CENTRO UNIVERSITÁRIO FEI

MURILO DARCE BORGES SILVA
RODRIGO SIMÕES RUY

IDENTIFICAÇÃO DE DIFERENÇAS DE DESEMPENHO ENTRE SISTEMAS ROBÓTICOS SIMULADOS COM E SEM SOFTWARE CONTEINERIZADO, UTILIZANDO SIMULADOR GAZEBO, ROS 2 E DOCKER.

São Bernardo do Campo
2025

MURILO DARCE BORGES SILVA
RODRIGO SIMÕES RUY

IDENTIFICAÇÃO DE DIFERENÇAS DE DESEMPENHO ENTRE SISTEMAS ROBÓTICOS SIMULADOS COM E SEM SOFTWARE CONTEINERIZADO, UTILIZANDO SIMULADOR GAZEBO, ROS 2 E DOCKER.

Trabalho de Conclusão de Curso apresentado ao Centro Universitário FEI, como parte dos requisitos necessários para obtenção do título de Bacharel em Ciência da Computação. Orientado pelo Prof. Dr. Leonardo Anjoletto Ferreira.

São Bernardo do Campo
2025

MURILO DARCE BORGES SILVA
RODRIGO SIMÕES RUY

IDENTIFICAÇÃO DE DIFERENÇAS DE DESEMPENHO ENTRE SISTEMAS ROBÓTICOS SIMULADOS COM E SEM SOFTWARE CONTEINERIZADO, UTILIZANDO SIMULADOR GAZEBO, ROS 2 E DOCKER.

Trabalho de Conclusão de Curso apresentado ao Centro Universitário FEI, como parte dos requisitos necessários para obtenção do título de Bacharel em Ciência da Computação.

Comissão julgadora

Prof. Dr. Leonardo Anjoletto Ferreira

Prof. Dr. Plinio Thomaz Aquino Junior

Prof. Dr. Fagner de Assis Moura Pimentel

São Bernardo do Campo
01/12/2025

AGRADECIMENTOS

Agradecemos ao professor Fagner Pimentel pelo modelo para simulação do labirinto que existe na K4-04, que foi de suma importância para o desenvolvimento do projeto.

RESUMO

Containerização é uma ferramenta muito útil quando se lida com projetos que precisam de diferentes dependências ou programas que podem ter conflitos entre si, que precisam de uma grande quantidade de configuração inicial, ou que precisam de portabilidade. Este projeto visa identificar e mostrar a diferença de desempenho entre robôs, que utilizam o Robot Operating System 2 (ROS 2), simulados com e sem conteinerização. Para verificar esta diferença, os testes foram realizados no ambiente simulado do software Gazebo Classic. Os resultados mostram que o desempenho da simulação não é impactado pelo uso do Docker, que ocorre um pequeno aumento do uso de memória, porém sem variações significativas no uso do processador, indicando que a simulação usando containers é viável durante o processo de desenvolvimento.

Palavras-chave: Robô, Robot Operating System 2 (ROS 2), Gazebo Classic, Docker, Containerização.

ABSTRACT

Containerization is a very useful tool when dealing with projects that require different dependencies or programs that may conflict with each other, that need a large amount of initial configuration, or that need portability. This project aims to identify and show the performance difference between robots, using Robot Operating System 2 (ROS 2), simulated with and without containerization. To verify this difference, tests will be performed in the simulated environment of the Gazebo Classic software. The results show that the simulation performance is not impacted by the use of Docker, that there is a small increase in memory usage, but without significant variations in processor usage, indicating that simulation using containers is viable during the development process.

Keywords: Robot, Robot Operating System 2 (ROS 2), Gazebo Classic, Docker, Containerization.

SUMÁRIO

1 INTRODUÇÃO

A conteinerização é uma ferramenta poderosa no campo de desenvolvimento e implementação, disponibilizando uma camada de isolamento entre componentes de um projeto, o que assegura que eles não entrarão em conflito, seja por funções internas ou devido a dependências de versões diferentes. Na robótica, conteinerização é vista como uma técnica para facilitar o desenvolvimento, portabilidade e consistência em projetos de robótica, mas há um problema com relação a isso, não foram feitas pesquisas detalhando sobre a mudança de desempenho que ocorre entre a integração do ROS 2 com relação ao Docker, existem poucas pesquisas sobre essa mudança de desempenho entre a integração do ROS 2 com o Docker, esta integração pode levar a perdas de desempenho para o robô, causando atraso de mensagens recebidas ou travando o robô, esse atraso pode ser causado por conta da sobrecarga do sistema por conta dos recursos a mais utilizados pelo Docker. O projeto inicialmente visava o estudo e avaliação desta integração em um robô real, mas por conta de atrasos na liberação de acesso aos recursos necessários para fazer os testes no robô real, as conclusões deste projeto estão limitadas a apenas o componente simulado.

1.1 OBJETIVO

O objetivo deste projeto é o de documentar e avaliar o desempenho de um robô simulado com e sem conteinerização em arenas simuladas baseadas na arena da sala K4-04 do Centro Universitário FEI sendo executado no software Gazebo Classic.

1.2 ESTRUTURA DO TRABALHO

O decorrer deste projeto está dividido da seguinte maneira: na seção 2, serão abordados os conceitos e ferramentas utilizados no projeto junto para que o leitor possa entender com clareza o tema abordado no projeto e os termos utilizados.

Na seção 3, são abordados os trabalhos que possuem alguma relação com o projeto, como métricas, ideias e etc.

Na Seção 4, é abordada a metodologia utilizada para o desenvolvimento deste projeto, demonstrando as técnicas utilizadas e os passos a serem realizados para atingir o objetivo final.

Na seção 5, são abordados os experimentos e resultados obtidos ao longo do projeto, como os experimentos foram executados, os problemas obtidos e, no final, mostrar os resultados que foram obtidos, explicando o que era esperado e o que acabou por ser obtido.

Na seção 6, são abordadas a discussão, conclusão e os trabalhos futuros. São explicados os resultados obtidos do projeto, o que foi concluído e, no final, explicar alguns passos que não foram realizados e que estão disponíveis para algum aluno desenvolver no futuro.

2 CONCEITOS FUNDAMENTAIS

Este capítulo aborda os conceitos teóricos e as ferramentas utilizadas no desenvolvimento deste trabalho, com o intuito de descrever e relacionar estes conceitos, para que haja um entendimento claro nas etapas subsequentes. É importante também apresentar, como cada um desses conceitos e tecnologias se relacionam com o objetivo central deste estudo.

O principal conceito a ser abordado é a Conteinerização, uma forma de virtualização feita para ser mais rápida e flexível que a emulação, é um processo de implantação que consegue ser executado em diversos dispositivos e sistemas operacionais. Isso ocorre pelo fato de que um contêiner consegue armazenar os arquivos e bibliotecas para ser executado, permitindo a um usuário executar uma aplicação de outro sistema operacional no sistema operacional que o mesmo possua. Além disso, o contêiner permite que falhas ocorram sem afetar outros processos que não estão agrupados no mesmo. (WEN et al., 2023). As carecteristicas de isolamento e empacotamento são extremamente importantes para este trabalho, pois é necessário entender se essa camada adicional introduz atrasos, consumo excessivo de recursos ou variações no desempenho do robô. a partir deste conceito, aplica-se o Docker que será utilizado para fazer a conteinerização, o Docker que é uma plataforma utilizada para desenvolvimento, envio e funcionamento de aplicações de maneira separada da infraestrutura por conta da conteinerização. Por conta deste fator, o usuário consegue gerir as aplicações da mesma maneira que gera sua infraestrutura. Outro fator importante é que o Docker permite que as aplicações desenvolvidas sejam testadas e executadas com menos atraso do que a maneira convencional. Contêineres são bons para fluxos de integrações e entregas de trabalho contínuas. (DOCKER, 2025)

O Docker, foi utilizado no robô para realizar os testes, para isso, foi utilizado o Robot Operating System 2 (ROS 2), que é um meta-sistema operacional um sistema operacional para robôs, ele realiza funções similares a outros sistemas operacionais, com exceção do controle de CPU, pois executa processos robóticos (planejamento de movimento, navegação, manipulação de objetos, entre outros), fluxos de dados, entre outros e possui bibliotecas e ferramentas que executam códigos em múltiplos computadores. Este conceito é utilizado pelo ROS 2, um dos métodos conjuntos de ferramentas e bibliotecas para o desenvolvimento de aplicações mais utilizado nos robôs atuais, sendo uma camada acima de um sistema operacional real, que oferece abstrações e serviços para os sistemas robóticos. O ROS 2 é justamente um meta-sistema operacional de código aberto utilizado para auxiliar a desenvolver aplicações para robôs. O mesmo possui serviços que outros sistemas operacionais normalmente possuem, mas com o foco maior

para a área da robótica, facilitando comunicação entre processos, funções que se comunicam com as demais e entre muitos outros. Para o desenvolvimento do projeto, foi utilizado o ROS 2, que mantém o conceito modular e distribuído, mas possui melhorias e mais funcionalidades que o ROS original (Figura 1) (OPEN et al., 2018).

Para conectar as aplicações, foram utilizados Data Distribution Service (DDS), que são protocolos Middleware. Os Middlewares são uma camada de software que conecta as aplicações a um sistema operacional, permitindo uma comunicação e compartilhamento de dados mais simples entre os componentes de um sistema. Esta facilidade permite que os desenvolvedores foquem no desenvolvimento das principais funções de uma aplicação, pois a comunicação entre a aplicação e o sistema operacional está sendo feita pelo middleware. (RED; HAT, 2023) O DDS é um protocolo middleware e uma API para conexão centrada em dados, este protocolo integra os componentes de um sistema que muitas aplicações precisam, como arquitetura escalável, confiabilidade e prover conectividade de dados de baixa latência. Este protocolo foi criado pela Object Management Group (OMG). (OMG et al., 2025) Para este projeto, foram utilizados dois tipos de DDS, o primeiro é o Fast DDS uma implementação de DDS feita em C++ que possui uma biblioteca que oferece uma API e protocolo de comunicação que disponibiliza um modelo Publisher-Subscriber centrado em dados. Este modelo visa ser eficiente e confiável para distribuir as informações para o sistema em tempo real. (EPROSIMA, 2019) O segundo DDS utilizado é o Cyclone DDS, que é uma implementação de DDS com alto desempenho, permite que os desenvolvedores que o utilizam possam criar gêmeos digitais das entidades de seus sistemas, permitindo compartilhar estados, eventos, fluxos de dados e mensagens pela rede em tempo real, visa ser rápido, consistente e seguro. (ECLIPSE; FOUNDATION, 2022)

Os testes simulados foram executados em modelos baseados na arena presente na sala K4-04 do Centro Universitário FEI, para isso, foi utilizado o software Gazebo Simulator Classic que é um software usado para desenvolver simulações, possui diversos projetos de código aberto para que os interessados possam utilizar e desenvolver suas próprias simulações. Neste software estão presentes também diversos modelos, tanto como objetos como também robôs (Figura 2). (OPEN et al., 2025) Para realizar estes testes, foi utilizado o modelo virtual do robô TurtleBot3 Burger (Figura 3) que é um robô customizável de preço acessível ao público baseado no modelo ROS para ser utilizado como um material educativo, de pesquisas, entretenimento pessoal e etc, é um robô que foi desenvolvido com o intuito de ser barato, por conta disto, o mesmo não possui uma grande funcionalidade ou qualidade, mas o mesmo compensa na relação da quantidade de aplicações que o mesmo consegue realizar. (ROBOTIS, 2025). Para verificar o desempenho do computador durante a execução dos testes, será utilizada a ferramenta Nigel's

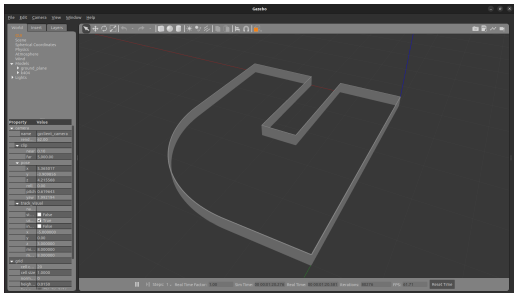
Monitor (Nmon), um administrador, otimizador e avaliador de desempenho de sistemas para o sistema operacional Linux que mostra importantes informações de desempenho do computador, como CPU, memória RAM, disco, kernel, entre outros. O mesmo pode fornecer as informações diretamente pelo terminal ou salvar em um arquivo para o usuário. (GRIFFITHS, 2025)

Figura 1 – Exemplo do ROS 2, utilizando turtlesim, ROS e RQT

Fonte: Autores

Legenda: A figura apresenta apenas um exemplo em que são utilizados o ROS 2, turtlesim e RQT, sendo apenas o ROS 2, utilizado no projeto

Figura 2 – Exemplo utilizando o Gazebo Simulator Classic



Fonte: Autores

Legenda: A figura apresenta a base da arena utilizada para os testes realizados no projeto

Figura 3 – TurtleBot3 Burger

Fonte: FOUNDATION, 2025

3 TRABALHOS RELACIONADOS

DESENVOLVER, COLOCAR METODOLOGIA DOS PROJETOS

3.1 BARE-METAL VS. HYPERVISORS AND CONTAINERS: PERFORMANCE EVALUATION OF VIRTUALIZATION TECHNOLOGIES FOR SOFTWARE-DEFINED VEHICLES

(WEN et al., 2023).

O projeto de Wen et. al. (2023), auxilia com relação ao entendimento da conteinerização em sistemas embarcados e também com relação ao seu desempenho. No artigo, é detalhada a utilização de diferentes formas de conteinerização e seu efeito no desempenho em diferentes tipos de hardware. Foram realizados testes gerais que envolviam CPU, memória, rede e disco, em três ambientes diferentes: máquina virtual, contêi(SOBIERAJ; KOTYńSKI, 2024).ner e um contêiner dentro de uma máquina virtual. Com isso, foi concluído que as máquinas virtuais e os contêineres possuem um desempenho semelhante ao bare-metal (servidor físico que é de uso exclusivo para apenas um cliente, sem a camada de virtualização que fica entre o hardware e o sistema operacional), onde entre a CPU, rede e memória, sofria uma perda de no máximo 5% enquanto no disco a diferença era de até 35%. Foi observado que o Docker e a KVM (máquina virtual baseada no Kernel) foram 5 a 10% mais lentos que o bare-metal, com o Docker sendo mais lento ainda na primeira inicialização, mas levando a concluir que conteinerização e virtualização podem ser utilizados em aplicações para automóveis. (WEN et al., 2023).

3.2 DOCKER PERFORMANCE EVALUATION ACROSS OPERATING SYSTEMS

(SOBIERAJ; KOTYńSKI, 2024)

O trabalho de Sobieraj e Kotyński (2024), seu projeto auxilia no entendimento dos conceitos de avaliação do Docker com relação a outros sistemas operacionais. Para verificar a diferença entre os sistemas operacionais, foram utilizados sistemas operacionais recém-instalados que eram MacOS Ventura, Ubuntu 22.04 e Windows 10 rodando em um MacBook Pro 13. Os testes consistiam em estressar o Docker com relação à CPU, rede e na resiliência do mesmo. Após realizar os testes, foi observado que o sistema operacional possui uma importante influência sobre o desempenho presente no conteiner Docker. Alguns possuíam benefícios em relação

a outros em uma determinada categoria. O macOS se destacou com relação aos dados obtidos nas configurações utilizadas nos sistemas docker, não sofrendo grandes perdas de desempenho, se mostrando extremamente versátil, o Linux se mostrou mais eficiente quanto às aplicações que raramente utilizam escrita no disco, se mostrando uma escolha melhor que o MacOS com relação a bancos de dados em memória, cache e entre outros, pois por não necessitarem de tanta escrita, essas aplicações se beneficiam mais quando executadas em um conteiner com Linux, o Windows acabou não se beneficiando tanto quanto os outros, a não ser pela taxa de transferência de rede entre os contêineres. Assim como o Linux, o Windows possui problemas com a velocidade em que a escrita é feita e com isso. Este artigo auxilia no entendimento com relação aos tipos de testes que podem ser realizados para analisar o desempenho do Docker, auxiliando de uma maneira que possa ser um pontapé inicial para o desenvolvimento dos testes com o Docker, e como medir o desempenho de um contêiner.

3.3 DA CONTAINERIZED MICROSERVICE ARCHITECTURE FOR A ROS 2 AUTONOMOUS DRIVING SOFTWARE: AN END-TO-END LATENCY EVALUATION

(SOBIERAJ; KOTYńSKI, 2024) (WEN et al., 2024).

Conforme Wen et. al. (2024), o artigo aborda a arquitetura baseada em microserviços para sistemas automotivos. Cada serviço foi realizado em contêineres separados, pois os testes realizados identificaram que este método é viável e acaba por melhorar a latência existente em sistemas operacionais Linux sem contêineres, que obteve uma latência de 5 a 8% end-to-end, além de reduzir a latência máxima, o que mostra a vantagem no uso de conteinerização para os sistemas automotivos em tempo real. Foi concluído que o ROS 2, utilizado para avaliar a arquitetura de microserviços para uma aplicação real de direção autônoma, foi de extrema importância por conta de sua arquitetura distribuída que é baseada em nós e possui comunicação DDS, o que levou ao isolamento das funções do veículo e facilitou a migração para contêineres. Na conteinerização, o ROS 2 perde um pouco de seu desempenho, mas em aplicações complexas como o Autoware, a mesma acaba por melhorar, reduzindo o uso de CPU e memória. Este artigo auxilia no entendimento do uso da conteinerização com relação ao ROS 2, seus problemas e seus acertos. (WEN et al., 2024).

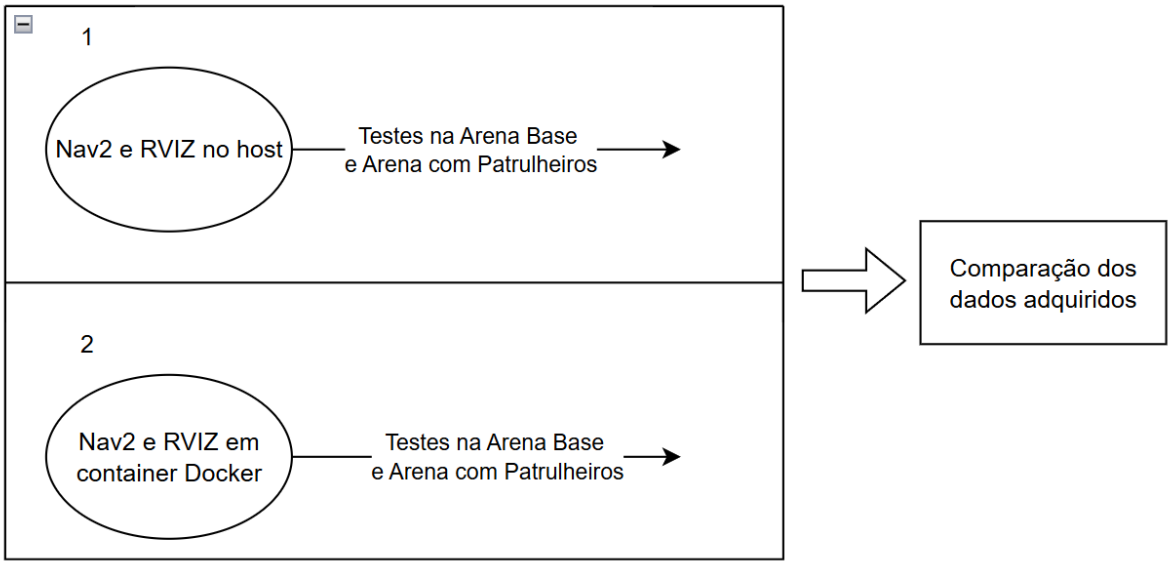
4 METODOLOGIA

ADICIONAR lista de materiais; uma lista de métodos utilizados; e uma lista de métricas utilizadas. Foi utilizado ROS 2 Humble, Docker Engine Jammy 22.04 Stable e Gazebo Classic. A escolha das versões ROS e Docker Engine foi por conta de serem versões Long Term Support (LTS), enquanto a escolha do Gazebo Classic se deu por conta de todos os recursos já disponíveis para esta versão de Gazebo, especificamente os disponibilizados pela ROBOTIS. Para a coleção de dados, foi escolhido o Nigel's Monitor (nmon) por conta da sua capacidade de obter dados gerais do sistema, e de ter programas já criados que conseguem transformar seus arquivos de log em gráficos, utilizamos NMONVisualizer exibindo as informações de forma acessível. Nmon obtém os dados gerais de desempenho da máquina a cada segundo, dados como, mas não limitados a, Utilização de CPU, Rede e RAM, permitindo que outros projetos avaliem categorias diferentes das propostas neste, parando de obter estes dados somente quando o robô principal chega na posição setada pelo teste. As métricas utilizadas foram uso de CPU, uso de RAM e uso de Rede, por conta de que todas essas métricas estão relacionadas diretamente ao desempenho do robô. Aumentos significativos e picos repentinos destas métricas podem sinalizar um perigo de aumento de latênciajitter, métricas relacionadas à diferença de tempo entre mandar uma mensagem e receber a resposta, e diminuição de RTF, métrica que mede o fator de processamento em tempo real na simulação, mas estes dados não foram diretamente medidos por conta do foco do projeto em recursos computacionais, e não em recursos temporais. O container docker usa a configuração "network_mode: host", utilizando a rede do host para comunicação. A metodologia (Figura 4) é dividida em 2 partes, sem e com Docker, respectivamente, sendo que a integração com o Docker é na parte de navegação, isolando a stack nav2 e rviz do host. Todos os testes são executados em uma arena similar à presente na sala K4-04 da FEI. Foi utilizado o programa Nmon para obter os dados durante os testes, obtendo o desempenho do sistema inteiro, servindo para garantir que todas as estatísticas estão sendo comparadas de forma íntegra. Foram utilizados scripts bash[1] para tornar a execução dos testes consistente, tendo cada parte do teste também em scripts, facilitando a criaçãomodificação de testes personalizados. Foi criado um pacote ROS 2 (TCC) DESENVOLVER para organizar scripts, mapas e arquivos de lançamento, sendo que alguns destes são modificações de arquivos presentes em outros pacotes deste projeto. Foi utilizado um arquivo modificado de AMCL (burger.yaml), que

[1] As configurações e arquivos para facilitar o setup estão presentes no repositório <https://github.com/joca2511/TCC_Docker>

é um arquivo usado para a localização dos robôs no ROS 2, para inicializar imediatamente a localização do TurtleBot, tornando os testes completamente autônomos e mais consistentes, já que a localização manual pode ser rejeitada. No primeiro teste, o robô simplesmente precisa percorrer a arena de ponta a ponta, tendo um SLAM pré-feito para o auxiliar, sem obstáculo algum. No segundo teste, o robô também precisa percorrer a arena de ponta a ponta com um SLAM pré-feito, mas dessa vez a arena está populada por outros robôs, que servem para atrapalhar o trajeto do robô principal, testando sua resiliência quanto à mudança de rotas com e sem Docker. DESENVOLVER Foram realizados testes em que o robô realizaria o mesmo objetivo dos dois primeiros testes, mas com o diferencial de a arena possuir um labirinto que pode ser montado na arena física (seguindo e respeitando as regras presentes na arena física da K4-04) como obstáculo para o robô.

Figura 4 – Fluxograma do projeto



Fonte: Autores

5 EXPERIMENTOS E RESULTADOS

Explicar melhor os experimentos, elaborar melhor os nomes dos testes (EXPLICAÇÃO CONFUSA)

Os experimentos usam um modelo digital da arena física presente no laboratório da sala K4-04 da instituição. O motivo de esta arena ter sido escolhida se deu por conta da comparação que futuramente pode ser realizada entre os testes virtuais e reais.

Foram feitos cinco experimentos com resultados satisfatórios, e um que acabou não fornecendo os resultados devidos por conta de instabilidades. Todos os testes foram feitos em um computador com as seguintes especificações: Linux Jammy 22.04 com um processador AMD Ryzen 5 1600X (12 Cores, 3.6GHz), 16 GB RAM DDR4. Todas as ferramentas e configurações destas estão disponíveis em um playbook, no repositório. A imagem Docker utilizada no projeto se baseia na imagem usada para ROS 2 Humble, presente no repositório oficial da robotis[1], modificada para ter as configurações e arquivos necessários. O primeiro deles (Setup) teve testes iniciais para confirmar o comportamento do ROS 2 dentro de um container Docker. O segundo experimento (Base), usaram-se simulações utilizando o mapa base sem patrulheiros. O terceiro experimento (Patrulheiros) usou simulações utilizando o mapa base com patrulheiros. O quarto experimento (Base + Docker) usou simulações utilizando o mapa base sem patrulheiros, com a stack nav2 e rviz dentro de um container Docker. No quinto experimento (Patrulheiros + Docker), usaram-se simulações utilizando o mapa base com patrulheiros, com a stack nav2 e rviz dentro de um container Docker. Foram também realizados experimentos usando arenas com labirintos para testar a navegação do robô, mas a navegação acabou tendo problemas em experimentos com e sem Docker, dando a indicar que é um problema com o mapa, ou com as bibliotecas ROS 2 utilizadas, e não com a conteinerização. EXPLICAR PROBLEMA

5.1 CONTEÚDO REPOSITÓRIO

O repositório GitHub criado para este projeto possui várias ferramentas que podem ser utilizadas para facilitar a reprodução dos testes executados. O README contém um overview de como instalar o projeto e utilizar os scripts presentes no projeto. Foi criado um playbook ansible (playbook.yaml) para facilitar a configuração e instalação do Docker, ROS 2 Humble, Gazebo Classic e suas dependências. Foram também criados vários arquivos .sh para facilitar

[1] (https://github.com/ROBOTIS-GIT/turtlebot3/tree/main/docker/humble)

a reprodução dos testes, por garantir que os comandos corretos serão executados na sequência correta, sem necessidade de intervenção do usuário, onde os arquivos “inicioRapido” possuem os comandos utilizados para cada um dos testes feitos para este projeto. Os arquivos .sh na pasta /scripts possuem funcionalidades genéricas criadas para os testes, como a inicialização do nmon (iniciarNmon.sh), mover o robô ao inicializar rviz e mandar uma mensagem de goal (moverMain.sh), entre outras. No pacote ROS 2 criado (localizado em /tcc) está uma versão modificada do arquivo turtlebot3_absolute_move.py (turtlebot3_absolute_move_Arena.py, localizada em /tcc/tcc), utilizado na movimentação dos robôs patrulheiros. Esta modificação faz com que os robôs inicializados entrem em um loop infinito de movimentação entre 2 pontos especificados, lógica utilizada no script rotasRobos.sh. Há também uma versão modificada de multi_robot_Arena.launch.py com as posições iniciais dos robôs patrulheiros e turtlebot3_world.launch.py (turtlebot3_Arena.launch.py), que permite carregar o robô em uma posição fixa em qualquer mapa, contanto que o nome seja especificado e o arquivo .world presente em /tcc/worlds.

5.2 RESULTADOS SETUP

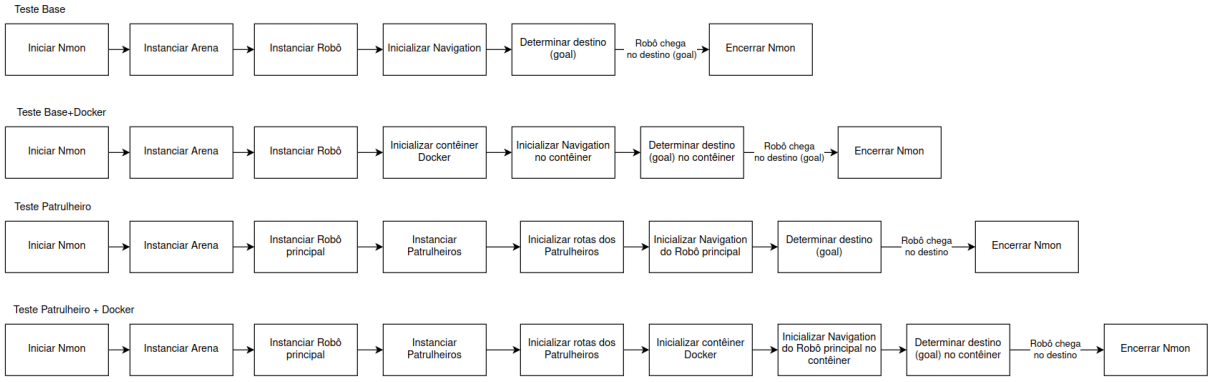
Foram feitos testes para implementar a parte simulada proposta (Figura 5), foi utilizado o manual (ROBOTIS, 2025) e pacotes[2] disponíveis pelo grupo ROBOTIS, tornando o desenvolvimento desta parte rápido. A utilização e desenvolvimento dos projetos ROS 2 dentro do Docker foi facilitada pela utilização de workflows no GitHub, onde as imagens de teste foram automaticamente construídas e publicadas como pacotes no repositório, o que reduziu o tempo do processo de construir as imagens localmente, que demorava mais de 20 minutos para no máximo 5 minutos, além de também as disponibilizar para outros usarem.

Houve certas dificuldades para integrar o ROS 2 dentro do contêiner Docker com o ROS 2 nativo, que lidaria com a simulação Gazebo. Foi descoberto que o FastDDS (middleware utilizado por padrão pelo ROS 2 Humble) não interage de forma consistente com Docker. Ele era capaz de compartilhar os tópicos entre os ambientes, mas causava falha na publicação e recebimento de mensagens, não mostrando nenhuma. (Figura 6) DESENVOLVER

Para solucionar isso, o FastDDS foi substituído por outro middleware disponível para ROS 2 Humble, CycloneDDS, o qual foi utilizado especificamente no container Docker. (Figura 7)

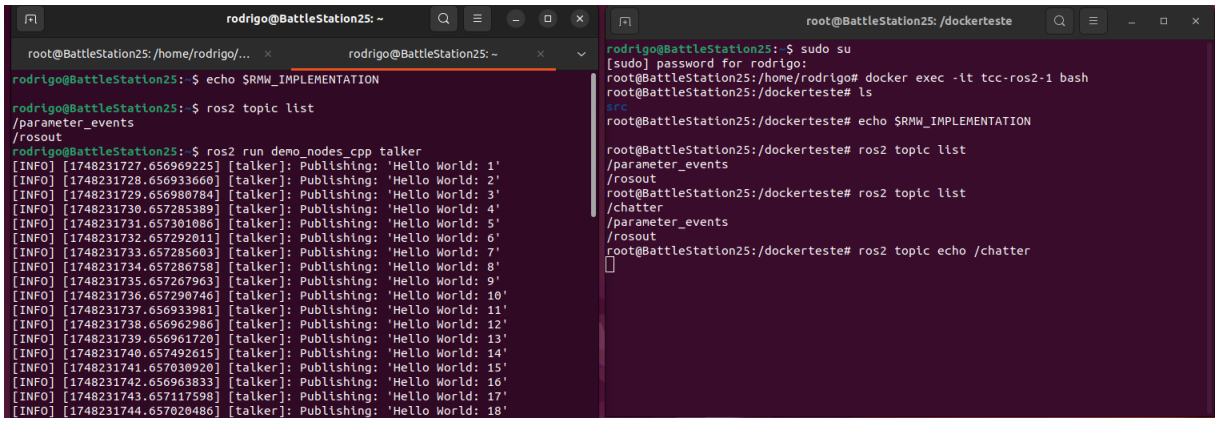
[2](https://github.com/ROBOTIS-GIT/turtlebot3)

Figura 5 – Fluxograma dos testes realizados



Fonte: Autores

Figura 6 – Falha no teste padrão



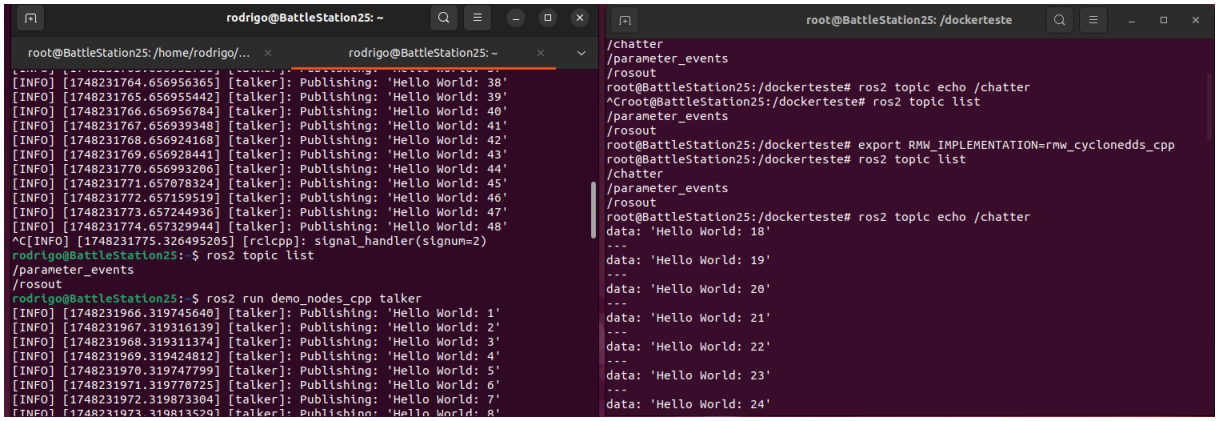
Fonte: Autores

Legenda: O terminal à esquerda é o host com implementação padrão (FastDDS), enquanto o terminal à direita está utilizando o Docker com implementação padrão (FastDDS).

5.3 TESTES BASE

Foram feitos testes para adquirir os dados de desempenho das simulações sem Docker, utilizando a arena base sem patrulheiros. Estes testes consistiam em fazer o robô percorrer um trajeto de uma ponta até a outra da arena. Durante a execução, foi utilizado o software Nmon para verificar o desempenho de CPU, memória RAM e rede, e assim armazenar os dados. Os testes com a arena base + Docker, o robô realizou o mesmo trajeto que o teste anterior de seguir de uma ponta da arena até a outra. Os testes foram realizados enquanto o software Nmon verificava o desempenho do sistema.

Figura 7 – Sucesso no teste com CycloneDDS



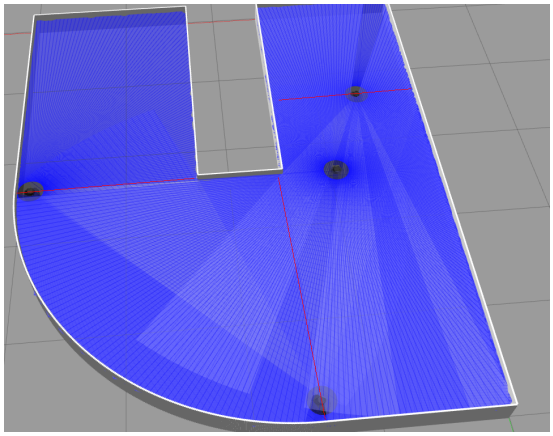
Fonte: Autores

Legenda: O terminal à esquerda é o host com implementação padrão (FastDDS), o terminal à direita utiliza o Docker com implementação rmw_cyclonedds_cpp.

5.4 TESTES PATRULHEIROS

Neste teste foram posicionados robôs patrulheiros que percorriam um simples trajeto trajeto de linha reta onde saiam e retornavam as seus pontos de origem (Figura 8) para servir de obstáculo para o robô que deveria percorrer o mesmo trajeto de antes. Novamente foi utilizada a arena base, e sem o uso do Docker. Foram feitos testes para adquirir os dados de desempenho de simulações com Docker, utilizando a arena base com patrulheiros.

Figura 8 – Arena Patrulheiros



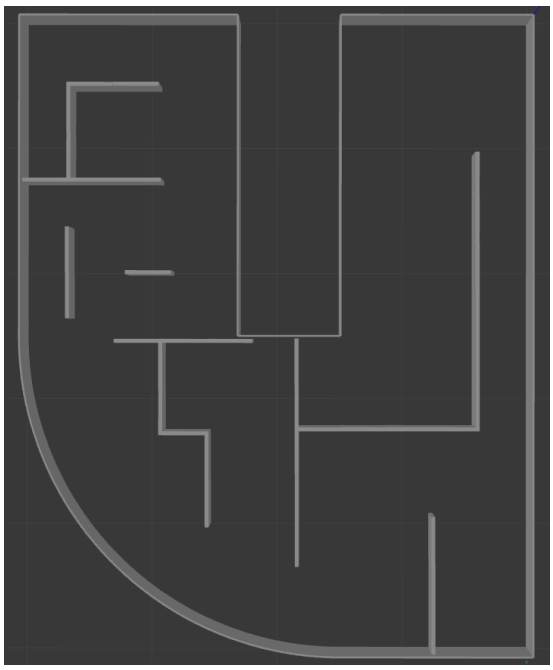
Fonte: Autores

Legenda: As retas em vermelho são os trajetos realizados pelos patrulheiros

5.5 TESTES LABIRINTOS

Inicialmente, foram realizados testes sem patrulheiros, com e sem o uso de Docker, em duas arenas com labirintos (Figuras 10 e 9) como obstáculo para o robô. Os labirintos foram desenvolvidos utilizando placas, que são os obstáculos presentes na arena física da K4-04, respeitando o espaço da arena, placas e possíveis posições. Estes testes acabaram por falhar, por conta da inconsistência da navegação do teste, tendo casos em que a navegação falhava no meio do caminho, ou outros em que eram necessárias entre 4 - 12 recuperações era necessário refazer o teste de 4 a 12 vezes para o robô conseguir realizar o trajeto completo. Por conta disso, os testes com os patrulheiros não tiveram resultados consistentes em relação as métricas que seriam medidas.

Figura 9 – Arena Labirinto 1

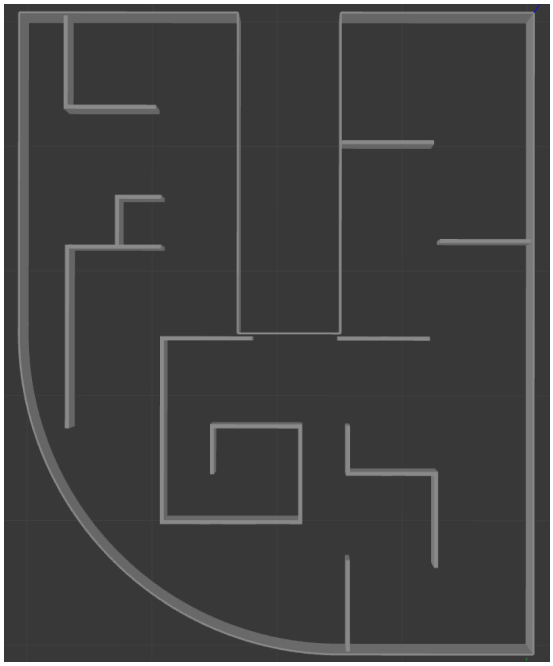


Fonte: Autores

5.6 RESULTADOS

Os resultados obtidos foram salvos e utilizados em gráficos gerados pela biblioteca Pandas. Os dados apresentados foram a porcentagem do uso do sistema (CPU, memória RAM e Rede), com relação ao passo executado no Nmon. Estes gráficos revelam que a utilização do Docker afetou o desempenho, mas de uma maneira mínima. Nos gráficos de CPU (Figura 11), pode-se ver que a mesma acabou por ser mais exigida quando não se utilizava o Docker, por

Figura 10 – Arena Labirinto 2



Fonte: Autores

volta de 1% da média de uso. Nos gráficos de memória RAM (Figura 12), pode-se ver que a diferença do uso é de cerca de 500 MB a mais quando se usa o Docker. Este aumento ocorreu por conta da necessidade do Docker de subir o container. Nos gráficos de rede (Figura 13), pode-se ver que o uso do Docker é mais evidente, tendo um pico inicial muito alto, mas que diminui logo em seguida. Este aumento ocorreu por conta da necessidade do Docker em se comunicar, via rede, com os outros tópicos utilizados pelo host.

Os testes nas arenas com patrulheiros se demonstraram muito semelhantes aos testes das arenas bases em termos de gráficos, mas os mesmos utilizaram bem mais recursos do sistema. Nos gráficos de CPU (Figura 14), pode-se ver que foi exigido bem mais da CPU do sistema. Enquanto na arena base era exigido até 30% da CPU, com os patrulheiros foi exigido 50%, nesta arena a falta do Docker exigiu mais do sistema, enquanto com o Docker houve poucos picos de uso. Nos gráficos de memória RAM (Figura 15) se pode ver que se diferenciaram no uso de cerca de 500 MB a mais no uso do Docker, semelhante à arena base, exigindo mais da memória no geral do teste. Nos testes de rede (Figura 16), pode-se ver que o uso do Docker é novamente mais evidente, tendo um pico de uso muito alto, que logo diminuiu, mas que ainda ficou mais acima dos testes sem Docker.

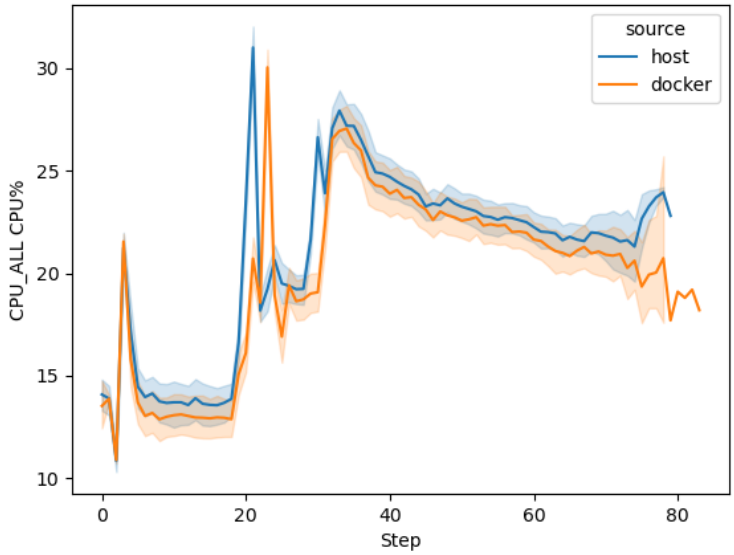
Com relação aos testes com os labirintos, os mesmos se demonstraram instáveis, então não foram testados por conta da quantidade de testes aleatórios que precisavam ser feitos para

ter um teste que gerasse bons resultados para serem observados. Porém a instabilidade foi a mesma com e sem o uso do Docker.

Os dados apresentados são a média e desvio padrão das 30 execuções dos experimentos que foram descritos na seção anterior e exibem DESENVOLVER

EXPLICAR MELHOR OS GRÁFICOS

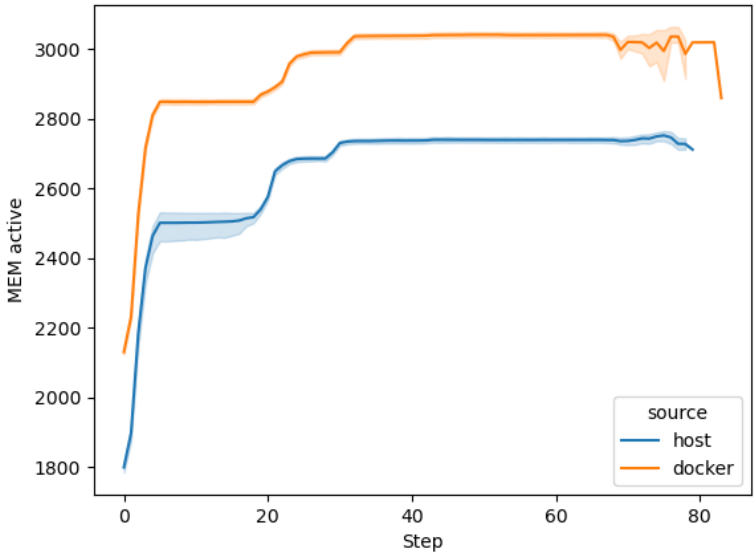
Figura 11 – Gráfico da métrica de porcentagem de uso total de CPU dos testes na arena base com e sem Docker com banda de erro ao tempo.



Fonte: Autores

Legenda: Cada passo (step) é igual a 1 segundo.

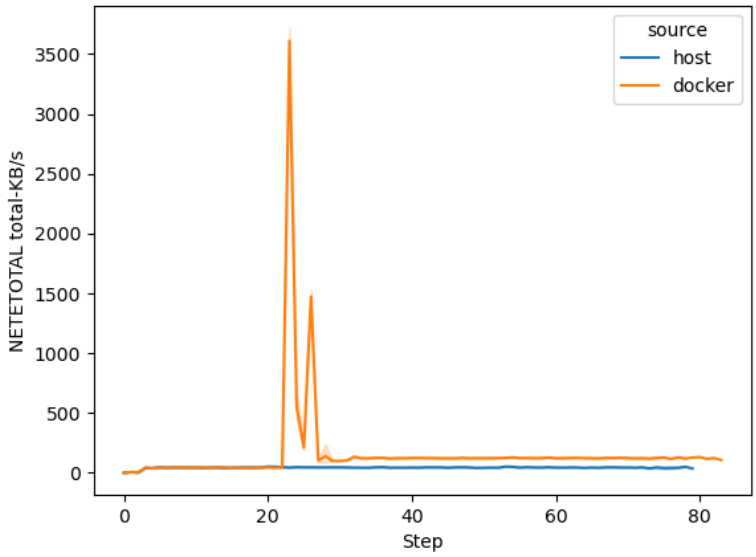
Figura 12 – Gráfico da métrica de porcentagem de uso total de memória RAM dos testes na arena base com e sem Docker com banda de erro ao tempo.



Fonte: Autores

Legenda: Cada passo (step) é igual a 1 segundo.

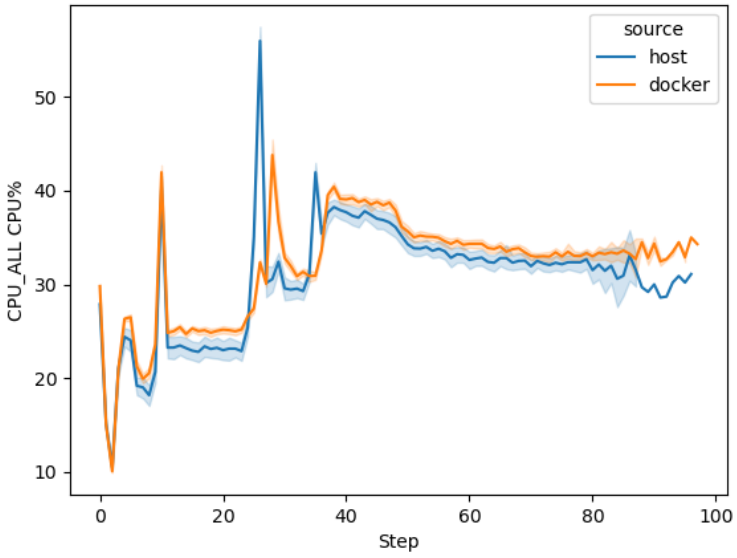
Figura 13 – Gráfico da métrica de uso total de Rede em kb/s dos testes na arena base com e sem Docker com banda de erro ao tempo.



Fonte: Autores

Legenda: Cada passo (step) é igual a 1 segundo.

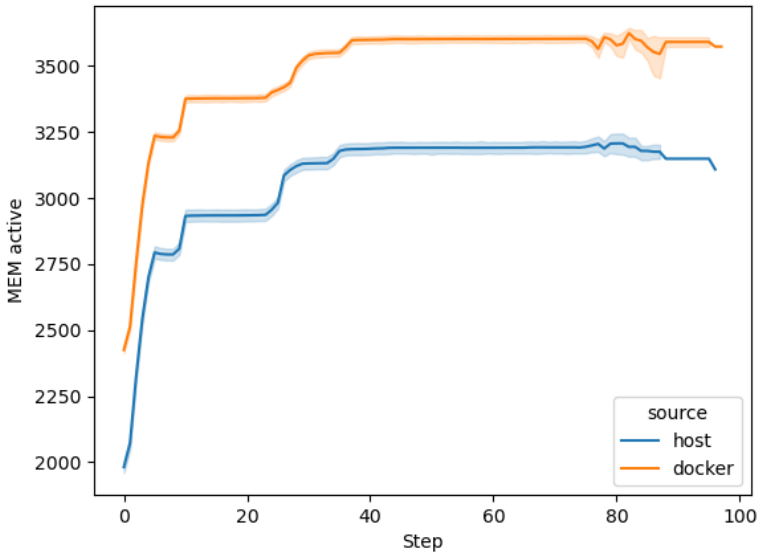
Figura 14 – Gráfico da métrica de porcentagem de uso total de CPU dos testes na arena com robôs patrulheiros, com e sem Docker com banda de erro ao tempo.



Fonte: Autores

Legenda: Cada passo (step) é igual a 1 segundo.

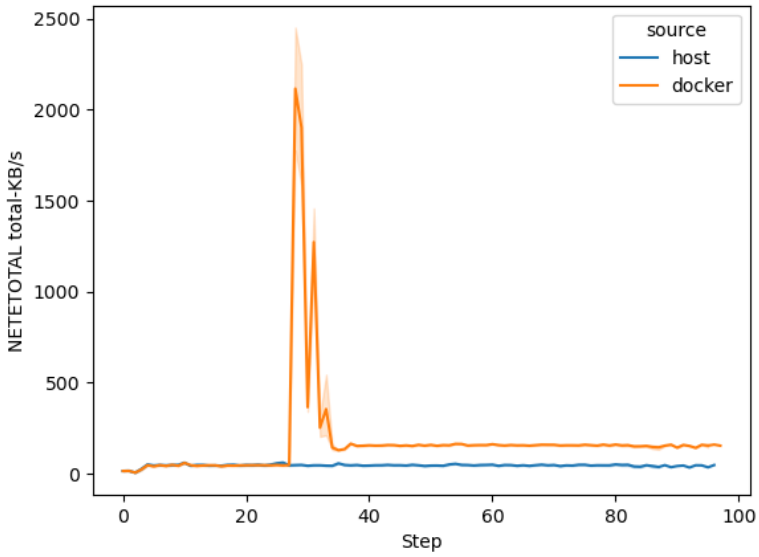
Figura 15 – Gráfico da métrica de porcentagem de uso total de memória RAM dos testes na arena com robôs patrulheiros, com e sem Docker com banda de erro ao tempo.



Fonte: Autores

Legenda: Cada passo (step) é igual a 1 segundo.

Figura 16 – Gráfico da métrica de uso total de Rede em kb/s dos testes na arena com robôs patrulheiros, com e sem Docker com banda de erro ao tempo.



Fonte: Autores

Legenda: Cada passo (step) é igual a 1 segundo.

6 CONCLUSÃO

A partir dos resultados obtidos por meio dos experimentos realizados, pode ser constatado que o uso do Docker gera um impacto no desempenho do robô simulado. MAIS CERTEZA NA AFIRMAÇÃO Esse impacto pode ser medido e observado por meio do consumo do desempenho da CPU, memória RAM e tráfego de rede, obtidos por meio da ferramenta Nmon, que mediu os testes realizados no simulador gazebo, onde o robô realizou trajetos com e sem o uso do Docker. Foi observado que a diferença no desempenho foi mínima, mostrando que o uso do Docker não compromete o funcionamento da simulação. Com isso, foi concluído que o Docker pode ser utilizado, mas pode ter alguns problemas, pois não foram realizados testes com tarefas críticas ou com pouco recurso disponível, para realizar testes simulados utilizando o ROS 2, o que acaba por gerar praticidade e portabilidade para o usuário que não precisa se preocupar com prejuízos significativos.

6.1 TRABALHOS FUTUROS

Devido a problemas com relação ao uso da sala K4-04 e seus equipamentos e à falta de tempo, os testes em um robô real não foram realizados, isso acabou por alterar o escopo do projeto, removendo o componente real das avaliações, sendo decidido que os testes e avaliações do robô real seriam realizados como um trabalho futuro.

Como trabalho futuro, ficaria a avaliação e estudo da integração em um robô real, utilizando as ferramentas, arquivos e metodologias criadas por este projeto para facilitar a obtenção de dados, como a Dockerfile customizada, os scripts para fácil utilização e organização de vários comandos.

Onde esta o trabalho de IHC com trabalho futuro? O texto está muito raso. Falta aprofundamento. Provavelmente o grupo fez mais do que essa descrito. Minha sugestão é que o grupo realize os testes no ambiente real e apresentem futuramente um texto mais completo. Banca ROS 2 + Docker

a) O TurtleSim foi utilizado?

b) Slide 5 e 6 - Podem colocar as imagens dos trabalhos relacionados no texto.

c) O que dos trabalhos relacionados foi utilizado no trabalho?

d) O que dos trabalhos relacionados foi utilizado no trabalho?

e) Ficou claro na apresentação o que eram os patrulheiro

f) Imagens nos slides dos experimentos devem estar no texto

g) Gostaria de mais detalhes dos testes de navegação. Não faz sentido ele se perder. O ambiente estava mapeado?

h) A explicação na apresentação está melhoro que do texto

REFERÊNCIAS

DOCKER, I. **Docker**. 2025. Accessed: 2025-05-25. Disponível em: <https://docs.docker.com/get-started/docker-overview/>.

ECLIPSE; FOUNDATION. **CycloneDDS**. 2022. Accessed: 2025-05-24. Disponível em: <https://cyclonedds.io/>.

EPROSIMA. **FastDDS**. 2019. Accessed: 2025-05-24. Disponível em: <https://fast-dds.docs.eprosima.com/en/stable/>.

GRIFFITHS, N. **Nigel's Monitor**. 2025. Accessed: 2025-05-24. Disponível em: <https://nmon.sourceforge.io/pmwiki.php>.

OMG, O. et al. **DDS**. 2025. Accessed: 2025-05-24. Disponível em: <https://www.omg.org/omg-dds-portal/>.

OPEN et al. **ROS**. 2018. Accessed: 2025-05-24. Disponível em: <https://wiki.ros.org/ROS/Introduction>.

____. **Gazebo Simulator**. 2025. Accessed: 2025-05-24.

RED; HAT. **Middleware**. 2023. Accessed: 2025-05-24. Disponível em: <https://www.redhat.com/en/topics/middleware/what-is-middleware>.

ROBOTIS. **TurtleBot3 e-Manual: Overview**. 2025. Accessed: 2025-05-24. Disponível em: <https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/>.

SOBIERAJ, M.; KOTYńSKI, D. Docker performance evaluation across operating systems. **Applied Sciences**, v. 14, n. 15, 2024. ISSN 2076-3417. Disponível em: <https://www.mdpi.com/2076-3417/14/15/6672>.

WEN, L. et al. **A Containerized Microservice Architecture for a ROS 2 Autonomous Driving Software: An End-to-End Latency Evaluation**. 2024. Disponível em: <https://arxiv.org/abs/2404.12683>.

____. Bare-metal vs. hypervisors and containers: Performance evaluation of virtualization technologies for software-defined vehicles. In: . [s.n.], 2023. Disponível em: <https://www.researchgate.net/publication/372496789_Bare-Metal_vs_Hypervisors_and_Containers_Performance_Evaluation_of_Virtualization_Technologies_for_Software-Defined_Vehicles>.