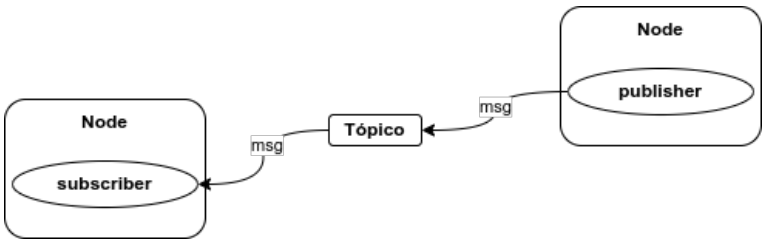

Figura 2.4: Estrutura de um tópico do ROS. Fonte: De autoria própria, baseado em [2].

Serviços

Um modelo alternativo a uma interação *publish/subscribe* é a *request/reply* que deve ser utilizada quando um node precisa receber resposta a uma requisição feita. Este tipo de interação é feita através de serviços no ROS. Um serviço é feito por duas mensagens, uma de requisição e outra de resposta. Então, um node *ROS* provedor oferece um serviço e um cliente utiliza esse serviço enviando uma requisição e esperando uma resposta. A figura 2.5 ilustra a estrutura de um serviço do ROS. [10]

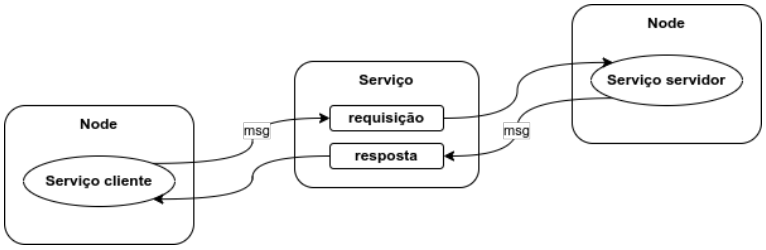

Figura 2.5: Estrutura de um serviço do ROS. Fonte: De autoria própria, baseado em [2].

Actions

Actions definem a execução de uma ação, como pegar um objeto, e foram criadas para serem utilizadas em processos longos. Uma **Action** [1] têm três partes: objetivo (goal), feedback e resultado (result). Cada uma dessas partes é um outro tipo de comunicação do *ROS*: objetivo e resultado são *serviços* enquanto que feedback é um *tópico*.

[1]Será chamada dessa forma no restante do trabalho, Action ROS

Actions utilizam o modelo cliente-servidor, a estrutura pode ser vista na figura 2.6. A "action do cliente" envia o objetivo através do serviço objetivo e a "action do servidor" recebe esse objetivo, processa e retorna se o aceita ou não. O cliente então envia uma requisição pelo serviço de resultado e, enquanto espera, fica ouvindo feedbacks pelo tópico de feedbacks. [11]

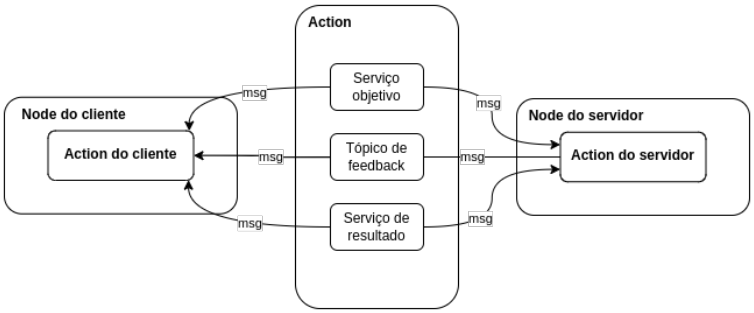

Figura 2.6: Estrutura de uma Action no ROS. Fonte: De autoria própria, baseado em [2].

2.2.3 Interfaces de Comunicação

Existem projetos relacionados ao *ROS* que incrementam suas funcionalidades de diferentes formas [12], tornando a comunicação mais robusta. Um destes projetos é o Nav2 [13]. Ele é um projeto com o objetivo de encontrar uma forma segura de fazer um agente robótico se mover de um ponto A a um ponto B. Ele pode ser utilizado por diferentes sistemas para controle da parte de navegação de um agente robótico. Outro projeto é o MoveIt [14], que é um outro que engloba as funcionalidades do Nav2 e mais várias outras, entre elas o controle das garras de um robô.

O HMR Sim utilizará interfaces ROS definidas nestes projetos para fazer a comunicação via ROS. Como eles são projetos já bastante utilizados na literatura, sua utilização permitirá que mais sistemas robóticos consigam interagir com o HMR Sim.

2.3 Trabalhos Relacionados

Já existem simuladores para sistemas robóticos no mercado e vários deles já implementam interfaces ROS para que possam se comunicar com softwares externos. A seguir serão tratados três simuladores existentes.

O Gazebo [3] possui integração tanto com ROS 1 quanto com ROS 2 [15]. Ele está em um processo de migração para o ROS 2 e a versão com o ROS 2 já está bem desenvolvida, com pacotes para diversos tipos de comandos ROS. Os pacotes são mais focados em comandos para sensores e atuadores [16].

O MORSE [17] também possui interfaces ROS mas como o projeto não é mais mantido o simulador utiliza a versão ROS 1 e não tem suporte para ROS 2. Parecido com o Gazebo, os pacotes presentes no MORSE são pacotes para sensores e atuadores. [18]

O Webots [4] também possui um pacote com interfaces para comunicação via ROS 2. A interface ROS 2 do Webots possui uma vasta gama de funcionalidades disponíveis mas focada em sensores e atuadores também [19].

Como o controle de apenas um robô a nível de sensores e atuadores é mais presente na literatura, os simuladores existentes tendem a focar mais na comunicação ROS específica para casos assim. O HMR Sim é um simulador para casos onde o controle dos robôs é feito a nível de missão, ou seja, onde o controle de sensores e atuadores é abstraído e de responsabilidade do próprio robô.

2.4 Behaviour Trees

Behaviour trees definem o comportamento de entidades. Uma entidade cujo comportamento é definido por uma *behaviour tree* pode ser, por exemplo, um robô ou um NPC em um jogo. Os nós desta árvore precisam ser processados e o resultado desse processamento é um **estado** daquele nó, que pode ser *RUNNING*, *SUCCESS* ou *FAIL*. O estado de um nó vai depender do(s) estado(s) do(s) nó(s) filho(s) caso ele(s) exista(m) e essa dependência pode ocorrer de diversas formas.

O processamento da árvore ocorre através de "ticks", que são sinais de ativação com uma certa frequência. Um nó só é executado se, e somente se, ele tiver recebido um tick. Tick é uma requisição que um nó faz ao nó filho. Quando é executado, o nó pode retornar um resultado ou passar a execução para um nó filho através de um tick. Diferentes tipos de nó podem ter diferentes formas de decidir qual nó filho deve ser executado e a depender de como cada nó executa a execução da árvore como um todo pode ficar parada em único nó enquanto ele não retorna um resultado diferente de *RUNNING*.

Cada nó executado retorna um estado e se o estado retornado por ele for *RUNNING*, significa que ele ainda não tem um resultado definitivo (*SUCCESS* ou *FAIL*). A partir do momento em que um nó tem um resultado definitivo, a execução para para o nó pai dele.

Uma *behaviour tree* pode ter **nós de controle** (que estão no meio da árvore) ou **nós folha** (no final dela). Os nós de controle podem ser de quatro tipos: sequência, fallback, paralelo e decoradores. Os nós folha podem ser nós de ação ou nós de condição. [20]

2.4.1 Nós de Controle

- **Sequência:** Ao receber um tick, transmite para os filhos percorrendo cada um deles, se algum deles retornar *FAIL* ou *RUNNING*, então o nó retorna *FAIL* ou

RUNNING. Ele retorna *SUCCESS* se todos os filhos retornarem *SUCCESS*.

- **Fallback:** Transmite o tick para os filhos percorrendo cada um deles, se algum filho retornar *SUCCESS*, ele também retorna. Ele retorna *FAIL* se todos os filhos retornarem *FAIL*.
- **Paralelo:** Se o nó tiver N filhos, ele retorna *SUCCESS* se pelo menos M filhos retornarem *SUCCESS*.
- **Decorador:** É um nó que tem apenas um filho e quando ele recebe um tick, passa o tick para esse filho que retorna um status que depende de alguma política customizada. Pode ser utilizado para adicionar uma semântica adicional à árvore ou para mudar o retorno de um nó por exemplo. [21]

2.4.2 Nós Folha

Um nó folha pode definir por exemplo uma ação ou uma condição. Os nós de ação executam um comando, retornando *SUCCESS* se o comando rodou corretamente ou *FAIL* caso contrário. Os de condição checam uma condição, pode retornar *SUCCESS* ou *FAIL* a depender da condição testada. [20]

A Figura 2.7 mostra um exemplo simples de uma *behaviour tree* para a ação de pegar um objeto. Nela, o nó com uma seta é sequência, nós com "?" são fallbacks, elipses são condições e retângulos são ações.

2.4.3 Bibliotecas Utilizadas

Existem bibliotecas Python para uso de *Behaviour Trees* e a que será utilizada neste projeto é a *Py Trees* [22]. Existe ainda uma biblioteca que mescla o uso da biblioteca *Py Trees* com o ROS, a *Py Trees ROS* [23]. Estas duas bibliotecas serão utilizadas neste projeto em um módulo responsável pelo controle dos robôs simulados, onde se terá uma *behaviour tree* onde um dos nós se comunicará com o HMR Sim para controlar o robô simulado.

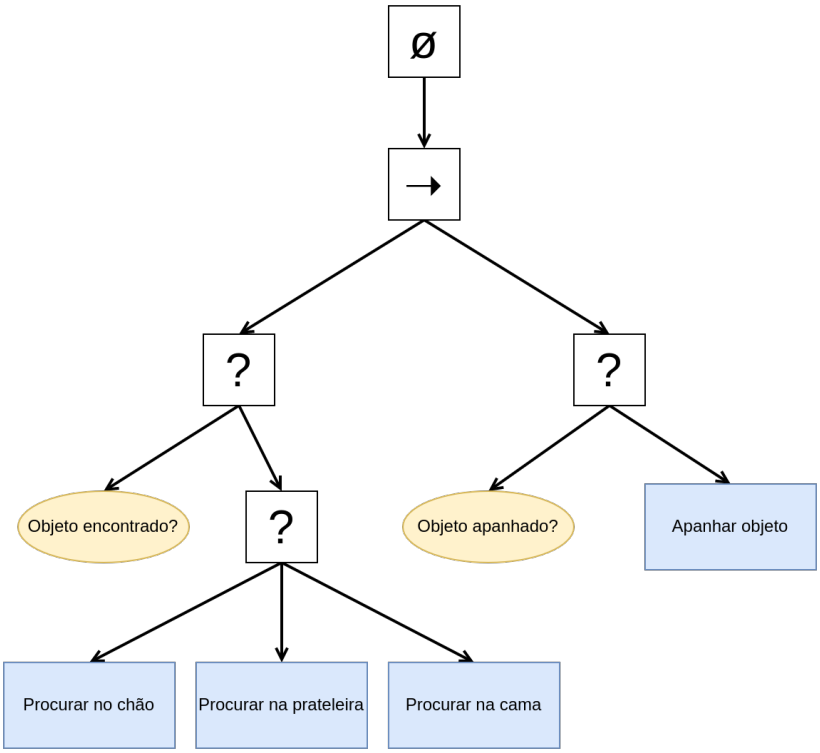

Figura 2.7: Exemplo de uma behaviour tree representando a ação de pegar um objeto. Fonte: De autoria própria.

Capítulo 3

Integração entre HMR Sim e ROS

Neste capítulo serão apresentados os pontos de extensão do HMR Sim para a realização dos objetivos deste trabalho. Este capítulo está estruturado da seguinte forma: A Seção 3.1 discorrerá sobre as mudanças feitas na arquitetura do simulador. A Seção 3.2 tratará da implementação da integração entre HMR Sim e ROS. Por fim, a Seção 3.3 falará sobre a implementação do software de controle dos robôs simulados utilizando **Behaviour Trees**.

Vale ressaltar que o termo **sistema** refere-se ao sistema de uma arquitetura Entity-Component-System (ECS), que é a utilizada no HMR Sim. Dessa forma, conforme previamente explicado no Capítulo 2, sistemas são módulos do simulador que implementam as capacidades e comportamentos das entidades e do simulador como um todo. A versão do ROS utilizada neste projeto é o ROS 2 Foxy [2].

3.1 Arquitetura do HMR Sim

Foram adicionados novos sistemas responsáveis por intermediar uma comunicação entre os sistemas já existentes no simulador e uma aplicação controladora fora dele. Na Figura 3.1 destaca-se as alterações feitas na arquitetura do simulador, estando em verde os novos módulos criados.

Os sistemas adicionados foram sistemas de comunicação para receber comandos via ROS, um para controle da navegação do robô e um para a garra do robô. O software controlador então terá disponível uma interface ROS para enviar comandos de navegação para um robô e também interfaces de para comandos de pegar um objeto e soltar um objeto. Foi também criada uma classe para gerenciamento dos sistemas de comunicação. Esta classe servirá como uma conexão entre o loop principal do simulador e os sistemas ROS criados, tornando possível a inserção e remoção destes sistemas mais dinamicamente.

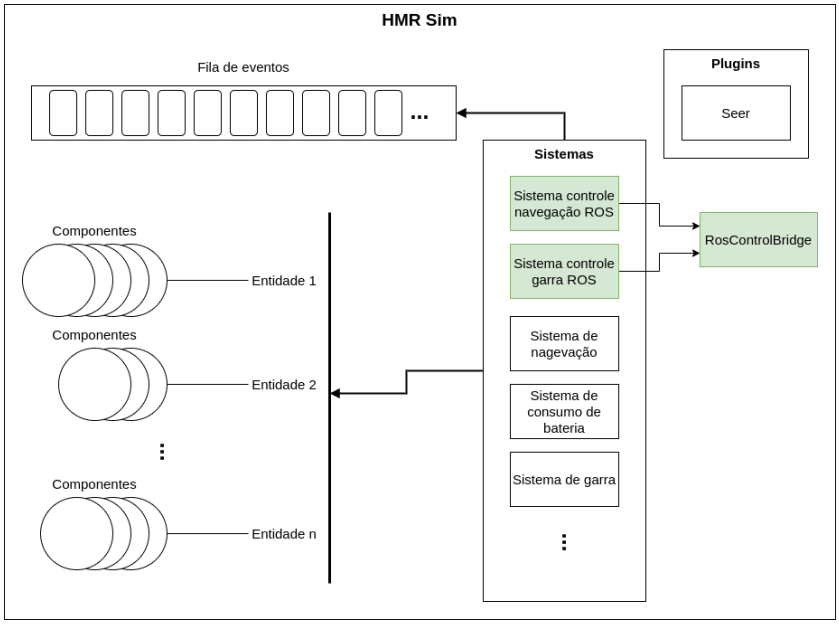

Figura 3.1: Novos módulos adicionados ao HMR Sim em destaque. Fonte: De autoria própria baseado em [1], página 35.

A grande diferença entre o estado anterior do HMR Sim e o atual com a implementação deste projeto está na dinâmica da interação entre controlador e o simulador. Por exemplo, anteriormente a este projeto, caso o controlador precisasse alterar os comandos que um robô deverá executar, ele teria que parar a simulação, alterar o arquivo onde é desenhado o mapa colocando os novos comandos nele e então reiniciar a simulação. Com a implementação deste projeto, ele poderá executar qualquer comando disponível a qualquer momento sem precisar parar a simulação.

3.2 Componentes e Sistemas da Integração

O objetivo deste trabalho é prover uma camada dentro do HMR Sim responsável pela comunicação dele via ROS com outros softwares responsáveis pelo controle dos robôs simulados.

Para que haja uma comunicação entre dois processos via ROS, eles precisam convencionar uma interface para realizá-la. Com o HMR Sim não deve ser diferente, então o simulador precisará escolher interfaces para se comunicar.

Foi decidido que devem ser escolhidas as interfaces que são mais populares na literatura robótica. Por exemplo, para controle da navegação de um robô (que deve ser uma Action) a biblioteca Nav2 [13] especifica a interface mais popular para ROS 2, a `nav2_msgs/action/NavigateToPose` (citada na Seção 3.3). Então, neste caso, a aplicação controladora utilizaria esta interface tanto para simulação quanto com robôs reais. A

principal vantagem desta decisão é a diminuição do retrabalho do software controlador de passar a enviar comandos para robôs reais no lugar dos robôs simulados, pois ele estará utilizando a mesma interface de comunicação para ambos.

Em casos de problemas com a conexão entre o simulador e o controlador, a própria biblioteca do ROS utilizada no HMR Sim fica responsável por lidar com eles. Por exemplo, caso o controlador se desconecte enquanto uma ação está sendo executada, o simulador enviará o resultado via ROS sem receber/esperar informações se ela foi recebida com sucesso ou não, deixando para a biblioteca ROS a responsabilidade de lidar com o problema, que neste caso apenas não enviará a resposta para o cliente.

3.2.1 Tática de Integração

A primeira decisão arquitetural para viabilizar a integração de nodos ROS com o HMR Sim consiste na adoção do padrão de projeto Bridge [24]. Este padrão permite a integração de diversas implementações de serviços similares. Este padrão de projeto foi utilizado pois existirá um tipo abstrato para sistemas que receberão comandos via ROS (definindo um comportamento geral para todos eles) e os sistemas que serão utilizados durante a simulação são definidos na inicialização da mesma.

A classe `RosControlBridge` foi criada para servir como uma forma de agrupar em uma mesma classe as diversas classes ROS que poderão ser utilizadas, pois elas terão comportamentos similares. Em uma instância desta classe serão adicionados objetos do tipo genérico `RosService`. `RosService` especifica como deve se portar um sistema que implementa uma comunicação via ROS [1] dentro do HMR Sim. Implementações de `RosService` são "cadastradas" em uma instância de `RosControlBridge` durante a configuração do simulador, antes da execução da simulação. A Figura 3.2 ilustra a estrutura da classe `RosControlBridge`.

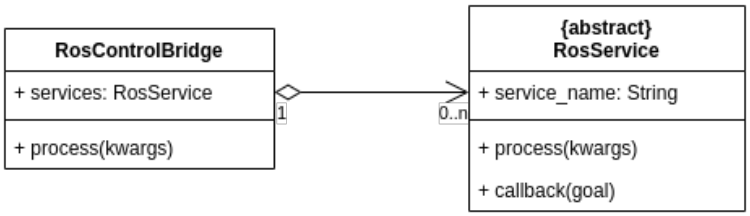

Figura 3.2: Estrutura da classe de controle dos serviços ROS. Fonte: De autoria própria.

A Figura 3.3 ilustra como funciona a execução da classe `RosControlBridge`. A cada intervalo de tempo o método `process()` é executado, ele faz um spin [2] no nó ROS, que

[1] Actions, Tópicos e Serviços

[2] processo da biblioteca ROS que executará os callbacks conectados caso necessário

poderá executar os callbacks [3] dos serviços conectados, e depois ele executa os métodos `process()` dos serviços conectados. Cada serviço pode executar um callback do ROS e/ou um método periódico (executado independente de uma chamada via ROS ou evento).

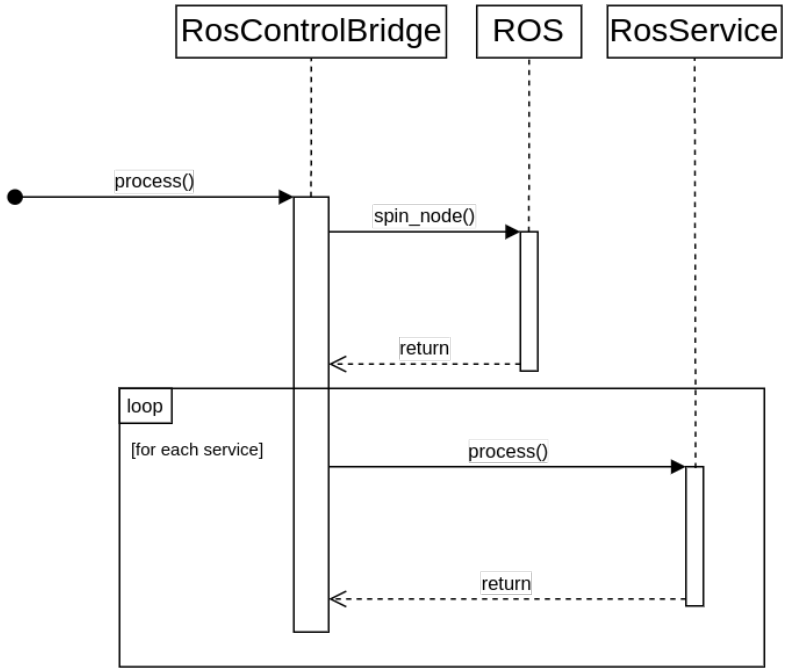

Figura 3.3: Diagrama de sequência mostrando a execução execução da classe RosControlBridge. Fonte: De autoria própria.

Uma especificação do `RosService` é a `RosActionServer`, que identifica uma *Action* ROS. E o sistema mostrado na Seção 3.2.1 é uma implementação de um `RosActionServer`.

Com a utilização do padrão de projeto *Bridge* obteve-se um desacoplamento em relação aos restante da simulação e as interfaces ROS. Isso permitiu implementar novas interfaces sem ter que modificar os outros sistemas existentes, colaborando com a extensibilidade do HMR Sim.

Navegação com ROS

Para que o HMR Sim pudesse receber comandos de navegação via ROS foi criado um sistema chamado `Nav2System`. Sua posição na arquitetura do simulador pode ser vista na Figura 3.1.

Este sistema implementa uma comunicação através de uma interface declarada no projeto **Nav2** [13]. O Nav2 é um projeto para ROS 2 que disponibiliza aos softwares ROS várias funcionalidades relacionadas à navegação de um robô. Ele já é um projeto

[3]processos que devem ser executados para tratar um comando que chegou via ROS. Por exemplo, se alguém enviou um novo objetivo em uma action ROS, o callback que lida com um novo objetivo chegando será executado.

bem estabelecido e padronizado na literatura robótica com ROS 2 e sua utilização tornará o HMR Sim mais acessível a outros projetos que já utilizam estes meios de comunicação.

O sistema `Nav2System`, além dos callbacks executados pelo ROS, também tem um método de processamento chamado periodicamente que enviará feedbacks sobre o estado dos robôs que estão em movimento. Além também de um método que observa a fila de eventos esperando pelo evento que indica a chegada do agente robótico ao destino.

Este sistema não movimenta os robôs dentro do HMR Sim, isto é feito pelo próprio sistema de navegação que já existia no projeto. O `Nav2System` traduz os comandos recebidos via ROS para a estrutura de navegação utilizada dentro do HMR Sim, além de enviar feedbacks com dados relevantes do robô em movimento.

Ele segue uma estrutura padrão criada dentro do simulador para adaptar Actions ROS, definida pelo tipo `RosActionServer`. A Figura 3.4 mostra como é a estrutura do `Nav2System`.

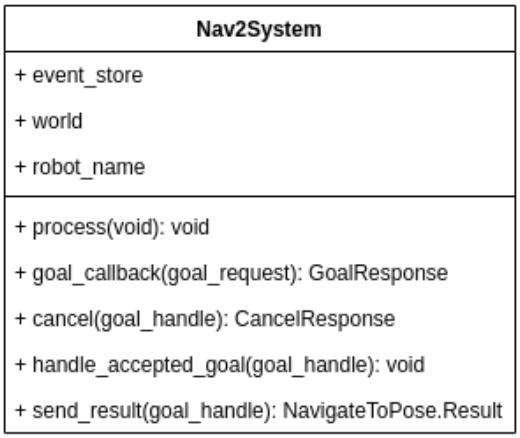

Figura 3.4: Estrutura da classe do Nav2System. Fonte: De autoria própria.

Os sistemas que implementam o tipo `RosActionServer` devem implementar os callbacks ROS. **Callback** é um método que poderá será chamado pela própria biblioteca ROS ao rodar o nó. Cada callback serve a um propósito e é chamado pelo ROS de acordo com ele.

A maioria dos callbacks executados pelo ROS, recebem como parâmetro um objeto `goal` que funciona como uma ferramenta para realizar tarefas como enviar feedbacks ou mudar o estado da execução da ação do robô para sucesso ou falha.

Abaixo seguem as explicações de como funcionam os callbacks e métodos principais que estruturas como esta devem seguir. O `Nav2System` será utilizado como exemplo para ajudar na explicação de cada método.

Método *process* Executado periodicamente junto à classe bridge de controle geral dos módulos ROS. No `Nav2System` ele envia os feedbacks via ROS dos robôs que estão em movimento, ou seja, se o robô não recebeu ainda um objetivo ele não irá enviar nenhum feedback.

Método *goal_callback* Executado assim que um novo objetivo é detectado para um agente robótico, mas ainda antes desse objetivo ser executado. Esse método verifica se já existe algum objetivo em andamento, se for o caso ele apaga o objetivo anterior e traça um novo, caso contrário ele aceita o novo objetivo.

Método *cancel* Callback executado quando uma requisição de cancelamento é feita via ROS. Para o caso do `Nav2System` se já existir um objetivo em andamento para o robô em questão, então esse objetivo é cancelado e é retornado que o cancelamento foi aceito, caso não tenha objetivo em andamento o cancelamento é rejeitado.

Método *handle_accepted_goal* Executado quando um novo objetivo é aceito. No `Nav2System` ele envia o evento de navegação para o robô que recebeu a missão via ROS.

Método *send_result* (ou *execute_callback*) É o método responsável por enviar via ROS o resultado da execução do objetivo de navegação. Ele é chamado quando o sistema `Nav2System` detecta que foi lançado o evento que indica o fim da navegação.

3.2.2 Manipulando Objetos com ROS

O HMR Sim foi ainda estendido com a capacidade de manipulação de objetos. Para tanto foi criada uma classe chamada `RosClawService` como sendo uma classe abstrata inicial para implementações de sistemas de controle de garra (ou manipulação de objetos) via ROS que herda da classe `RosActionServer`. Duas implementações desta classe foram feitas, `RosClawGrabService` e `RosClawDropService`.

Foram utilizadas as interfaces do *MoveIt* [14] de `Pickup` e `Place` para definir o tipo da comunicação com o HMR Sim via ROS para estas funcionalidades. o MoveIt é uma plataforma de manipulação robótica que além de navegação engloba também outras formas de controle como planejamento, manipulação e percepção 3D. [14]

3.2.3 Componente *NavToPoseRosGoal*

Foi verificada a necessidade dos robôs simulados carregarem um componente a mais que contivesse informações quanto ao objetivo que estivesse sendo executado pelo robô via ROS, então foi criado o `NavToPoseRosGoal` para mitigar esta necessidade.

Durante a inicialização da simulação, cada robô simulado agora será inicializado também com este componente e será identificado com um nome. O termo *componente* aqui refere-se ao componente de uma arquitetura ECS.

O componente `NavToPoseRosGoal` possui o nome do agente robótico e opcionalmente um objeto que referencia o objetivo que aquele robô poderá está executando no momento. Este objeto pertence à biblioteca Python do ROS e é utilizado para lidar com objetivos recebidos para o robô, com ele é possível por exemplo definir a missão com sucesso ou falha, enviar o resultado final da missão, enviar feedbacks. A maioria dos callbacks ROS recebem este objeto como parâmetro, chamado normalmente de `goal_handle`.

Para que este componente pudesse ser colocado nos robôs simulados, foi criado um novo model [4] no HMR Sim para o tipo robô, que será responsável por inicializar os agentes robóticos da simulação com os componentes necessários e em especial o `NavToPoseRosGoal`.

3.3 Controle dos Robôs Simulados

O controle dos agentes robóticos simulados que anteriormente podia ser realizado apenas através de scripts próprios acoplados ao simulador, agora também pode ser feito utilizando a infra-estrutura do ROS. Essa extensão nos permite desenvolver código de controle que pode ser testado inicialmente no HMR Sim e depois migrado diretamente para um simulador com diferentes características e para robôs reais.

Um software controlador pode ser implementado com diversos paradigmas e ferramentas, bastando apenas que se comunique com os robôs através das mesmas interfaces implementadas pelo HMR Sim.

Uma forma de controle é através de **Behaviour Trees**, o que permite um controle mais complexo dos robôs sendo simulados, havendo tomadas de decisões e revisão dos objetivos em tempo de execução por exemplo. Behaviour Trees já são muito utilizadas para descrever o comportamento de personagens não-jogáveis (NPC) em jogos e vem sendo cada vez mais utilizadas também para descrever e implementar o comportamento de robôs. [21]

Behaviour Trees utilizadas para controle dos robôs simulados fazem parte do software controlador dos robôs e o simulador não depende mais da estrutura deste software.

[4] é uma forma de padronizar atributos/componentes de entidades a depender de atributos colocados nelas no diagrama original que gera o ambiente, por exemplo, se no diagrama uma entidade tem um atributo de tipo com valor “robot”, o model de robô identifica este atributo e define os componentes padrões para este tipo de entidade.

3.3.1 Controlando os Robôs Simulados Através de Behaviour Trees em Python

Como Behaviour Trees estão sendo cada vez mais utilizadas na literatura robótica, o módulo controlador feito neste projeto para exercitar o HMR Sim foi feito utilizando Behaviour Trees. O controle dos robôs simulados agora é realizado fora do simulador, em um sistema externo. E a arquitetura do sistema que controla os robôs não depende mais da arquitetura do HMR Sim, desde que o sistema se comunique utilizando as mesmas interfaces ROS que o simulador utiliza.

A Listagem 3.1 ilustra um código Python de um método que envia um objetivo de navegação para um agente robótico. O nome da Action utilizada é definido exteriormente a este método dentro do atributo `self._action_client`.

```
def send_goal(self, x, y):
        self.get_logger().info('Waiting for action server...')
        self._action_client.wait_for_server()

        goal_msg = NavigateToPose.Goal()
        goal_msg.pose.pose.position.x = x
        goal_msg.pose.pose.position.y = y

        self.get_logger().info('Sending goal request...')

        self._send_goal_future = self._action_client.send_goal_async(
            goal_msg,
            feedback_callback=self.feedback_callback)

        self._send_goal_future.add_done_callback(self.goal_response_callback)
```

Listagem 3.1: Exemplo envio de objetivo de navegação por código Python. Fonte: De autoria própria.

No módulo controlador foram utilizadas as bibliotecas *Py Trees* [22] e *Py Trees for ROS* [23]. Py Trees é uma biblioteca que implementa os conceitos de Behaviour Trees vistos na Seção 2.4. Py Trees for ROS é uma biblioteca que implementa uma comunicação via ROS utilizando a biblioteca Py Trees.

A seguir, os pacotes em Python das bibliotecas citadas acima utilizados neste exemplo e uma breve explicação do que tem neles.

- **py_trees** Possui várias classes utilitárias como a `Behaviour` que define as propriedades e métodos básicos que todo comportamento dentro de uma behaviour tree deve ter, além de implementações desta classe que definem vários comportamentos simples.
- **py_trees_ros** Possui várias classes utilitárias que mesclam a utilização das bibliotecas de Py Trees e ROS. Uma classe presente nesse pacote por exemplo é a

`ActionClient` que define um `Behaviour` que verifica através de ticks a execução de uma Action ROS.

Na biblioteca py_trees_ros o cliente de uma Action ROS é um nó folha de uma behaviour tree.

Para ilustrar a integração entre Behavior Trees e uma simulação no HMR Sim, apresentamos um exemplo simples, a Figura 3.5 mostra uma behaviour tree onde o objetivo do robô é ir até uma posição. Essa Behavior Tree possui uma action de navegação. Com a integração implementada nesse trabalho essa action pode ser executada no ambiente simulado conforme ilustrado a seguir. A Figura 3.6 mostra um diagrama de sequência ilustrando o processo em que o nó action da behaviour tree de exemplo envia um comando de navegação pelo ROS e a chegada dele no HMR Sim. Os diferentes softwares executando estão separados por cores. Na figura, a behaviour tree envia o comando de acordo com uma interface ROS `nav2_msgs/action/NavigateToPose`. Depois que o nó envia o objetivo, o ROS chama um callback de um sistema dentro do simulador informando o objetivo. Então o sistema coloca o evento correspondente àquele objetivo na fila de eventos.

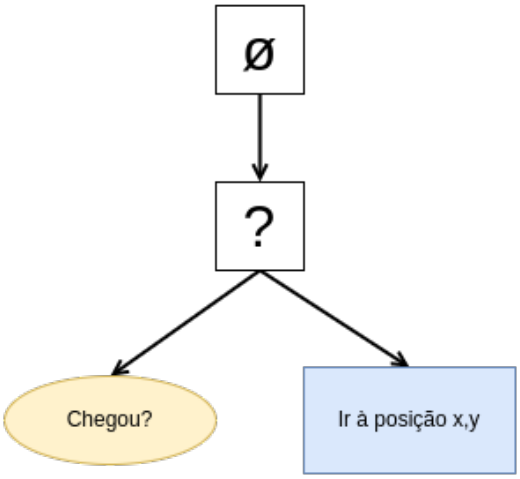

Figura 3.5: Exemplo de behaviour tree para um objeto ir de uma posição a outra. Fonte: De autoria própria.

Com a nova integração, o controle de robôs agora independente da arquitetura do simulador, podendo ser realizado por software com diferentes arquiteturas e utilizando diversas ferramentas diferentes, como a utilização de Behaviour Trees mostrada nesta seção.

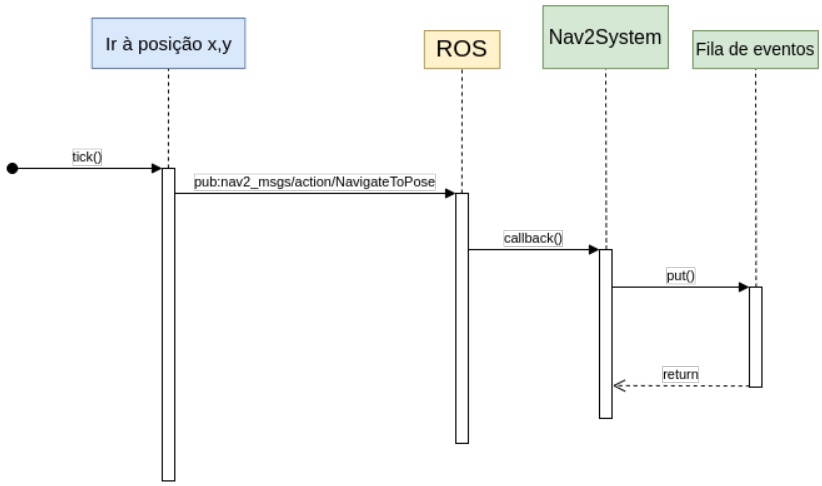

Figura 3.6: Diagrama de sequência mostrando o processo do nó da behaviour tree enviar um comando de navegação para o HMR Sim. Fonte: De autoria própria.

Capítulo 4

Experimentação

Neste capítulo serão mostrados dois cenários de uso do HMR Sim utilizando a interface ROS para controle dos robôs simulados. O principal objetivo das execuções a seguir é validar a utilização da camada de comunicação via ROS. Esta validação será feita a partir da execução de cenários para avaliação dos módulos criados, verificando o desempenho do simulador depois das novas implementações.

O primeiro cenário, mostrado na Seção 4.1, com um robô com garra e um objeto a ser apanhado, mostra a execução de um exemplo com chamadas de actions ROS para controle do robô simulado.

O segundo cenário, mostrado da Seção 4.2 mostra a execução utilizando um código controlador em Python que utiliza os conceitos de *Behaviour Trees* explicados na Seção 2.4. Este cenário busca validar a possibilidade do controle dos robôs dentro do simulador através do ROS, verificando se o simulador envia ao controlador feedbacks do andamento da execução dos objetivos, se envia o resultado final da navegação e se o desempenho continua aceitável se comparado com a maneira como a execução era feita anteriormente.

Os dois cenários utilizam duas formas diferentes de controle dos robôs simulados e o código do simulador possui um tutorial explicando como os robôs podem ser controlados. O tutorial pode ser encontrado no projeto do simulador no GitHub.

Todas as simulações presentes neste capítulo foram realizadas em ambiente com sistema operacional *Linux Mint 20.3 Cinnamon*, processador Intel Core i7-10510U e 16GB de memória RAM. A versão do Python utilizada foi a 3.8.10.

4.1 Cenário 1 - Chamadas via ROS

O principal objetivo deste cenário é verificar se o robô simulado dentro do HMR Sim pode responder aos comandos de navegação e comandos relacionados à garra enviados via ROS.

O mapa do cenário utilizado pode ser visto na Figura 4.1. A figura ilustra o cenário no início de sua execução. As setas pretas são os caminhos possíveis para o robô. O robô está em amarelo no cenário e seu nome é wall_e. O objeto em vermelho é um que pode ser apanhado.

O objetivo do robô neste experimento é basicamente ir até o objeto vermelho, apanhá-lo, levar o objeto até outro lugar do mapa e largar o objeto lá. O que será verificado é se esse processo poderá ser controlado via ROS e não mais por comandos de dentro do próprio simulador.

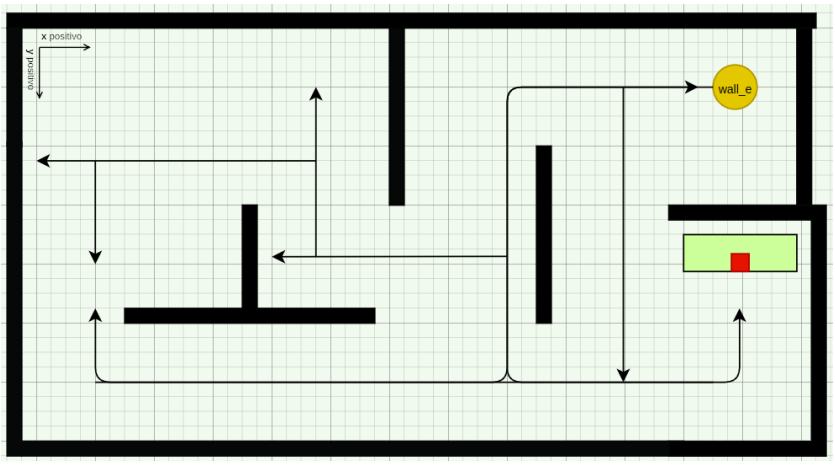

Figura 4.1: Exemplo de ambiente de execução do simulador. No canto esquerdo superior tem uma legenda mostrando o sentido positivo dos eixos X e Y. Fonte: De autoria própria.

A execução deste cenário será composta da seguinte forma:

1. Comando de navegação para perto da posição do objeto a ser apanhado (objeto vermelho na Figura 4.1), comando feito para a posição $x = 512, y = 215$.
2. Comando para o agente robótico apanhar o objeto.
3. Comando para o robô ir para outra posição qualquer, feito para a posição $x = 100, y = 180$.
4. Comando para o robô largar o objeto apanhado anteriormente.

Após a inicialização da simulação, é enviado o comando mostrado na Listagem 4.1 que envia o agente robótico para uma posição perto do objeto a ser apanhado.

```
ros2 action send_goal navigate_to_pose/wall_e nav2_msgs/action/NavigateToPose "{pose: {
    pose: { position: {x: 512, y: 215, z: 1} } } }" --feedback
```

Listagem 4.1: Primeiro comando de navegação enviado ao robô.Fonte: De autoria própria.

A Figura 4.2 mostra os terminais depois da execução do comando de navegação. O que vale notar aqui é que são dois terminais separados, ou seja, dois processos separados, o terminal à direita é onde os comandos via ROS são enviados e o à esquerda onde o HMR Sim está sendo executado. No terminal à direita pode-se ver que foram recebidos **feedbacks** das posições do robô à medida em que ele foi se movimentando, na figura aparece o último feedback enviado.

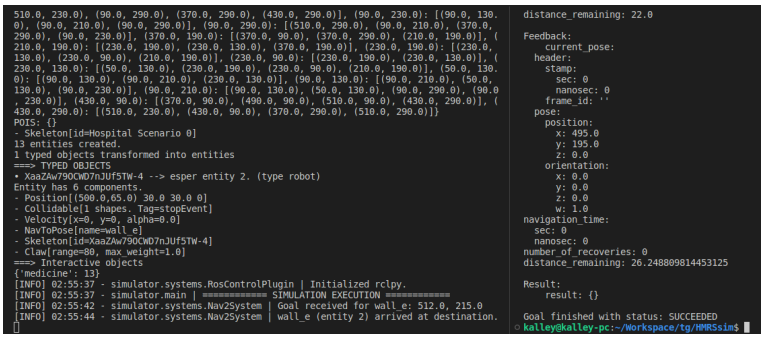

Figura 4.2: Terminal depois da execução do primeiro comando de navegação. Fonte: De autoria própria.

A Listagem 4.2 mostra o comando de *pick up* enviado depois do robô chegar em uma posição perto do objeto a ser apanhado. No comando enviado, pode-se perceber que é enviado o nome do objeto a ser apanhado, no caso, **medicine**.

```
ros2 action send_goal wall_e/grab moveit_msgs/action/Pickup "{ target_name: 'medicine' }"
    --feedback
```

Listagem 4.2: Comando de *pick up* enviado ao robô. Fonte: De autoria própria.

Agora será enviado outro comando de navegação como mostrado na Listagem 4.3 que enviará o robô para uma outra posição qualquer do ambiente de simulação.

```
ros2 action send_goal navigate_to_pose/wall_e nav2_msgs/action/NavigateToPose "{pose: {
    pose: { position: {x: 100, y: 180, z: 1} } } }" --feedback
```

Listagem 4.3: Segundo comando de navegação enviado ao robô. Fonte: De autoria própria.

Por fim será enviado um comando para o robô largar o objeto apanhado na posição em que ele está. É o comando mostrado na Listagem 4.4.

```
ros2 action send_goal wall_e/drop moveit_msgs/action/Place "{attached_object_name: '
    medicine' }" --feedback
```

Listagem 4.4: Comando enviado para o robô largar o objeto apanhado anteriormente. Fonte: De autoria própria.

A Figura 4.3 mostra como foi no navegador o processo descrito acima. No canto superior esquerdo tem-se a posição do robô depois no primeiro comando de navegação, mostrado na Listagem 4.1. Canto superior direito o robô pega o objeto vermelho com o

comando da Listagem 4.2, isto é demonstrado através da área onde estava o objeto, que fica acinzentada. No canto inferior direito o robô navega novamente para outra posição, mais a esquerda, por causa do comando da Listagem 4.3. Por fim, o objeto é largado como pode ser visto no canto inferior esquerdo, depois do comando da Listagem 4.4.

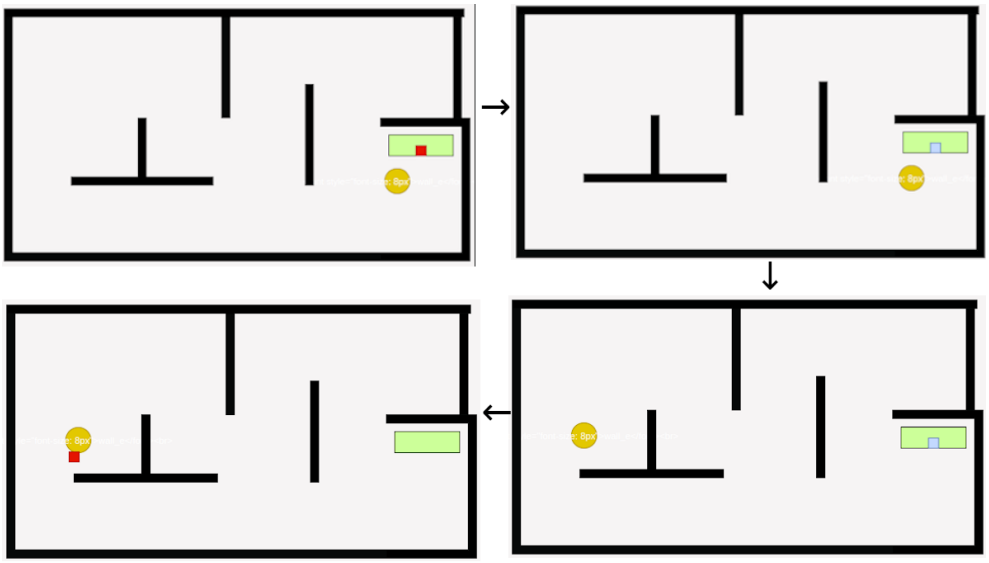

Figura 4.3: Processo realizado para o cenário 1. As setas indicam a evolução das fases da execução. Fonte: De autoria própria.

A Figura 4.4 mostra a situação final do terminal depois da execução de todo o processo relatado acima. Observe que para os comandos de navegação o HMR Sim imprime um aviso de que recebeu o comando e outro avisando que robô chegou ao destino, informando o nome do agente robótico e a posição de destino. Observe também que para os comandos de *pick up* e *drop* o simulador imprime um aviso com o nome do robô que recebeu o comando e a ação a ser realizada. Estas impressões mostram o recebimento correto das informações contidas nos comandos enviados.

```
[INFO] 03:06:11 - simulator.main | ============ SIMULATION EXECUTION ============
[INFO] 03:06:27 - simulator.systems.Nav2System | Goal received for wall_e: 512.0, 215.0
[INFO] 03:06:29 - simulator.systems.Nav2System | wall_e (entity 2) arrived at destination.
[INFO] 03:06:49 - simulator.systems.ClawDESProcessor | Received order for wall_e to grab medicine
[INFO] 03:07:06 - simulator.systems.Nav2System | Goal received for wall_e: 100.0, 180.0
[INFO] 03:07:09 - simulator.systems.Nav2System | wall_e (entity 2) arrived at destination.
[INFO] 03:09:22 - simulator.systems.ClawDESProcessor | Received order for wall_e to drop medicine
```

Figura 4.4: Terminal depois da execução do exemplo. Fonte: De autoria própria.

Depois da execução acima demonstrada, podemos concluir que o HMR Sim agora é sim capaz de responder corretamente a comandos enviados via ROS, em particular, aos comandos de navegação do robô e comandos de apanhar e largar um objeto.

O cenário 1 foi operacionalizado enviando comandos ROS pelo terminal. No entanto o controle do robô poderia ter sido realizado utilizando um componente de controle imple-

mentado usando as APIs ROS para as diferentes linguagens de programação suportadas. O controle do robô poderia ser feito ainda utilizando abstrações de controle, tais como Finite State Machines e Behavior Trees, utilizando pacotes de terceiros do repositório do ROS. O cenário 2 demostra o uso do HMR Sim no exercício de uma Behavior Tree.

4.2 Cenário 2 - Behaviour Tree

Neste cenário é utilizado o mapa que pode ser visto na Figura 4.5, que possui um robô com nome *wall_e*. As setas indicam os caminhos possíveis para o robô. O mapa mostra o cenário no início da execução dele. O objetivo do robô é chegar à posição $x = 450, y = 330$. Aqui, busca-se avaliar o correto recebimento e envio de mensagens via ROS do simulador e o tamanho do impacto em desempenho da implementação da camada ROS.

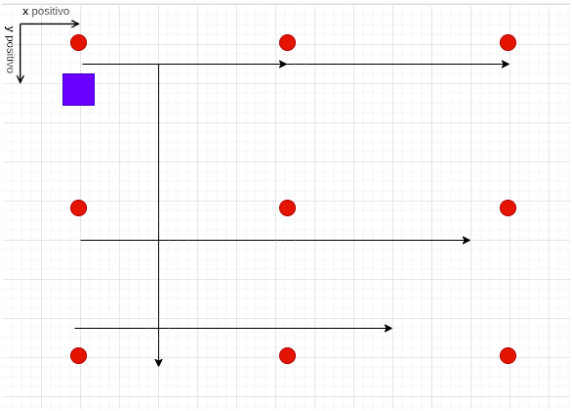

Figura 4.5: Mapa de execução do segundo cenário. No canto esquerdo superior tem uma legenda mostrando o sentido positivo dos eixos X e Y. Fonte: De autoria própria.

4.2.1 Avaliação Funcional

Nesta avaliação, será verificado o envio de comandos via ROS do simulador por um software controlador que utiliza Behaviour Trees.

O simulador deve receber um comando de controle de navegação via ROS, retornando o aceite ao controlador, que deve continuar a dar ticks na behaviour tree, recebendo periodicamente do simulador feedbacks com as distâncias remanescentes para o destino. Ao chegar no destino, o simulador deve retornar ao controlador que o robô chegou.

A Figura 4.6 mostra o mapa no navegador já durante a execução do simulador. A posição $x = 450, y = 330$ fica próxima ao X marcado na figura.

A Listagem 4.5 mostra o código da behaviour tree utilizada para execução deste cenário. A estrutura da árvore é bem simples, sendo a parte principal os nós folha onde

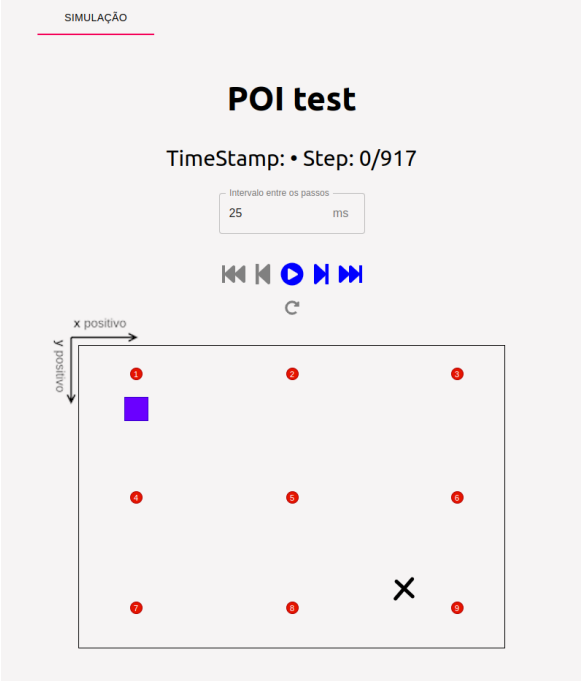

Figura 4.6: Mapa da execução deste cenário no navegador já com o simulador executando. O X mostra a posição para onde o robô deve ir. Na parte esquerda superior do mapa tem uma legenda mostrando o sentido positivo dos eixos X e Y. Fonte: De autoria própria.

será verificado se o robô já chegou ao destino (variável `arrived` no código, que é um nó da behaviour tree) e, caso negativo, será executado um *tick* no nó cliente da action de navegação (variável `goto_action` no código). Ela pode ser vista na Figura 4.7. A árvore verifica a existência da action ROS para controle do robô wall_e e depois seguirá com a execução.

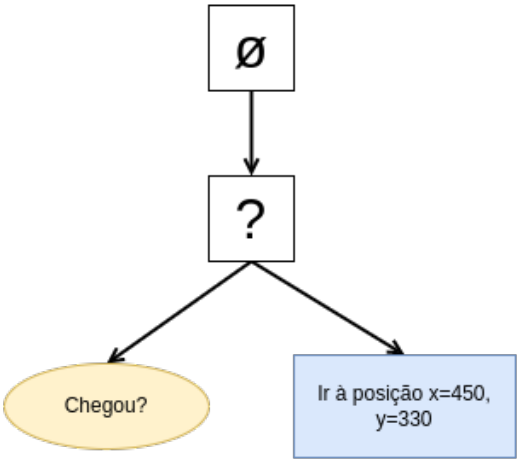

Figura 4.7: Behaviour tree utilizada para a execução da avaliação. Fonte: De autoria própria.

```
def create_root() -> py_trees.behaviour.Behaviour:
    root = py_trees.composites.Sequence("GoToPoseSequence")
    goto_fallback = Fallback("GoToPoseFallback")
    arrivedBB = py_trees.blackboard.Client(name="Arrived")
    arrivedBB.register_key(key="arrived", access=py_trees.common.Access.WRITE)
    arrivedBB.arrived = py_trees.common.Status.RUNNING
    arrived = py_trees.behaviours.CheckBlackboardVariableValue(
        name="CheckArrived",
        check=py_trees.common.ComparisonExpression(
            variable="arrived",
            value=py_trees.common.Status.SUCCESS,
            operator=operator.eq
        )
    )
    goal_msg = NavigateToPose.Goal()
    goal_msg.pose.pose.position.x = 150.0
    goal_msg.pose.pose.position.y = 50.0
    goto_action = NavToPoseActionClient(
        name="GoToPoseAction",
        robot_name="wall_e",
        goal=goal_msg,
        generate_feedback_message=lambda msg: "remaining: {0}".format(msg.feedback.
    distance_remaining)
    )
    root.add_child(goto_fallback)
    goto_fallback.add_children([arrived, goto_action])
    return root
```

Listagem 4.5: Método de criação da *behaviour tree* a ser utilizada neste cenário. Fonte: De autoria própria.

A execução do código é feita e a Figura 4.8 mostra o processo de execução da árvore, são impressões no terminal que buscam ilustrar as diferentes fases da execução da behaviour tree sendo executada. Segue uma breve explicação dos estágios mostrados na figura:

1. **Enviando Requisição** Aqui é verificado se o robô já chegou no destino através de uma variável booleana, o que é negativo e então é dado um tick no nó da action que envia uma requisição de novo objetivo do robô, já que é a primeira vez que está sendo executado.

2. **Feedbacks** Aqui são recebidos os feedbacks das posições do robô à medida em que ele vai se movimentando. Foram vários feedbacks, mas foi colocado só um deles aqui para exemplo.

3. **Chegada do Robô** A action ROS retorna sucesso para o nó cliente que também retorna sucesso na árvore, mudando o estado da variável de chegada para verdadeiro.

4. **Validação de Chegada** É checado novamente a variável booleana de chegada e validado que o robô chegou ao destino, finalizando o processamento da árvore.

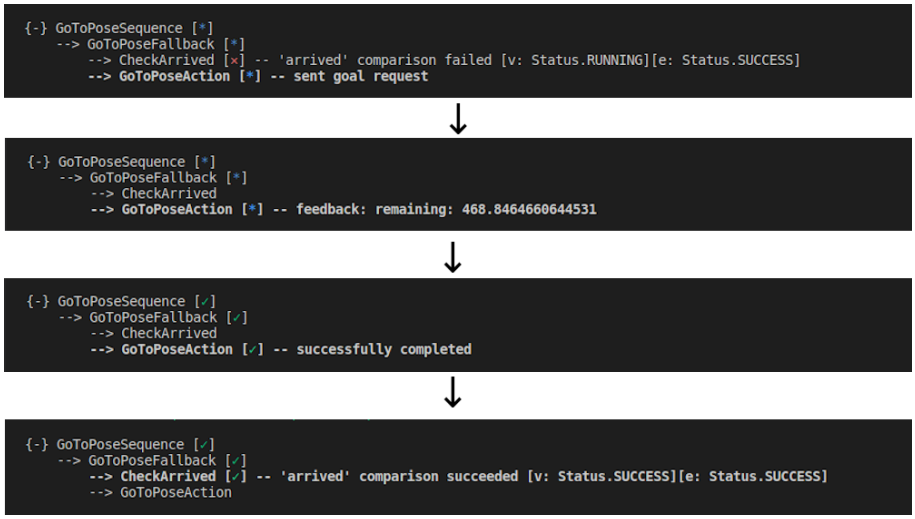

Figura 4.8: Processo de navegação no terminal do processo utilizando Py Trees, foram colocadas na imagem apenas as partes mais importantes e a parte de feedback foi colocado apenas um dos feedbacks dados para servir como referência. Fonte: De autoria própria.

A Figura 4.9 mostra a posição final do robô no mapa depois da execução deste exemplo e também o caminho percorrido por ele durante a execução.

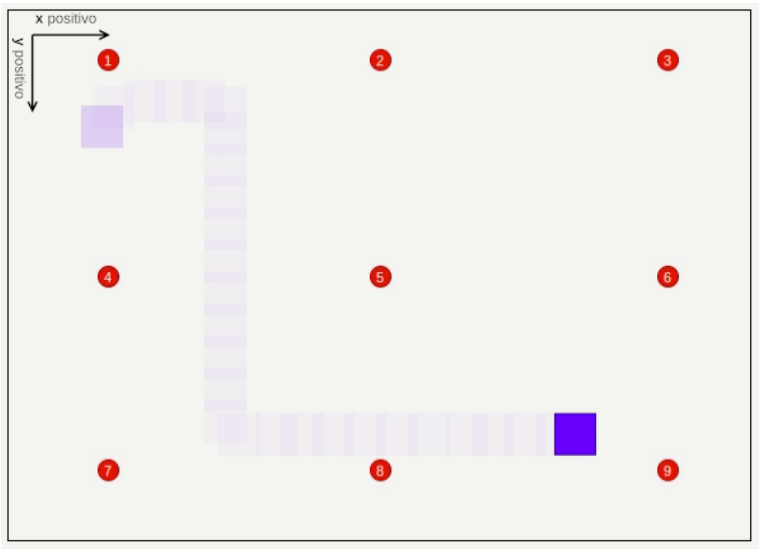

Figura 4.9: Mapa depois da execução do cenário, mostrando o percorrido pelo robô durante a execução. No canto esquerdo superior tem uma legenda mostrando o sentido positivo dos eixos X e Y. Fonte: De autoria própria.

Algumas observações depois da execução deste cenário:

- O HMR Sim foi capaz de receber corretamente os comandos enviados via ROS.
- O HMR Sim retornou os feedbacks ao software controlador.

- A behaviour tree executou como esperado, enviando e recebendo as mensagens esperadas do HMR Sim.

4.2.2 Avaliação de Desempenho

Aqui será verificado o impacto em desempenho da implementação da nova camada no HMR Sim se comparado ao desempenho sem esta camada. O número de execuções por segundo do loop principal da simulação foi aumentado para permitir a realização de mais execuções. O módulo de visualização da simulação é desligado para verificar apenas a execução em si da navegação do robô. A posição de destino do robô também é $x = 450, y = 330$.

Foram feitas avaliações de velocidade de execução da navegação do robô, tanto com a camada nova utilizando ROS quanto utilizando a forma anterior de controle através de comandos sequenciais dentro do próprio simulador. A Tabela 4.1 mostra as execuções realizadas para esta avaliação utilizando a camada de comunicação via ROS e a Tabela 4.2 mostra as execuções sem utilizar a camada.

Tempo inicial (s)	**Tempo final (s)**	**Tempo decorrido (s)**
1676383564,9850473	1676383565,0034568	0,0184095
1676383627,0930855	1676383627,1146288	0,0215433
1676383654,3721004	1676383654,4059336	0,0338332
1676383676,3627758	1676383676,3805770	0,0178012
1676383710,7113256	1676383710,7246878	0,0133622
Média (s)		$0,02099 \pm 0,0031 (\pm 14.77\%)$

Tabela 4.1: Execuções realizadas utilizando a camada de comunicação via ROS. Margem de erro baseada em desvio padrão utilizando nível de confiança de um desvio padrão.

Tempo inicial (s)	**Tempo final (s)**	**Tempo decorrido (s)**
1676384157,3860784	1676384157,3961850	0,0101066
1676384172,7829692	1676384172,7919445	0,0089753
1676384205,2460048	1676384205,2553325	0,0093277
1676384219,1667752	1676384219,1761960	0,0094208
1676384233,5774026	1676384233,5879630	0,0105604
Média (s)		$0,009678 \pm 0,000257 (\pm 2,65\%)$

Tabela 4.2: Execuções realizadas sem utilizar a camada de comunicação via ROS. Margem de erro baseada em desvio padrão utilizando nível de confiança de um desvio padrão.

A média do tempo de execução da navegação nestas condições para a avaliação da nova camada ROS foi de $\approx 20,99$ mili-segundos, enquanto que para a execução com a forma anterior de controle foi de $\approx 9,68$ mili-segundos. Isso significa que agora com a

implementação da camada de comunicação via ROS, a navegação de um robô dentro do simulador teve um overhead significativo. No entanto, observa-se que o tempo ainda é bastante aceitável na prática. Tal impacto no desempenho pode-se justificar pelo fato de que a implementação da camada de comunicação via ROS implica em uma maior complexidade do simulador, tendo uma maior quantidade de sistemas e eventos sendo processados.

A memória RAM total utilizada durante a execução do cenário foi, em média, de $35,3$ MB, esse valor foi o mostrado sendo utilizado pelo processo do HMR Sim antes e também depois da execução do comando de navegação em diferentes execuções. Este valor para uso de memória RAM pode ser considerado aceitável uma vez que outros simuladores com maior nível de detalhamento físico podem exigir memória na ordem de gigas, como o Gazebo [25].

4.2.3 Conclusão

Funcionalmente a integração permite o controle do agentes robóticos simulados utilizando ROS, inclusive utilizando pacotes de terceiros como PyTrees. Mas esta integração acarreta um certo overhead de desempenho que deve ser tomado em consideração pelo usuário final do simulador em um cenário concreto.

4.3 Limitações da Execução

Um grande desafio encontrado durante a implementação do projeto foi o problema com o callback de execução de uma action ROS. A Listagem 4.6 mostra um exemplo clássico de como é a estrutura deste callback em Python [26]. Este exemplo é o cálculo da sequência de Fibonacci, em que o cálculo de cada número da sequência é assumido como sendo um passo da execução da action e para cada passo é enviado um feedback ao cliente sobre o estado atual da execução. Os passos ficam dentro de um loop `for`. A princípio não nota-se um problema, mas existe um conflito entre como este exemplo (e outros online) funciona e como funciona a execução de um processo dentro do HMR Sim.

```
def execute_callback(self, goal_handle):
        self.get_logger().info('Executing goal...')

        feedback_msg = Fibonacci.Feedback()
        feedback_msg.partial_sequence = [0, 1]

        for i in range(1, goal_handle.request.order):
            feedback_msg.partial_sequence.append(
                feedback_msg.partial_sequence[i] + feedback_msg.partial_sequence[i-1])
            self.get_logger().info('Feedback: {0}'.format(feedback_msg.partial_sequence))
            goal_handle.publish_feedback(feedback_msg)
            time.sleep(1)

        goal_handle.succeed()

        result = Fibonacci.Result()
        result.sequence = feedback_msg.partial_sequence
        return result
```

Listagem 4.6: Exemplo de execute_callback. Fonte: De autoria própria.

O ROS por padrão assume que a execução de uma action se dará dentro de um único método, retornando o resultado no final dele, tendo uma fila de métodos a serem executados e permitindo a execução em paralelo deles se necessário [27]. Podemos observar isso nos exemplos clássicos encontrados online para implementação deste callback, normalmente chamado `execute_callback`. No exemplo mostrado, toda a execução da action é dentro deste callback incluindo o envio de feedbacks ao cliente.

Mas dentro do HMR Sim isso gera um problema, pois ele utiliza um loop de eventos onde a execução da simulação é dividida em passos, como acontece em um jogo por exemplo. Isso implica que a execução de uma action também deveria ser dividida em passos. Mas na forma como é padrão na biblioteca ROS, a execução do callback não é dividida em passos, o que significa que executar a action como nos exemplos encontrados online dentro do simulador é bloquear a execução da simulação enquanto a action não termina de executar.

Solução

Foram encontradas algumas soluções possíveis para a resolução deste problema. Uma delas seria utilizar alguma alternativa ROS que utiliza threads para executar as actions em threads separadas da simulação. Acontece que esta solução entra em conflito com a proposta de funcionamento do simulador, que utiliza um loop que divide a simulação em passos sendo uma boa alternativa justamente para não precisar da execução de múltiplas threads. A solução utilizada para este trabalho foi executar o execute_callback apenas no momento em que for enviar o resultado ao cliente, enviando os feedbacks fora deste método.

Capítulo 5

Conclusão e Trabalhos Futuros

O principal objetivo deste trabalho é fazer com que o HMR Sim pudesse ser utilizado como simulador para softwares de controle de robôs capazes de controlar robôs reais. Isto é realizado através de uma nova camada de comunicação feita no simulador.

Para uma melhor implementação deste objetivo, é melhor que sejam utilizadas ferramentas já bem conhecidas na literatura, como é o caso do ROS. A utilização do ROS como um middleware entre HMR Sim e outro software controlador de robôs facilita a transição do controlador para controlar robôs reais.

A implementação da camada de comunicação dentro do HMR Sim é realizada principalmente através da classe `RosControlBridge` descrito na Subseção 3.2.1. A implementação desta classe conseguiu embarcar controle de navegação e de garra dos robôs simulados. Agora um processo que pretenda enviar comandos de navegação ou para a garra dos robôs simulados pode fazê-lo através de comandos via ROS se preocupando apenas com a interface que a comunicação deverá ser feita.

A camada de comunicação do HMR Sim permite que o software controlador tenha liberdade para utilizar a arquitetura que preferir. E uma boa forma de controlar robôs é utilizando *Behavior Trees*, que é uma forma de controlar uma entidade utilizando árvores, um conceito muito utilizado em jogos mas cada vez mais sendo utilizado na literatura robótica também. Por este motivo foi citada a contribuição de permitir o controle de robôs utilizando Behavior Trees.

Esta contribuição é implementada através do código Python fora do simulador criado para testá-lo, descrito na Seção 4.2. As implementações para exercício do HMR Sim foram feitas em repositório próprio e podem ser acessadas uma no arquivo hmrsim_tester.py e outra no arquivo go_to_pose_tree.py, sendo que esta segunda implementação é a que utiliza a biblioteca `py-trees-ros` [23].

Além das implementações para os objetivos citados, houve também refatorações realizadas no projeto que permitiram subir a aplicação em container docker com comandos

mais simples.

Para trabalhos futuros, algumas possíveis implementações para o HMR Sim são:

- Melhorias no sistema interno de navegação do HMR Sim, atualmente o próprio simulador planeja a rota dos agentes robóticos simulados. Mas sendo só um simulador de ambiente, ele deveria apenas seguir caminhos simples em linha reta e um sistema externo ficaria responsável pelo planejamento da rota.
- Implementar um handshake inicial do HMR Sim com o sistema controlador externo onde os dois sincronizam os caminhos possíveis para os robôs.
- Testar o simulador com ambientes mais complexos com mais agentes robóticos e diferentes tarefas para cada robô sendo executadas ao mesmo tempo.
- Implementação de retorno de erros via ROS para casos em que o robô não consegue finalizar a navegação.
- Implementar sistema na interface ROS para publicação de dados de consumo de bateria dos robôs.
- Realizar testes do HMR Sim com sistemas de controle que também controlam robôs reais.

Muitos dos objetivos listados acima não eram possíveis antes deste projeto pois o controle dos robôs dentro do simulador ainda era muito acoplado ao próprio simulador. A camada de comunicação com sistemas externos via ROS torna possível agora estes objetivos. O HMR Sim agora está em uma fase onde implementações de novas funcionalidades não serão mais prejudicadas pela falta de comunicação do simulador com outros sistemas.

Referências

[1] Guidini, Giovanni e Cristiane Cardoso: *Desenvolvimento e teste da ferramenta hmr sim: Um simulador de ambientes multi-robôs*. Universidade de Brasília, 1(1), 2021. (acessado em 14/02/2023). xi, 1, 4, 5, 6, 15

[2] Open Robotics: *Ros 2 documentation*, 2023. https://docs.ros.org/en/foxy/index.html, (acessado em 15/02/2022). xi, 8, 9, 10, 14

[3] Open Robotics: *Gazebo*, 2023. http://gazebosim.org/, (acessado em 27/01/2022). 1, 10

[4] Cyberbotics Ltd: *Webots*, 2023. https://cyberbotics.com/, (acessado em 15/02/2022). 1, 11

[5] Open Robotics: *Ros - robot operating system*, 2021. https://ros.org/, (acessado em 15/02/2023). 2

[6] Open Robotics: *Why ros?*, 2021. https://www.ros.org/blog/why-ros/, (acessado em 15/02/2023). 2

[7] Benjamin Moran: *Esper is a lightweight entity system module for python, with a focus on performance*, 2022. https://github.com/benmoran56/esper, (acessado em 29/01/2023). 5

[8] Open Robotics: *Nodes*, 2018. https://wiki.ros.org/Nodes, (acessado em 20/12/2021). 8

[9] Open Robotics: *Topics*, 2019. https://wiki.ros.org/Topics, (acessado em 20/12/2021). 9

[10] Open Robotics: *Services*, 2019. https://wiki.ros.org/Services, (acessado em 17/05/2022). 9

[11] Open Robotics: *Actions*, 2023. https://docs.ros.org/en/foxy/Tutorials/Understanding-ROS2-Actions.html, (acessado em 17/05/2022). 10

[12] Open Robotics: *Related projects*, 2023. https://docs.ros.org/en/foxy/Related-Projects.html, (acessado em 17/05/2022). 10

[13] ROS Planning: *Nav2*, 2020. https://navigation.ros.org/, (acessado em 17/05/2022). 10, 15, 17

[14] PickNik Robotics: *Moveit 2 tutorials*, 2023. https://moveit.picknik.ai/foxy/index.html, (acessado em 27/01/2023). 10, 19

[15] Open Source Robotics Foundation: *Ros 2 integration overview*, 2014. https://classic.gazebosim.org/tutorials?tut=ros2_overview, (acessado em 24/02/2023). 10

[16] Open Source Robotics Foundation: *Ros 2 migration: gazebo_ros_api_plugin*, 2019. https://github.com/ros-simulation/gazebo_ros_pkgs/wiki/ROS-2-Migration:-gazebo_ros_api_plugin, (acessado em 24/02/2023). 10

[17] LAAS-CNRS: *The morse simulator documentation*, 2016. https://www.openrobots.org/morse/doc/stable/morse.html, (acessado em 24/02/2023). 11

[18] LAAS-CNRS: *Integrate morse in your software architecture*, 2016. https://www.openrobots.org/morse/doc/stable/user/integration.html, (acessado em 24/02/2023). 11

[19] Cyberbotics Ltd.: *Home*, 2023. https://github.com/cyberbotics/webots_ros2/wiki, (acessado em 24/02/2023). 11

[20] Araújo, Gabriel: *Simulation and execution of dynamic behavior trees in the service robots context*. Universidade de Brasília, 1(1), 2021. 11, 12

[21] Colledanchise, Michele e Lorenzo Natale: *On the implementation of behavior trees in robotics*. IEEE Robotics and Automation Letters, 6(3):5929–5936, 2021. (acessado em 28/01/2023). 12, 20

[22] Splintered Reality: *Py trees*, 2023. https://py-trees.readthedocs.io/en/release-2.2.x/, (acessado em 05/02/2023). 12, 21

[23] Splintered Reality: *Py trees for ros*, 2021. https://py-trees-ros.readthedocs.io/en/release-2.1.x/, (acessado em 05/02/2023). 12, 21, 35

[24] Gamma, Erich, Richard Helm, Ralph Johnson e John M. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, páginas 151–171. Addison-Wesley Professional, 1ª edição, 1994, ISBN 0201633612. http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1, (acessado em 16/02/2023). 16

[25] The MathWorks, Inc.: *Gazebo simulation environment requirements and limitations*, 2023. https://www.mathworks.com/help/robotics/ug/gazebo-simulation-requirements.html, (acessado em 16/02/2023). 33

[26] Open Robotics: *Writing an action server and client (python)*, 2023. https://docs.ros.org/en/foxy/Tutorials/Intermediate/Writing-an-Action-Server-Client/Py.html, (acessado em 28/01/2023). 33

[27] Nicolò Valigi: *Concurrency and parallelism in ros 1 and ros 2: application apis*, 2019. https://nicolovaligi.com/articles/concurrency-and-parallelism-in-ros1-and-ros2-application-apis/, (acessado em 28/01/2023). 34