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 Os resultados mostram que o apesar do aumento do uso de memória e da uma variação significative do uso de processador (segundo o Teste T realizado), o desempenho na simulação não é impactado pelo uso do Docker, 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 require a large amount of initial configuration, or that require portability. This project aims to identify and show the difference in performance between robots using Robot Operating System 2 (ROS 2), simulated with and without containerization. To verify this difference, tests were performed in the simulated environment of the Gazebo Classic software. The results show that simulation performance is not impacted by the use of Docker, which causes a small increase in memory usage, but without significant variations in processor usage The results show that despite the increase in memory usage and a significant variation in processor usage (according to the T-test performed), simulation performance is not impacted by the use of Docker, indicating that simulation using containers is feasible during the development process.

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

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 Este projeto realiza a comparação de uso de recursos (CPU, RAM e rede) em tarefas simuladas usando o ROS e Gazebo, com o robô simulado sendo executado direto no Sistema Operacional ou containerizado usando o Docker. Testes iniciais em robôs reais foram realizados com o objetivo de validar se os resultados obtidos em simulação podem ser observados com os robôs reais.

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).

Além do conceito da conteinerização, é importante conhecer sobre o jitter, que é um problema presente no mundo da robótica e do desenvolvimento de softwares. Jitter é um termo utilizado para o desvio de tempo, causando variações ou flutuações indesejáveis e imprevisíveis nos movimentos e comportamentos do robô, causando ações imprecisas que resultam em erros e, em casos mais graves, podendo forçar o sistema do robô a modos de segurança. Mesmo com esses problemas, o jitter não é o maior problema com relação a problemas de desempenho, existe também a latência, que é o atraso entre a entrada dos dados (input) até a "saída" (output) no sistema robótico, é um problema inevitável que acontece por ser um limite tecnológico ainda não solucionado. Quanto maior for a latência, maior o potencial de problemas com a precisão do sistema robótico. (QNX, 2024) As características de isolamento e empacotamento da conteinerização são extremamente importantes para este trabalho, por ser 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, 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 Um meta-sistema operacional não é um sistema operacional completo o mesmo adiciona novas funções para o sistema operacional base, no caso do ROS 2, o mesmo possui um foco maior para a robótica, 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. Para conectar o Docker com o sistema operacional, foram utilizados Middlewares, 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 Os Middlewares utilizados são DDS, que 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)

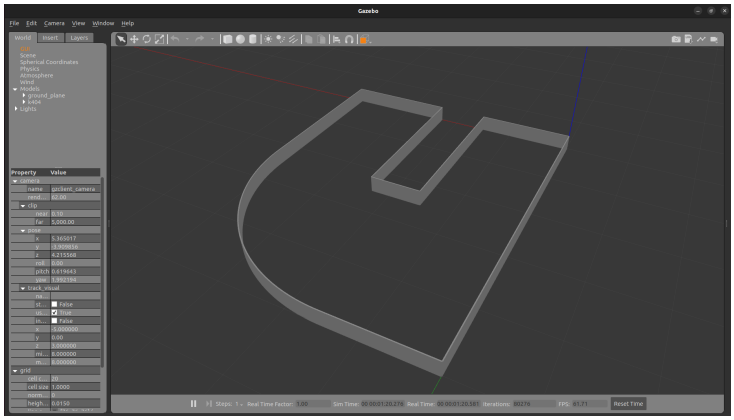
Os testes simulados foram executados em modelos baseados na arena presente na sala K4-04 do Centro Universitário FEI, para isso, foi utilizado o software Todos esses conceitos, foram utilizados nos testes simulados, realizados em uma arena simulada baseada na arena presente na sala K4-04 do centro universitário FEI, essa arena junto dos testes foram executados no 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 Para se realizar os testes, foi escolhido um robô presente no centro universitário FEI, com documentação e versões simuladas para uso, esse robô é o 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 Durante a execução dos testes, para se obter os dados do robô e então se realizar a comparação, foi 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 versão base da arena utilizada para os testes realizados no projeto

Figura 3 – TurtleBot3 Burger

Fonte: FOUNDATION, 2025

3 TRABALHOS RELACIONADOS

Este capítulo apresenta trabalhos que tratam de conteinerização de forma próxima a proposta deste trabalho, ou seja, tanto sobre a comparação de desempenho no uso de contêiner quanto do uso com o ROS para projetos de robótica.

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

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 O artigo de Wen et al. (2023) detalha 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, contêiner e um contêiner dentro de uma máquina virtual. Integrado (utilizando uma Raspberry Pi 4 Modelo B), Computador desktop (Dell Optiplex 7040 PC) e uma estação de trabalho customizada de alto desempenho Para realizar estes testes, foram utilizados Whetstone (versão 1.2), Dhrystone (versão 2.2a) e Kcbench (0.9.5), ferramentas de medição de desempenho, para realizar os testes de CPU. Os testes de memória foram medidos pelo software RAMspeed (versão 3.5.0). Foi utilizado o iperf3 (versão 3.9) para medir os testes de rede, enquanto para analisar o desempenho do disco, utilizou-se o Dbench (versão 4.00), Bonnie++ (versão 2.00) e o Sysbench (versão 1.0.20). Foram utilizados e analisados diversos contêineres engine, sendo estes o Docker (versão 20.10.17), KVM (v1.25.3), Podman (versão 3.4.4) e Systemd-Nspawn (versão 249.11). Para realizar os testes em diversos contêineres, foi utilizada uma versão leve do Kubernetes, o k3s (versão V1.25.3). Foi utilizado o Autoware, um framework para realizar simulações de condução autônoma, para verificar a inicialização dos ambientes virtualizados e contêinerizados.

Os testes de CPU consistiam em avaliar o desempenho no 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), contêineres (Docker e Podman) e em virtualizações (KVM). Para isso, utilizou-se o Whetstone para realizar 50 milhões de cálculos de ponto flutuante para medir a eficiência da CPU (a métrica usada foi o mflops, que são as milhões de operações rea-

lizadas por segundo), o Dhrystone para executar um código de cálculo simples continuamente, para analisar as operações comuns da computação (onde analisou as interações realizadas por segundo como métrica) e o Kcbench para compilar o código do kernel do Linux e medir quanto tempo para a CPU completar a tarefa (utilizou-se a métrica de Kernel/hora). Ao medir a memória, foi utilizado o RAMspeed, que analisa a velocidade da escrita e leitura na memória e quanto tempo se leva para realizar essas operações, repetindo este processo várias vezes nas três plataformas (a métrica para este teste foi a velocidade de escrita e leitura por segundo). Para os testes de rede, utilizaram o iperf3, que mede a largura de banda de envio e recebimento de dados por meio de transmissões TCP e UDP (o Throughput, sendo a taxa de transferência, e a latência foram as métricas deste teste). Por último, foi analisado o desempenho do disco, que foi medido pelo Dbench, que simula uma aplicação de operações de Input/Output em sistemas de arquivos, e o Bonnie++, que testa arquivos realizando operações de criação, leitura, gravação e exclusão de arquivos (as métricas usadas foram throughput de leitura e gravação, medido em megabytes por segundo, e o tempo para criar e excluir arquivos). Com isso, foi concluído que as máquinas virtuais e os contêineres possuem um desempenho semelhante ao bare-metal, 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. Este trabalho auxilia no entendimento da aplicação e avaliação de ambientes conteinerizados físicos e virtualizados. Para o desenvolvimento deste trabalho, serão utilizadas as análises realizadas com relação ao Docker e suas aplicações.

3.2 DOCKER PERFORMANCE EVALUATION ACROSS OPERATING SYSTEMS

O trabalho artigo de Sobieraj e Kotyński (2024) auxilia no entendimento dos conceitos de avaliação do Docker com relação a outros sistemas operacionais. Este estudo realiza testes onde o Docker é executado em diversos sistemas operacionais instalados em um MacBook Pro 13, com uma CPU da Intel i5-8257u @ 1.40GHz, 16GB LPDDR3 2133 Mhz de RAM e 256GB NVME SSD de armazenamento, com a versão 4.20 do Docker Engine e configurado para utilizar 8 CPUs lógicos, 16GB de RAM e 1GB de memória swap. 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 foram utilizados os sistemas operaiocnais MacOS Ventura 13.5.1, Ubuntu 22.04-6.4.8-t2-jammy e

Windows 10 22H2 recem-instalados. Os testes consistiam em estressar o Docker com relação à CPU, rede e na resiliência do mesmo. Neste projeto, o objetivo foi estressar o Docker para medir o desempenho da CPU usando um programa escrito em linguagem C para realizar o cálculo de Pi pela fórmula de Leibniz, sendo a métrica escolhida o tempo de execução do programa. Outro teste foi o teste do Sysbench, em que eram realizados cálculos dos números primos para verificar o desempenho do processador, com as métricas sendo a quantidade de operações por segundo.

Além disso, foram feitas diversas leituras e gravações aleatórias e sequenciais no disco rígido para medir o tempo de resposta e taxa de transferência das operações (teste de Input/Output). As métricas foram os testes de leitura e escrita. O teste do Iperf3 foi realizado a comunicação entre contêineres usando o protocolo TCP, depois utilizando o UDP, e por último a comunicação entre o host e o contêiner usando novamente o TCP. Este teste tinha como foco verificar o desempenho da rede, as métricas usadas foram Throughput (taxa de transferência) em TCP e UDP e a perda de pacotes. Em seguida, foi realizado o teste utilizando o 7Zip, em que era medida a velocidade de compactação e descompactação dos arquivos, e verificar o comportamento e desempenho do Docker e do sistema operacional. Foi realizado o teste de verificação de desempenho de um banco de dados no Docker, foi utilizado o Pgbench, onde era medida a taxa de transações por segundo (TPS) e verificado como o desempenho variava. Por último, foi feito o teste de Apache slowhttp attack, em que foram abertas diversas conexões HTTP lentas e, com isso, testada a capacidade e resistência ao sofrer um ataque de negação de serviço (Denial of Service - DoS) e verificar qual sistema acabou sofrendo mais. Foi medido o tempo de resposta durante o ataque DoS. Após realizar os testes, foi observado que o sistema operacional possui uma importante influência sobre o desempenho presente no conteiner. 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 Este

trabalho auxilia com o entendimento sobre o desempenho do Docker em diferentes sistemas operacionais, mostrando qual opção mais se aplica a este trabalho, que no caso deste trabalho, foi o Linux, por ser mais acessível que o macOS e por possuir um desempenho superior ao do Windows, além de mostrar tipos de testes que podem ser realizados para verificar o desempenho e quais métricas utilizar.

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

Conforme Wen et. al. (2024), o artigo O trabalho de Wen et al. (2024) aborda a arquitetura baseada em microserviços para sistemas automotivos autônomos usando ROS 2. Cada serviço foi realizado executado 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. Neste projeto foram utilizadas duas plataformas de computação distintas, uma com arquitetura x86 (InoNet Mayflower-B17) e outra aarch64 (Armv8 ADLINK AVA COM-HPC), com ambas utilizando o sistema operacional Ubuntu 22.04.3 LTS Jammy Jellyfish, GPU NVIDIA RTX A6000 48 GB e Kernel 6.2.0-34-generic. Foi utilizado o ROS 2 Humble Hawksbill com o middleware Eclipse CycloneDDS e, por conta de utilizar esta versão do ROS 2, foi utilizado o Autoware baseado nessa versão também. Utilizou-se o Docker engine (versão 24.0.5) para fazer a conteinerização e o k3s (versão v1.27.3+k3s1) para orquestrar os contêineres. Foram realizados alguns testes, sendo eles a avaliação de desempenho da comunicação DDS que verifica o impacto que o desempenho sofre por conta da contêinerização na comunicação do DDS. O teste consistia em verificar a comunicação entre dois nós utilizando o DDS padrão e verificando o tempo de ida e de volta das mensagens e a taxa de entrega de pacotes, sendo as métricas utilizadas. O segundo teste era de verificar o desempenho do ROS 2 com o ros2_benchmark, verificando o impacto da contêinerização sobre o desempenho das aplicações do ROS 2. O teste verificava a comunicação e processamento de dados, usando como métrica a latência de ponta a ponta, o jitter e o quanto da CPU foi utilizado nos diferentes cenários. Após esses testes, foi verificada a aplicação de direção autônoma com o Autoware, onde foi avaliado como a conteinerização afeta o desempenho de uma aplicação de direção autônoma real. Neste teste, o Autoware foi dividido em oito contêineres que isolavam módulos funcionais diferentes. Com isso, foi simulado um trajeto em Nishi-Shinjuku, localizada em Tóquio, em um ambiente

simulado da AWSIM, para medir como a conteinerização era impactada pela latência ponta a ponta. Foi registrado o uso de CPU e de memória como métricas de comparação de impacto entre as configurações. 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 ter um desempenho melhor, 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 Este artigo demostrou que o uso de containeres para o tipo de problema proposto neste trabalho é viável e que pode trazer benefícios em relação a execução em bare-metal.

4 METODOLOGIA

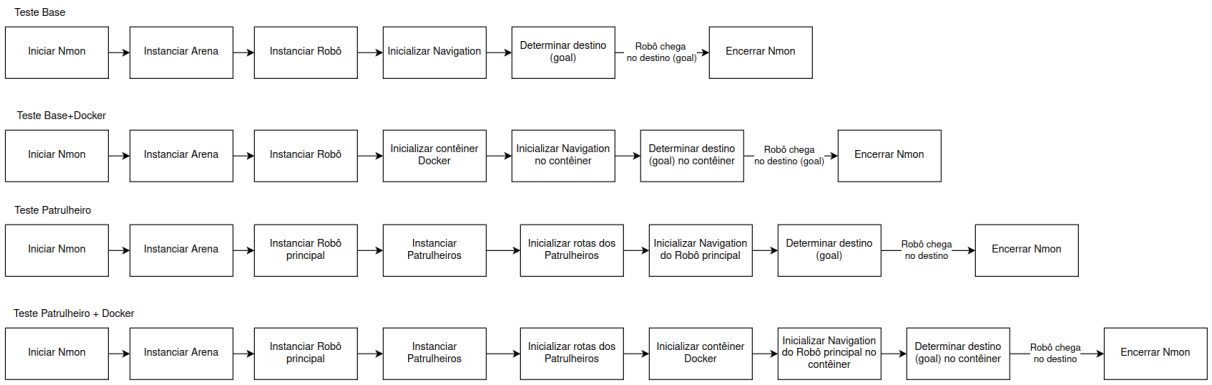
Foi utilizado ROS 2 Humble, Docker Engine Jammy 22.04 Stable e Gazebo Classic. A escolha das versões ROS e Docker Engine foi por conta de serem versões Long Term Support (LTS), enquanto a escolha do Gazebo Classic se deu por conta de todos os recursos já disponíveis para esta versão de Gazebo, especificamente 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. O 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) Para facilitar a execução dos testes, foram criados scripts bash para a execução dos testes, desde scripts para executar a movimentação do robô na arena como também scripts para iniciar e encerrar o nmon. Esses scripts estão disponíveis no GitHub do projeto[2]

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

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

para facilitar o desenvolvimento dos testes e as capturas dos dados. 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 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 Cada teste foi realizado cerca de 30 vezes, nestes testes o robô era posicionado em uma das pontas da arena e devia alcançar a outra ponta, por possuir um SLAM pré-feito da arena não se perderia e conseguiria alcançar o objetivo selecionado, antes do robô percorrer o trajeto, o Nmon era inicializado e armazenava os dados de CPU, memória RAM e rede em uma pasta para assim ser possível obter e comparar os dados, estes testes foram feitos novamente, mas com o diferencial de que o robô estava utilizando o Docker com o Nav2 e o RVIZ inseridos no mesmo, este teste foi realizado tanto virtualmente na arena simulada no Gazebo Classic quanto na arena física. Foram realizar testes que foram feitos apenas virtualmente estes testes foram feitos com uma arena que possuía robôs patrulheiros que funcionavam como o obstáculo para o robô que devia realizar o mesmo trajeto. Foram realizados testes em que o robô realizaria o mesmo objetivo Outro teste realizado apenas virtualmente foi o teste 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

Este capítulo apresenta os experimentos realizados que utilizam os conceitos apresentados no capítulo anterior.

5.1 EXPERIMENTOS EM SIMULAÇÃO

Os experimentos usam um modelo digital da arena física presente no laboratório da sala K4-04 da instituição e usam um robô TurtleBot3 Burger simulado disponível no ROS e Gazebo. O motivo de esta arena combinação ter sido escolhida se deu por conta da comparação que futuramente pode ser realizada entre os testes virtuais e reais.

Foram feitos cinco propostos sete 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.No 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.

Os sete experimentos propostos foram:

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

a) Setup: testes iniciais para confirmar o comportamento do ROS 2 dentro de um contêiner Docker e a viabilidade do projeto proposto.

b) Arena: navegação do robô de um lado a outro na arena da sala K404 sem nenhum obstáculo, conforme o caminho exibido na Figura 5a.

c) Patrulheiros: usa o mesmo mapa do experimento anterior, porém 3 robôs TurtleBot3 Burger foram adicionados para realizar um percurso pré-definido (Figura 5b) com o objetivo de atrapalhar a movimentação do robô que faz a navegação na arena. Os robôs patrulheiros faziam um trajeto em linha reta onde saiam e retornavam aos seus pontos de origem.

d) Arena com Docker: Este experimento é idêntico ao experimento Arena, porém o robô executa a parte do ROS de dentro de um contêiner Docker.

e) Patrulheiros com Docker: Assim como o anterior, este experimento repete o que foi realizado no experimento Patrulheiros mas com o robô usando o ROS em um contêiner.

f) Labirintos: dois mapas com paredes (Figuras 5c e 5d) foram criados para verificar se ocorre alguma alteração no desempenho do robô em relação a navegação. O uso destas paredes se devem ao fato da arena física presente na K4-04 e os mapas desenhados respeitam o espaço da arena, a quantidade de placas disponíveis e as possíveis posições que estas podem ser adicionadas na arena.

g) Labirintos com Docker: assim com os anteriores, este experimento replica o que foi proposto no Labirintos, mas com o robô executando o ROS de dentro do contêiner.

Para todos os experimentos as métricas de uso de CPU, RAM e rede foram medidas usando o nmon e armazenadas em arquivos para que os resultados fossem analisados após a execução de todos os testes. Cada experimento foi executado pelo menos 30 vezes para garantir que a amostragem dos resultados é estatisticamente relevante.

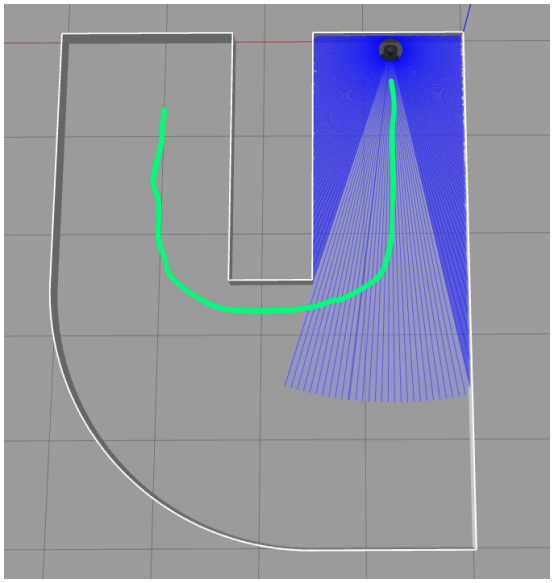
5.2 CONTEÚDO REPOSITÓRIO

O repositório GitHub[2] criado para este projeto possui várias ferramentas que podem ser utilizadas para facilitar a reprodução dos testes executados. O arquivo `README.md` contém um overview de como instalar o projeto e utilizar os scripts presentes no projeto. Foi criado

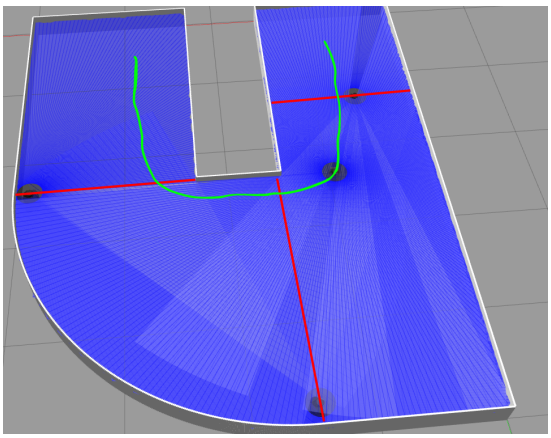
[2]Disponível em https://github.com/joca2511/TCC_Docker

Figura 5 – Modelos de arena usadas para os experimentos simulados.

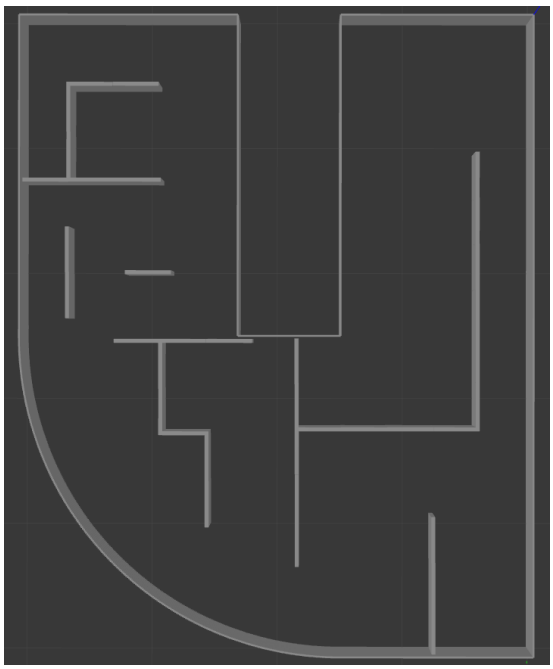
(a) Arena base com robô TurtleBot3 Burger. A linha verde representa o trajeto que deve ser realizado pelo robô.



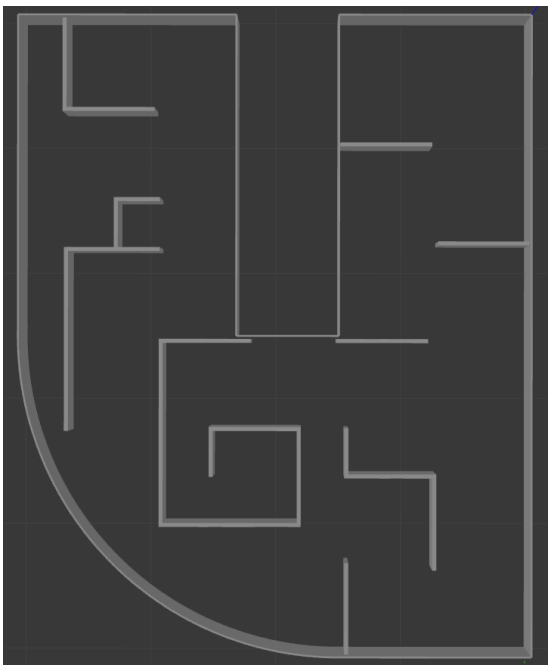
(b) Arena Patrulheiros com 4 robôs TurtleBot3 Burger. As retas em vermelho são os trajetos realizados pelos patrulheiros, enquanto a curva em verde, representa o trajeto do robô.



(c) Labirinto 1. Labirinto feito adicionando paredes à arena da K404.



(d) Labirinto 2. Segunda arena montada para testes com obstáculos.



Fonte: Autores

um playbook ansible (arquivo `playbook.yaml`) para facilitar a configuração e instalação do Docker, ROS 2 Humble, Gazebo Classic e suas dependências. Foram também criados vários scripts para shell (arquivos com extensão `.sh`) para facilitar a reprodução dos testes, para garantir que os comandos corretos serão executados na sequência correta, sem necessidade de

intervenção do usuário, onde os arquivos `inicioRapido*.sh` possuem os comandos utilizados para cada um dos testes feitos para este projeto. Os arquivos com extensão `.sh` na pasta `/scripts` possuem funcionalidades genéricas criadas para os testes, como a inicialização do nmon (script `iniciarNmon.sh`), mover o robô ao inicializar rviz e mandar uma mensagem de goal (`moverMain.sh`), entre outras. No pacote ROS 2 criado (localizado na pasta /tcc/tcc) possui o arquivo `turtlebot3_absolute_move_Arena.py` que é uma versão modificada do arquivo `turtlebot3_absolute_move.py`, 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 do arquivo `turtlebot3_world.launch.py` modificado em `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.

5.3 TESTES ARENA

Foram feitos testes para adquirir os dados de desempenho das simulações sem Docker, utilizando a arena base sem patrulheiros (Figura 5a). 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.

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 5b) 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.

6 RESULTADOS

Este capítulo apresenta os resultados obtidos para os experimentos propostos no capítulo anterior. Para facilitar a comparação, as seções apresentam os resultados dos experimentos com e sem Docker em conjunto.

6.1 EXPERIMENTO SETUP

Foram feitos testes para implementar a parte simulada proposta, foi utilizado o manual (ROBOTIS, 2025) e pacotes[1] disponíveis pelo grupo ROBOTISOs testes foram implementados usando o material disponível por ROBOTIS (2025) junto com pacotes disponobilizados no repositório oficial da ROBOTIS[2], 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 dos contêineres 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.

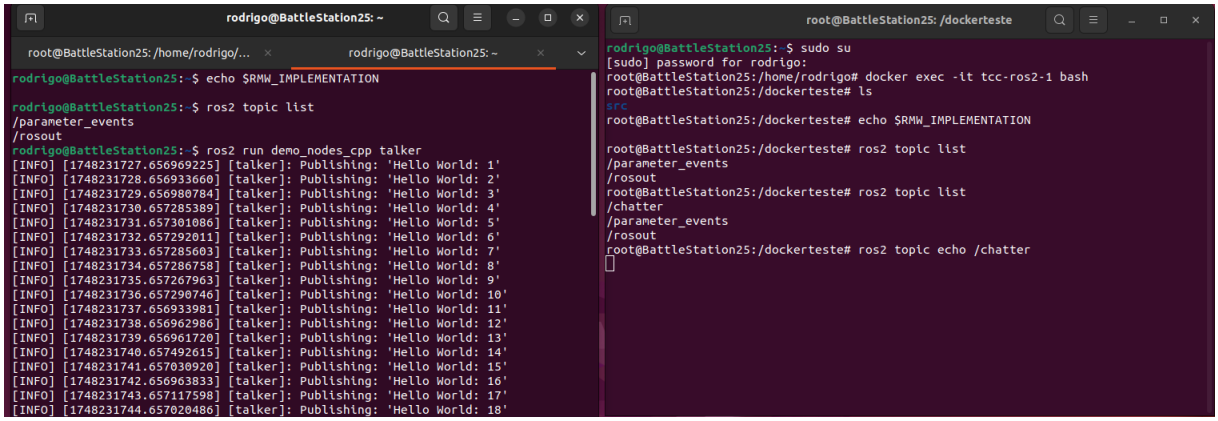
Houve certas dificuldades para integrar o ROS 2 dentro do contêiner Docker com o ROS 2 nativo, que lidaria com a simulação Gazebo. Foi descoberto que o FastDDS (middleware utilizado por padrão pelo ROS 2 Humble) não interage de forma consistente com Docker. Ele era capaz de compartilhar os tópicos entre os ambientes, mas causava falha na publicação e recebimento de mensagens, não mostrando nenhuma. O problema encontrado foi no uso do FastDDS (middleware utilizado por padrão no ROS 2 Humble) que não interagia de forma consistente com o Docker, conforme exibido na Figura 6a, em que o terminal à esquerda mostra o ROS sendo executado no host, enquanto o terminal à direita mostra o ROS sendo executado de dentro do Docker. Enquanto o host consegue enviar as mensagens, o contêiner consegue interagir com o host.

Para solucionar isso, o FastDDS foi substituído por outro middleware disponível para ROS 2 Humble, CycloneDDS, o qual foi utilizado especificamente no container Docker. Como mostra a Figura 6b, o CycloneDDS fez a troca de mensagens sem problemas.

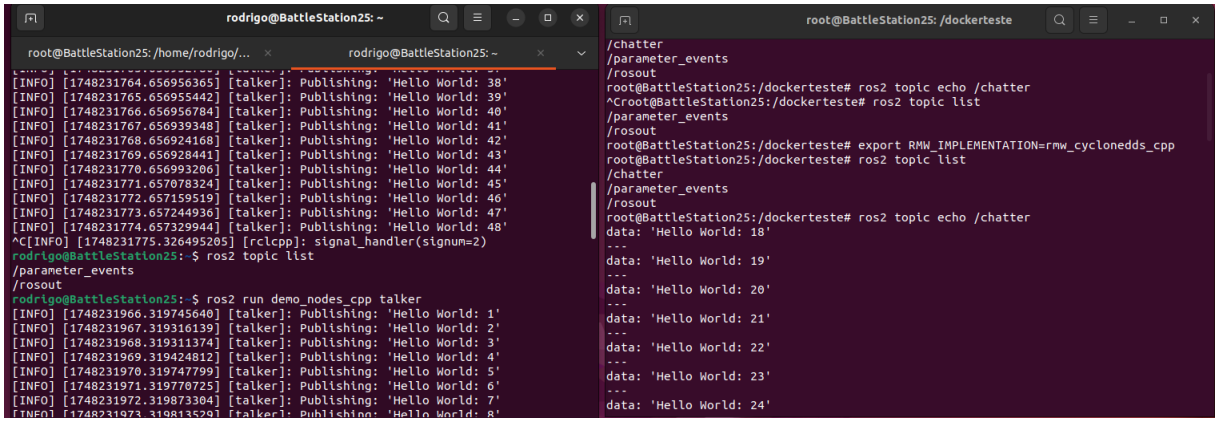
[1] (https://github.com/ROBOTIS-GIT/turtlebot3)
[2] O spacotes estão disponíveis em https://github.com/ROBOTIS-GIT/turtlebot3

Figura 6 – Comparação de troca de mensagens usando FastDDS e CycloneDDS.

(a) Falha no teste usando o FastDDS como middleware



(b) Sucesso no teste com CycloneDDS como middleware



Fonte: Autores

6.2 EXPERIMENTO ARENA

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. No gráfico de CPU (Figura 7a), pode-se ver que a mesma acabou por ser mais exigida quando não se utilizava o Docker, por volta de 1% da média de uso. Nos gráfico de memória RAM (Figura 7c), 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 7e), 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 gráficos da Figura 7 mostram os valores medidos pelo nmon para os experimentos com e sem Docker para as métricas de uso de CPU, RAM e rede no experimento Arena.

Os resultados obtidos mostram que o uso de CPU (Figura 7a) é em média por volta de 1% maior nos experimentos sem Docker quando comparado com o uso do Docker. Apesar desta diferença ser pequena, o Teste T (Figura 7b) mostra que as medidas é estatisticamente diferentes quando usamos um valor-p< 0.05, sendo consistente com os resultados obtidos por Wen et al. (2024).

Nos resultados de uso de RAM (Figura 7c) pode-se ver que a diferença do uso é de cerca de 500MB a mais quando se usa o Docker. Este aumento ocorreu por conta da necessidade do Docker de executar parte do sistema operacional dentro do contêiner, duplicando alguns processos que não podem ser utilizados de forma compartilhada com o host. Da mesma forma, o resultado do Teste T (Figura 7d) mostra que as amostragens são diferentes e existe diferença no uso do recurso.

No gráfico de rede (Figura 7e), 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. Portanto, este aumento no uso de rede era esperado por conta da forma como o ROS funciona e, conforme exibido na Figura 7f, as amostragens são estatisticamente diferentes.

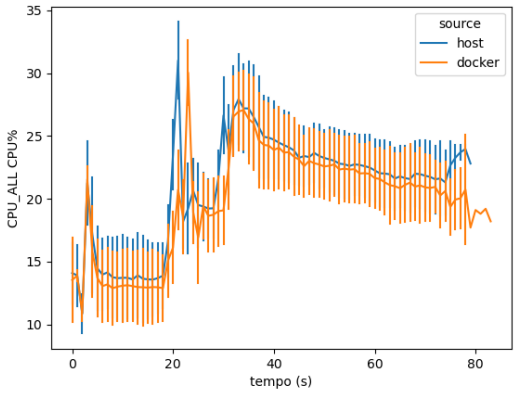
Portanto, considerando os resultados obtidos neste experimento, é possível concluir que o uso de Docker gera o aumento do uso de RAM e de rede, mas este era um resultado esperado por conta da forma como o Docker e ROS funcionam, e pode levar a uma pequena diminuição do uso de CPU, sendo consistente com o que foi encontrado durante a revisão bibliográfica.

6.3 EXPERIMENTO PATRULHEIROS

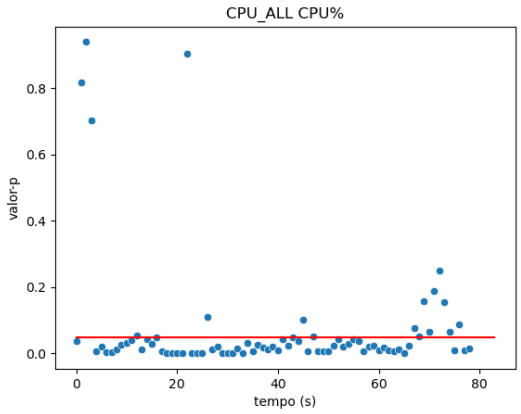
Os testes nas arenas com patrulheiros se demonstraram muito semelhantes aos testes das arenas bases em termos de gráficos, mas os mesmos utilizaram bem mais recursos do sistema. Nos gráficos de CPU (Figura 8a), pode-se ver que foi exigido bem mais da CPU do sistema. Enquanto na arena base era exigido 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 8c) 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

Figura 7 – Resultados obtidos no experimento Arena e Arena com Docker

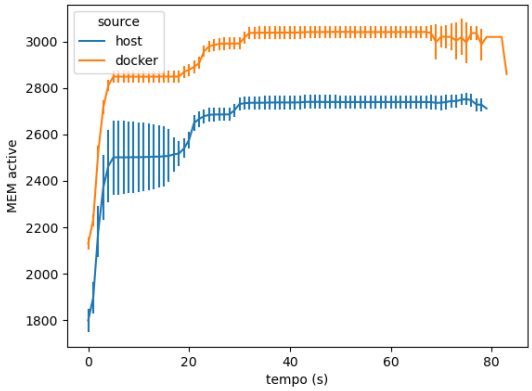
(a) Uso de CPU



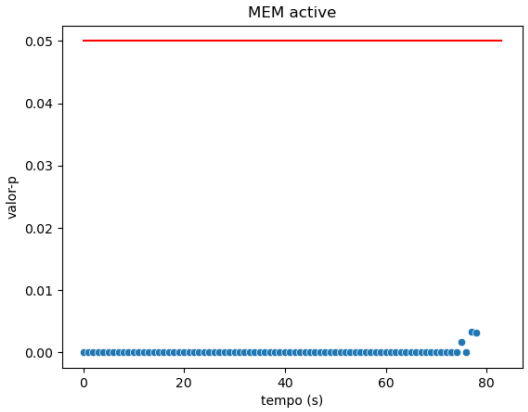
(b) Teste T para o uso de CPU



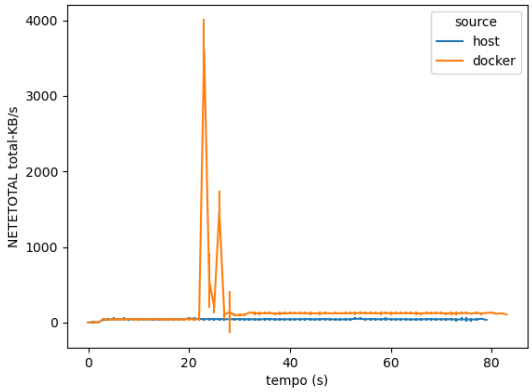
(c) Uso de RAM



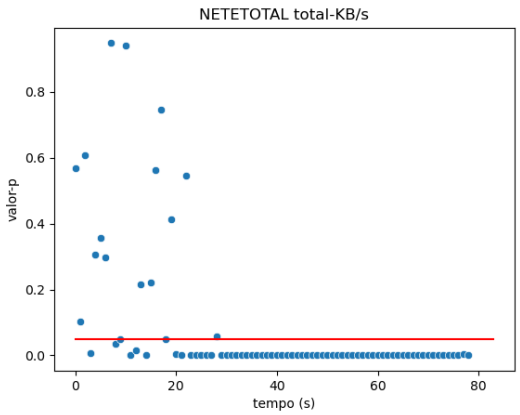
(d) Teste T para uso de RAM



(e) Uso de rede



(f) Teste T para uso de rede



memória no geral do teste. Nos testes de rede (Figura 8e), 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.

Os resultados obtidos no experimento dos patrulheiros apresentam resultados muito semelhantes aos resultados do experimento anterior, porém utilizando mais recursos do sistema. Este aumento do uso de recursos possivelmente ocorreu pelo fato da simulação ter 3 robôs adicionais quando comparado ao experimento anterior, o que demanda maior uso de CPU e RAM pela simulação.

A Figura 8a apresenta as medidas obtidas para o uso de CPU neste experimento e pode-se ver que foi exigido bem mais da CPU. Enquanto no experimento anterior era exigido até 30% da CPU, com os patrulheiros foi exigido 50% e neste experimento o uso Docker exigiu um pouco mais do sistema quando comparado ao teste sem Docker. Assim como no anterior, o Teste T (Figura 8b) mostra que apesar das medidas serem próximas, elas ainda são estatisticamente diferentes.

Os resultados obtidos para o uso de RAM (Figura 8c) se pode ver que se diferenciaram no uso de cerca de 500MB a mais no uso do Docker, semelhante ao experimento anterior e dentro do resultado experado, como descrito na análise do experimento anterior. Da mesma forma, o Teste T (Figura 8d) demonstra que existe diferença entre os valores medidos pelo nmon.

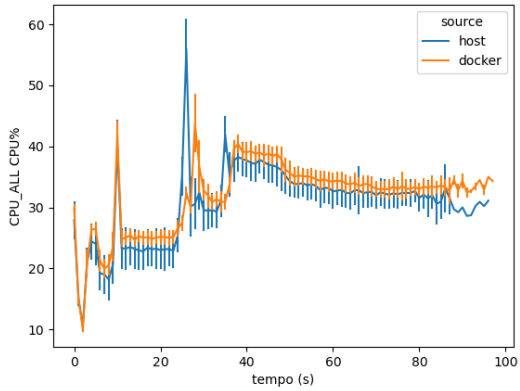
Assim como nos resultados de uso de RAM, o uso da rede (Figura 8e) segue o mesmo padrão do experimento anterior e com os resultados dentro do esperado. Pode-se ver que o uso do Docker aumenta o uso de rede, tendo um pico de uso muito alto entre 20 e 40 segundos de experimento, que logo diminuiu, mas que ainda ficou mais acima dos testes sem Docker. O Teste T para este experimento (Figura 8f) mostra que os resultados estão perto do esperado, mas que no início do experimento (quando o tempo é menor que 20 segundos) existem momentos em que o uso de rede do Docker é praticamente nulo, ficando igual ao uso sem o Docker.

6.4 TESTES LABIRINTOS

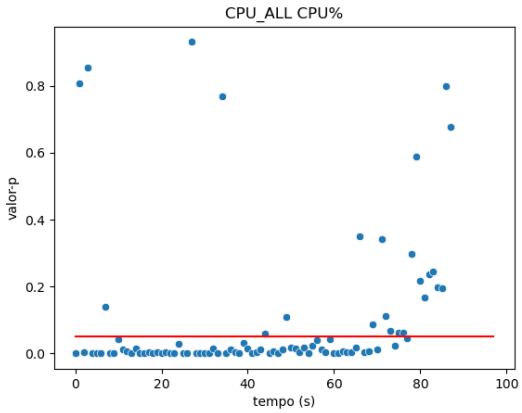
Inicialmente, foram realizados testes sem patrulheiros, com e sem o uso de Docker, em duas arenas com labirintos (Figuras 5c e 5d) como obstáculo para o robô. Os labirintos foram desenvolvidos utilizando placas, que são os obstáculos presentes na arena física da K4-04, respeitando o espaço da arena, 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

Figura 8 – Resultados obtidos no experimento Patrulheiros e Patrulheiros com Docker

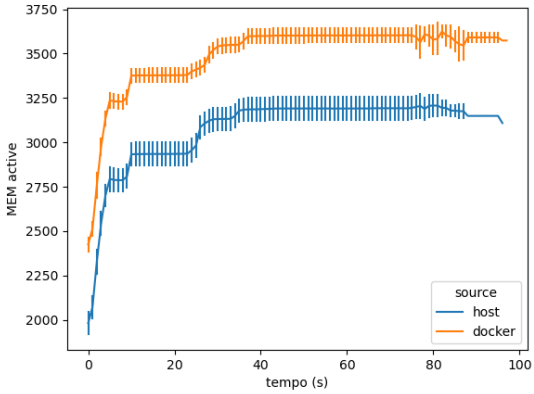
(a) Uso de CPU



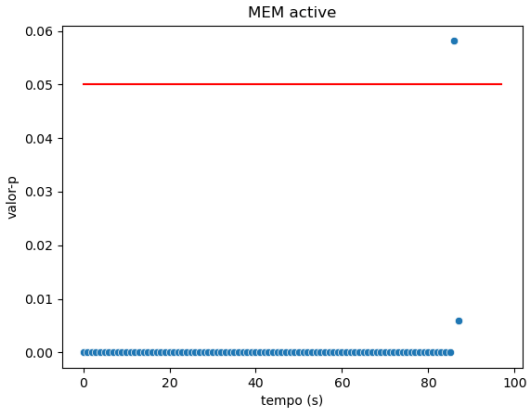
(b) Teste T do uso de CPU



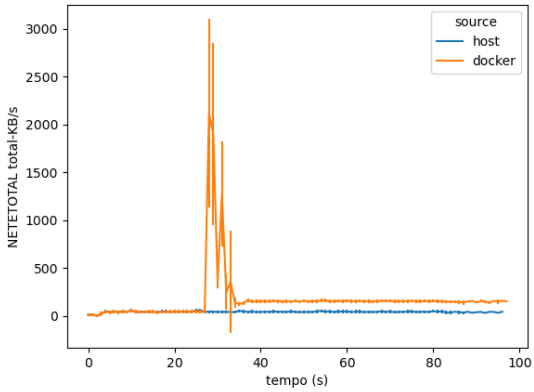
(c) Uso de RAM



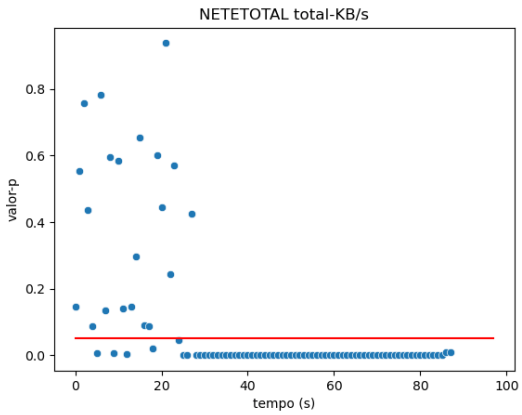
(d) Teste T do uso de RAM



(e) Uso de rede



(f) Teste T do uso da rede



disso, os testes com os patrulheiros não tiveram resultados consistentes em relação as métricas que seriam medidas.

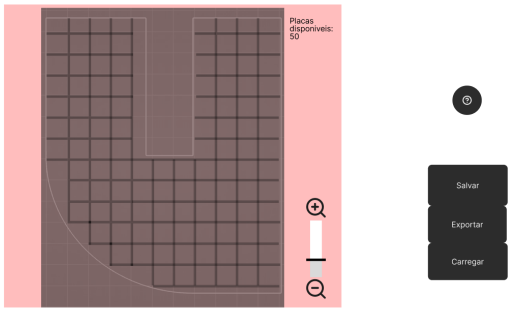
O último teste proposto usava duas arenas sem outros robôs, mas com paredes usadas para formar labirintos (Figuras 5c e 5d) que poderiam ser montados na arena real da K4-04. Por conta de problemas na navegação que por vezes forçavam a recuperação entre 4 e 12 vezes, os testes falharam e não foi possível obter as medidas com o nmon para este experimentos. Porém, os problemas ocorreram tanto na execução dos experimentos sem e com o Docker de forma consistente de forma que o problema em si não foi no ambiente de execução (com Docker ou sem), mas na interação entre o ROS e o Gazebo. Portanto, mesmo não obtendo resultados possíveis de serem analisados, o experimento demonstra que mesmo em casos de falhas na simulação o comportamento do ROS com e sem Docker permaneceu o mesmo.

6.5 INTERFACE PARA PRODUZIR ARENAS PARA A ARENA K4-04 (IHC)

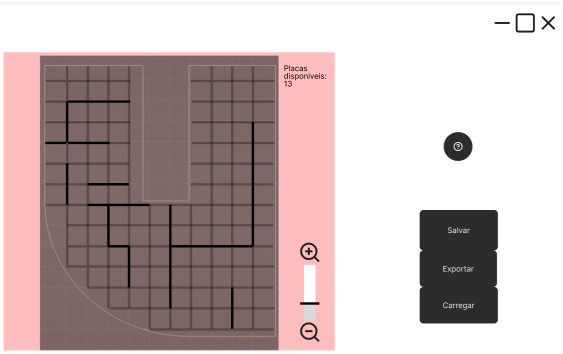
Durante as aulas da matéria de CC8122 - Interface Humano-Computador, foi desenvolvido um protótipo da interface que possui a função de permitir que um usuário desenvolva as próprias arenas para a sala K4-04. A interface é uma imagem da versão simulada da arena da K4-04. Esta imagem possui botões que correspondem às posições onde se é possível posicionar placas que funcionam como paredes para montar labirintos na arena, facilitando o desenvolvimento de arenas para serem usadas em testes ou para se desenvolver as próprias simulações para então executar na arena física.[3]

Figura 9 – Interface do projeto desenvolvida em IHC

(a) Protótipo da interface feita no FIGMA



(b) Exemplo de arena desenvolvida na interface



Fonte: Autores

[3] Todas as informações da interface estão no <https://github.com/mdarce765/ProjetoIHC>

7 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. 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. foi observado que o uso do Docker afeta o desempenho do robô, tanto simulado quanto real. Os testes foram realizados primariamente na simulação por conta de problemas envolvendo o acesso à sala K4-04 e seus equipamentos que nos auxiliariam com os testes. Com isso, pode-se concluir, por meio das análises do consumo da CPU, memória RAM e tráfego de rede, obtidos pelo Nmon, que o Docker apresenta diferenças de desempenho, principalmente no uso do tráfego de rede, mas que o mesmo não impacta de maneira a inutilizar o uso do robô. Para a parte física, foi realizado o teste apenas na arena base, os testes constataram a diferença mínima de consumo, exigindo bem mais na inicialização do teste, mas que em seguida se estabiliza e conclui a tarefa exigida. Com isso, foi concluído que o Docker pode ser utilizado, mas pode ter alguns problemas, pois podendo apresentar alguns problemas, devido a não foram realizados terem sido 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. Mesmo com alguns possíveis problemas, o Docker pode ser implementado de maneira segura para realizar testes simulados ou práticos com o ROS 2. O uso de contêineres gera praticidade e maior portabilidade ao usuário. Com os dados obtidos, foi observado que o usuário pode utilizar sem preocupações por conta do baixo impacto sofrido pelo uso do Docker.

7.1 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.

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.

QNX. **Optimizing Robotic Precision: Unleash Real-Time Performance With Advanced Foundational Software Solutions**. 2024. Accessed: 2025-12-10. Disponível em: <https://www.automate.org/robotics/tech-papers/optimizing-robotic-precision-unleash-real-time-performance-with-advanced-foundational-software-soluti

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>.