INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO PIAUÍ
CAMPUS TERESINA CENTRAL
TECNÓLOGO EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

DIEGO DE SOUSA ARAÚJO

DESENVOLVIMENTO DE UMA INTERFACE WEB PARA VISUALIZAÇÃO DOS DADOS DE UM ROBÔ ASPIRADOR

TERESINA, PIAUÍ
2025

DIEGO DE SOUSA ARAÚJO

DESENVOLVIMENTO DE UMA INTERFACE WEB PARA VISUALIZAÇÃO DOS DADOS DE UM ROBÔ ASPIRADOR

Trabalho de Conclusão de Curso (Relatório Técnico de Software) apresentado como exigência parcial para obtenção do diploma do Curso de Tecnólogo em Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia do Piauí, Campus Teresina Central.

Orientador: Prof. M^{e}. Francisco Marcelino Almeida de Araújo

TERESINA, PIAUÍ
2025

Dados Internacionais de Catalogação na Publicação (CIP) de acordo com ISBD

Araújo, Diego de Sousa

A658d Desenvolvimento de uma interface web para visualização dos dados de um robô aspirador / Diego de Sousa Araújo. - 2025.

65 f.: il. color.

Trabalho de Conclusão de Curso (Tecnologia em Análise e Desenvolvimento de Sistemas) - Instituto Federal de Educação, Ciência e Tecnologia do Piauí, Campus Teresina Central, 2025.

Orientador : Prof Me. Francisco Marcelino Almeida de Araújo.

1. ROS . 2. React JS . 3. WebSocket . 4. Fastify . I.Título.

CDD - 004.21

Elaborado por Maria Rosismar Farias CRB 3/631

DIEGO DE SOUSA ARAÚJO

DESENVOLVIMENTO DE UMA INTERFACE WEB PARA VISUALIZAÇÃO DOS DADOS DE UM ROBÔ ASPIRADOR

Trabalho de Conclusão de Curso (Relatório Técnico de Software) apresentado como exigência parcial para obtenção do diploma do Curso de Tecnólogo em Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia do Piauí, Campus Teresina Central.

Aprovada em: 16/12/2025

BANCA EXAMINADORA:

Prof. Me. Francisco Marcelino Almeida de Araújo
Instituto Federal de Educação, Ciência e Tecnologia do Piauí - IFPI

Prof. Me. Fernando Castelo Branco Gonçalves Santana
Instituto Federal de Educação, Ciência e Tecnologia do Piauí - IFPI

Prof. Me. Wilson de Oliveira Junior
Instituto Federal de Educação, Ciência e Tecnologia do Piauí - IFPI

Matheus Araujo Dantas
Instituto Federal de Educação, Ciência e Tecnologia do Piauí - IFPI

À minha família

Aos meus educadores.

A todos que seguiram comigo nessa trajetória.

AGRADECIMENTOS

A todas as pessoas que me acompanharam até aqui, eu agradeço profundamente pelo apoio, sem ele chegar até a esta etapa da minha jornada acadêmica não seria possível.

Agradeço aos meus amigos pelas boas conversas nos momentos de descontração e os conselhos dados nos momentos sérios.

Aos meus professores cada momento de dúvida tirada, agradeço toda tentativa de me instruir nesse caminho acadêmico, sem as suas orientações o caminho seria muito mais difícil. Agradeço especialmente ao professor Francisco Marcelino por todas as oportunidades e momentos a que ele me expôs, todos eles foram momentos de aprendizagem.

Aos meus pais eu agradeço todo o apoio durante esse período, todo o conforto, estrutura e atenção que me foi dado, eu agradeço de coração.

A minha namorada eu agradeço por estar ao meu lado nesse período e por sempre me fazer acreditar que daria tempo de terminar.

“A maioria dos bons programadores faz programação
não porque eles esperam ser pagos
ou obter adulação pelo público,
mas porque é divertido programar”.
Linus Torvald

RESUMO

Com o avanço das tecnologias de automação e a crescente integração de robôs em diferentes setores, este trabalho busca unir duas ferramentas essenciais em seus respectivos domínios: ROS (Robot Operating System) e React JS. O objetivo é desenvolver uma interface web para visualização dos dados de um robô aspirador iRobot Roomba® 614, permitindo o monitoramento em tempo real dos sensores do dispositivo. Para isso, foram utilizadas a biblioteca de código aberto ROS, que conta com um ecossistema para comunicação entre sensores e sistemas. Outrossim, o uso do React garante uma interface modular, responsiva e focada no usuário. A integração entre ROS e React foi realizada por meio do WebSocket, utilizando bibliotecas da Rosbridge Web Tools para converter e transmitir os dados dos sensores em formato JSON. Ademais, um banco de dados PostgreSQL, acessado através de uma API Fastify, possibilitou a criação de um histórico de uso do robô.

Palavras-chave: ROS, React, WebSocket, Fastify, iRobot Roomba.

ABSTRACT

With the advance of automation technologies and the growing integration of robots in different sectors, this work seeks to unite two essential tools in their respective domains: ROS (Robot Operating System) and React JS. The aim is to develop a web interface for visualizing the data of an iRobot Roomba® 614 vacuum cleaner robot, enabling real-time monitoring of the device's sensors. To do this, the open source ROS library was used, which has an ecosystem for communication between sensors and systems. Furthermore, the use of React guarantees a modular, responsive and user-focused interface. The integration between ROS and React was carried out through WebSocket, using libraries from Rosbridge Web Tools to convert and transmit sensor data in JSON format. In addition, a PostgreSQL database, accessed via a Fastify API, made it possible to create a history of the robot's use.

Keywords: ROS, React, WebSocket, Fastify, iRobot Roomba.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

ROS	*Robot Operating System*
API	*Application Programming Interface*
UML	*Unified Modeling Language*
HTML	*HyperText Markup Language*
CSS	*Cascading Style Sheets*
JS	*JavaScript*
DOM	*Document Object Model*
JSON	*JavaScript Object Notation*
LABIRAS	*Laboratory of Intelligent Robotics, Automation and Systems*
IANA	*Internet Assigned Numbers Authority*

LISTA DE SÍMBOLOS

®	Marca registrada

SUMÁRIO

1 INTRODUÇÃO

A interação humano-computador é uma importante área de estudo que se preocupa em estudar o design, avaliação e implementação de sistemas para uso humano. Durante as interações '"humano + máquina"' interfaces de usuário funcionam como uma ponte entre os seres humanos e os computadores, de maneira que os usuários processam ou interpretam os dados (LIU; OSVALDER; KARLSSON, 2010). Assim, é imperativo o uso de interfaces gráficas na construção de um software moderno.

As interfaces bem projetadas reduzem a carga cognitiva do usuário e aumentam a confiança, satisfação e aceitação por parte dos usuários, sendo um fator decisivo para a usabilidade efetiva de qualquer sistema digital (ISRAEL; TOBILOBA, 2025; ROLLIN et al., 2025). Segundo, (ROLLIN et al., 2025), mesmo na presença de grandes volumes de dados, *designs* esteticamente atrativos moldam o comportamento do usuário quanto ao produto, influenciando positivamente suas preferências, percepção e avaliações.

Nesse contexto, ferramentas como dashboards compostos por gráficos e estruturas visuais são opções eficientes para visualizar dados em tempo real. Esses painéis permitem integrar e apresentar múltiplas fontes de informação em layouts estruturados, geralmente baseados em grids, facilitando a apresentação de dados por meio de visualizações coesas e intuitivas. Outrossim, estudos mostram que esse tipo de visualização reduz significativamente a sobrecarga cognitiva, acelera a compreensão das informações e contribui diretamente para decisões mais ágeis e assertivas por parte de usuários técnicos e não técnicos (SULTANA, 2025; DOSUNMU et al., 2025).

Dentro da robótica, robôs podem ser definidos como mecanismos atuadores programados com um grau de autonomia para realizar locomoção, manipulação ou posicionamento, possuindo ainda sistemas de controle (International Organization for Standardization, 2024). Com base nessa definição, entende-se que essas estruturas são ideais para integração com sistemas de coleta e visualização de dados, pois os sensores e atuadores geram dados continuamente sobre si e o ambiente ao seu redor. Logo, um sistema visual bem estruturado atua como uma ponte entre os dados brutos produzidos pelo robô e as ações humanas baseadas nesses dados, a exemplo de robôs móveis de uso doméstico ou laboratorial, como o aspirador iRobot Roomba.

O iRobot Roomba® 614 possui um conjunto robusto de funcionalidades e uma construção que conta com uma arquitetura de alta confiabilidade. Além disso, o robô traz consigo maneiras de permitir o controle de suas funcionalidades motoras e o acesso aos dados de seus sensores.

Perante o exposto, o presente trabalho apresenta a construção de um sistema de monitoramento dos dados de um robô aspirador de pó (iRobot Roomba® 614). A aplicação *web* permite a visualização dos dados do robô, controle do mesmo e

verificação do status de conexão com a máquina.

Para isso, o sistema possui uma interface gráfica construída com a biblioteca React, faz o uso do Sistema Operacional Robótico (ROS) para atuar como ponte entre o hardware dos sensores do dispositivo robótico e a camada de apresentação. Além disso, a aplicação conta com uma API desenvolvida com Fastify, responsável por intermediar o registro e comunicação com os serviços de banco de dados. Portanto, permitindo que os dados do robô possam ser compartilhados, visualizados e gerenciados em tempo real, mediante um sistema multiplataforma.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Desenvolver um software *web* para visualização em tempo real dos dados dos sensores de um robô móvel iRobot Roomba® 614, utilizando o ROS como meio de comunicação para o hardware do robô.

1.1.2 Objetivos Específicos

- Construir uma interface de comunicação para todos os diversos sensores e atuadores do robô aspirador, utilizando as ferramentas do ROS.
- Criar um serviço *web* de comunicação para acesso da interface *web* ao banco de dados.
- Criar interfaces gráficas do lado do cliente para comunicação, visualização e manipulação dos dados da infraestrutura do sistema robótico.
- Implementar a comunicação entre as interfaces gráficas do lado do cliente com o sistema robótico e a API de comunicação com o serviço *web*.

2 TECNOLOGIAS ENVOLVIDAS

Para completar com sucesso o objetivo do trabalho, a aplicação foi dividida em duas partes, *frontend* e *backend*, ambas fazem a utilização dos serviços do NodeJs (Node) durante o desenvolvimento da aplicação. O *frontend* da aplicação, que faz o uso do NodeJs durante o desenvolvimento, utiliza o ViteJs (Vite) para criar um ambiente com um *bundler* e um servidor de maneira rápida. O *backend* da aplicação, contará com todo o arcabouço de bibliotecas e tecnologias do ROS, juntamente com os serviços da infraestrutura do Node.

Ainda nesse cenário, o Node é uma ferramenta que fornece um ecossistema para execução de código JavaScript. Nesse sentido, com a biblioteca a manipulação da linguagem JavaScript pode ser realizada dentro e fora das fronteiras de um navegador, trazendo consigo funcionalidades, como servidores, conversão de código e acesso a uma série de pacotes facilitadores do desenvolvimento de aplicações, fornecidos através do "npm"(ALURA, 2015).

Além do Node, a reunião de funcionalidades do ROS é a peça chave desse trabalho, com o seu ecossistema robusto de comunicação, ele permite o recebimento, conversão e compartilhamento de dados. Assim, a integração das infraestruturas, NodeJs e ROS, permite que a aplicação cliente gere, em uma interface web, a visualização dos dados processados do robô.

2.1 NODEJS

O NodeJs, mais conhecido somente como Node é uma plataforma de execução JavaScript de desenvolvimento construída em cima do interpretador de JavaScript, o Google Chrome V8, que pode ser considerado a máquina virtual JavaScript, através dele o Node consegue realizar a execução de JavaScript tanto do lado do cliente quanto do lado do servidor (VERMA VISHAL SHRIVASTAVA; SHRIVASTAVA, 2024). Diante disso, no lado do cliente (*frontend*), ele auxilia na criação de interfaces web, por exemplo compilando em servidor local os arquivos JSX, TSX (da biblioteca React) e TypeScript. No lado do servidor (*backend*), para onde ele foi efetivamente criado para ser utilizado, ele se torna uma ferramenta para construção, de servidores web, microsserviços e aplicações da internet das coisas (VERMA VISHAL SHRIVASTAVA; SHRIVASTAVA, 2024).

Atualmente, Node é essencial para o desenvolvimento de aplicações com foco na web contando com um estrutura designada para atender a demandas intensivas de dados, como jogos *multiplayers*, chats ou ambiente colaborativos online. Ele é perfeito para aplicações cujos dados requisitados precisam ser retornados em tempo real, pois

com uma composição *single-threaded* e assíncrona, ele lida muito bem com respostas eficientes à operações (HUANG, 2020).

Outrossim, essa biblioteca se destaca por fornecer serviços para aplicações web que vão além de somente apresentar conteúdo em sites, conta também com uma série de módulos e bibliotecas de código aberto criadas para auxiliar no desenvolvimento eficiente dentro do seu ecossistema. Para tal, o Node possui o seu próprio gerenciador nativo de pacotes, que permite o armazenamento de dependências em diretórios isolados por projeto.

Esse modelo reduz conflitos de versões e facilita o controle das bibliotecas utilizadas no projeto. Esse diferencial se estabelece principalmente a outras linguagens como python, que tendem a instalar pacotes globalmente por padrão, exigindo a configuração de ambientes virtuais para alcançar a mesma modularização das dependências. Essa diferença estrutural contribui significativamente para a previsibilidade no desenvolvimento de aplicações web com Node (GIBB et al., 2025).

2.2 TECNOLOGIAS DO CLIENTE

2.2.1 TypeScript

O *superset* de JavaScript, também conhecida como TypeScript, é uma linguagem de programação que traz consigo todas as capacidades do JavaScript mais um conjunto de ferramentas relacionadas a tipagem dos recursos da linguagem JavaScript. Com TypeScript, todo código criado é transpilado/compilado para JavaScript, permitindo que o código seja executando, no ambientes do navegador, Node e Deno (MICROSFT, 2024).

Além disso, o processo de desenvolvimento de algoritmos utilizando os recursos do TypeScript pode ser muito mais rápido, pois com JavaScript alguns erros de sintaxe e tipos em variáveis e funções somente é possível de ser visualizado após o código ser executado. Nesse sentido, com o suporte da linguagem à inferência de tipos, interfaces, classes e *enums* o código torna-se mais legível e organizado, facilitando a manutenção e acrescentando produtividade (MICROSFT, 2024).

2.2.2 React JS

Uma das mais famosas bibliotecas de *frontend* da comunidade de desenvolvedores. O React, ferramenta criada pelo *Facebook*, defini-se como uma biblioteca de código aberto para construção de interfaces modernas e interativas. A biblioteca, através de sua arquitetura baseada em componentização, rapidamente se tornou uma das favoritas ao desenvolvimento web, devido às facilidades que sua construção trouxe para o desenvolvimento e manutenção (SHAH, 2024).

Ademais, o React, mudou a maneira como as aplicações web são criadas, com o seu uso é possível dividir eficientemente as responsabilidades de cada parte da interface da aplicação (componentes), lida isoladamente com seus estados (RAWAT; MAHAJAN, 2020). Desse modo, esse "autogerenciamento"de cada página, permite que os diversos dados da página possam ser alterados individualmente, guardando a lógica de cada parte da interface em diferentes componentes (RAWAT; MAHAJAN, 2020). Com isso, os componentes da interface podem ser reusados sem a necessidade de repetição de código, acelerando o processo de desenvolvimento da aplicação e facilitando a sua manutenção (GURUNG, 2024).

Para que isso fosse possível, o conceito de Virtual DOM foi criado e implementado no React, esse conceito de programação se baseia em guardar em memória uma representação da árvore do DOM da interface gráfica original. Essa representação, é utilizada como intermediário entre as alterações de estado da página e o DOM. Desse modo, os novos estados alterados somente são alterados na árvore virtualizada (GURUNG, 2024).

Por fim, a biblioteca realiza comparações entre as alterações acumuladas no DOM virtual e o DOM original, e desse modo, passa a fazer pequenas atualizações no DOM original, sem realmente o renderizar novamente (GURUNG, 2024). Essa abordagem, trouxe para o desenvolvimento web a capacidade de criar páginas web mais eficientes e com maior velocidade de carregamento, além de melhorar a experiência do usuário com a aplicação (GURUNG, 2024; SHAH, 2024).

Portanto, a ferramenta de desenvolvimento web, criada em 2018, ainda se faz uma poderosa ferramenta nos dias de hoje, pois, mesmo após anos de sua criação, o React ainda entrega para o desenvolvedor um conjunto de instrumentos que garantem simplicidade, eficácia e performance.

2.2.3 Chakra UI

“Chakra UI é uma biblioteca de componentes simples, modular e acessível que fornece os blocos de construção que você precisa para construir seus aplicativos React.” (UI, 2024).

A ferramenta de estilização traz consigo um poderoso conjunto de ferramentas para construção rápida de uma aplicações. Com uma série de componentes base, previamente estilizados, o Chakra UI, permite um desenvolvimento centralizado na lógica da aplicação sem a preocupação de garantir que os componentes básicos da UI sejam totalmente customizáveis e responsivos. Além disso, todos os componentes fornecidos pela biblioteca adotam normas de acessibilidade, a exemplo dos padrões da WAI-ARIA, seguidos por todos os componentes da biblioteca (UI, 2024).

Ainda, a biblioteca possui um sistema de fácil instalação e implementação dentro do software, permitindo que rapidamente um ambiente de desenvolvimento para

criação de aplicações React seja montado. Portanto, essa ferramenta de fácil utilização garante ao desenvolvedor uma experiência garantida de produtividade durante toda a etapa de estilização da aplicação *React*, acelerando a criação de componentes complexos que sejam responsivos e reutilizáveis nas diferentes partes da UI (ISLAM, 2023; UI, 2024).

2.2.4 RoslibJS

As "Roslibs" são um conjunto de APIs criadas pela *Robot Web Tools*, uma comunidade de bibliotecas e sistemas *open source* que realizam interações entre um sistema web e um sistema robótico construído com o uso de ROS. Dentre essas APIs, existe a RoslibJS, a versão JavaScript para comunicação à rosbridge, essa versão ainda possui suporte à tipagem, permitindo o uso da biblioteca com a inferência de tipos do TypeScript em tempo de compilação. O pacote que pode se encontra nos registros do npm de biblioteca, fornece uma API JSON para comunicação com os sistemas de um robô configurado para se comunicar via ROS. A mesma comunidade, também criou APIs para realizar esse mesmo tipo de comunicação com a Rosbridge em Rust e Python (TORIS, 2023; TOOLS, 2024; TORIS, 2024).

Nesse sentido, com o uso da RoslibJs, é possível se conectar com todo o sistema do robô através do uso do protocolo de comunicação de WebSockets realizando uma conexão ponta a ponta com o servidor rodando internamente nas estruturas do ROS. Com isso, através da API funcionalidades essenciais do ROS passam a ser fornecidas para a aplicação cliente, como publicar, subscrever, realizar chamadas de serviço (TOOLS, 2024).

2.3 TECNOLOGIAS DO SERVIDOR

2.3.1 Fastify

O Fastify é um framework para desenvolvimento web baseado em Node, projetado para oferecer alto desempenho e baixa sobrecarga no processamento de requisições (TEAM, 2025a). Esse framework, comparado a outras soluções populares, como ExpressJs, que também está entre os rápidos frameworks de Node, se destaca por sua arquitetura otimizada, que reduz o tempo de processamento de requisições e melhora a escalabilidade do sistema (NILSSON; DEMIR, 2023). De acordo com estudos comparativos, o Fastify tem um dos menores tempos de resposta entre os frameworks avaliados, garantindo a construção de aplicações robustas sem comprometer a velocidade de execução (DEMASHOV; GOSUDAREV, 2019).

Outrossim, o Fastify se sobressai não apenas no quesito desempenho, mas também na experiência do desenvolvedor, oferecendo suporte nativo a TypeScript, além de um ecossistema modular e extensível de *plugins* (NILSSON; DEMIR, 2023).

Essas características permitem a construção de serviços web de forma mais eficiente e estruturada, permitindo maior produtividade no desenvolvimento. Dessa forma, sua utilização é altamente recomendada para projetos que visam otimizar o consumo de recursos e aprimorar a eficiência na manipulação de requisições HTTP, consolidando-se como uma solução inovadora para arquiteturas modernas (DEMASHOV; GOSUDAREV, 2019).

2.3.2 PostgreSQL

O PostgreSQL (Postgres) é um sistema avançado de gerenciamento de banco de dados relacional (RDBMS), criado inicialmente na Universidade da Califórnia o banco de dados se sobressai pela sua confiabilidade e conformidade com os princípios ACID (Atomicidade, Consistência, Isolamento e Durabilidade) o que aumenta o seu uso em aplicações. Na atualidade, o Postgres evoluiu para uma solução robusta amplamente utilizada no desenvolvimento de inteligência artificial (IA). Dentre as funcionalidades mais importantes desse banco e que eximem a suas capacidades, está o suporte a tipos de dados complexos, técnicas sofisticadas de indexação e controle de concorrência baseado em Multi-Version Concurrency Control (MVCC), garantindo maior eficiência no acesso simultâneo aos dados (WENG; WU, 2024).

Nesse contexto, apesar de ser amplamente utilizado em grandes sistemas e aplicações de alta demanda, o PostgreSQL também se apresenta como uma solução eficiente para aplicações menores, uma vez que oferece baixo custo de manutenção inicial e escalabilidade progressiva. Com isso, aplicações pequenas podem se beneficiar de suas funcionalidades avançadas sem comprometer o desempenho do banco, garantindo uma migração fluida para um banco de dados mais complexo à medida que o sistema cresce. Desse modo, a sua adoção desde as fases iniciais do projeto proporciona uma base sólida para futuras expansões, evitando a necessidade de mudar posteriormente de banco de dados de acordo com o aumento da complexidade do sistema (WENG; WU, 2024).

2.3.3 WebSocket

A conexão via WebSockets é um protocolo de comunicação que trouxe uma revolução na transmissão de dados em tempo real na web. Criado em 2010 pela Internet Engineering Task Force (IETF), o WebSocket foi desenvolvido para superar as limitações dos modelos de comunicação baseados em HTTP, como o polling e o long polling. Esses métodos eram amplamente utilizados, mas tinham limitações importantes para manter uma conexão em tempo real de maneira eficaz. Assim, o WebSocket, que proporciona uma comunicação bidirecional contínua entre cliente e servidor, sem a necessidade de múltiplas requisições HTTP, se tornou uma alternativa

de uso para minimizar os desafios do modelo HTTP para comunicação em tempo real (MARQUES, 2023).

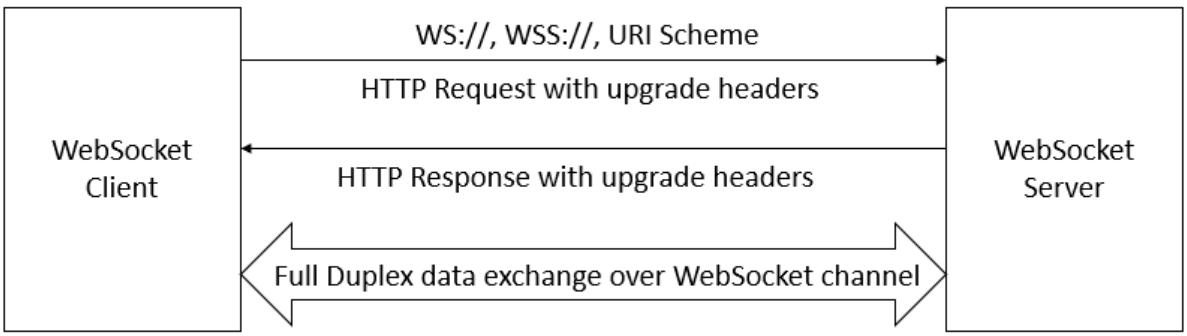
A seguir são descritos os dois modelos tradicionais mencionados no parágrafo anterior:

- Polling: Neste modelo, múltipla requisições HTTP para o servidor são efetuadas. Por consequência, um alto consumo de recursos do servidor é utilizado, mantendo-o em alta atividade o tempo todo, e não necessariamente devolvendo para o cliente do servidor respostas diferentes para cada requisição (NAIK; KHARE, 2020; MARQUES, 2023).
- Long Polling: Neste processo, no caso de um requisição HTTP, o servidor mantém aberta a conexão com o cliente até que novos dados sejam recebido no servidor ou um *timeout* aconteça. Após o servidor enviar a resposta ao cliente, uma nova requisição precisa ser feita novamente. Desse modo, essa solução, embora mais eficaz que o polling, ainda apresenta gargalos de performance e escalabilidade para serem considerados, o que não a torna um protocolo ideal para conexão em tempo real (NAIK; KHARE, 2020).

Diante dessas limitações, o protocolo WebSocket foi criado como parte do HTML5 oferecendo uma solução mais robusta para a comunicação em tempo real em detrimento ao protocolo HTTP (NAIK; KHARE, 2020). Diante disso, para atingir os objetivos esperados no protocolo WebSocket uma conexão entre o cliente e o servidor é criada e persistida, de modo a garantir uma conexão bidirecional e full duplex (NAIK; KHARE, 2020). Para isso, ao invés de múltiplas conexões serem efetuadas cada vez que novos dados sejam necessários, uma única conexão é feita e através dela um processo conhecido como *handshake* é efetuado (MARQUES, 2023).

Durante esse processo, que pode ser visto na figura 1, após a requisição para o servidor, se o suporte ao protocolo *WebSocket* existir, uma conexão *full duplex* é estabelecida e, com isso um fluxo contínuo de mensagens entre o cliente e o servidor é iniciado sem a necessidade de requisições HTTP adicionais (NAIK; KHARE, 2020).

Figura 1 – Procedimento de conexão inicial *WebSocket*



Fonte: (NAIK; KHARE, 2020)

Portanto, com uma conexão em tempo real com o servidor efetivamente estabelecida, a conexão WebSocket reduz a sobrecarga dos servidores, garantindo-o maior eficiência e desempenho através da diminuição do desperdício de recursos, como os associados ao fechamento constante de conexões com o cliente, como visto nos protocolos HTTP. Desse modo, esse recurso oferece uma experiência de uso mais fluida para aplicações com alto tráfego de dados, como chats, jogos online, telemetria em tempo real de robôs ou *streaming* (MARQUES, 2023).

2.3.4 Robot Operating System

ROS (Robot Operating System) é amplamente reconhecido como uma estrutura essencial para o desenvolvimento de sistemas robóticos. Sua arquitetura open source oferece uma série de ferramentas e bibliotecas voltadas para a construção de robôs móveis e manipuladores, abrangendo funcionalidades como navegação, planejamento de movimento, reconhecimento de voz, visão computacional e processamento de sensores (JUNIOR, 2024).

Com mais de 7000 pacotes desenvolvidos para suas diversas distribuições, o ROS se destaca por sua flexibilidade, pois ele fornece ferramentas para adquirir, compilar, escrever e executar códigos em múltiplas plataformas, favorecendo a interoperabilidade entre diferentes sistemas e dispositivos. Desse modo, o ROS atua como um meta-sistema operacional que facilita a abstração de hardware, controle de dispositivos de baixo nível e gerenciamento de pacotes, permitindo a comunicação eficiente entre diferentes softwares (JUNIOR, 2024; LALANCETTE; POUBEL, 2022).

Outrossim, uma das principais capacidades do ROS é sua arquitetura de comunicação entre processos distribuídos, que permite o compartilhamento de dados entre diferentes componentes do sistema robótico. Esses componentes, chamados de nós, são responsáveis por centralizar a computação e processamento de tipo de dados específicos de uma determinada parte do robô. Para que um nó se comunique com outro são utilizados tópicos e serviços, estruturas fundamentais no gerenciamento de dados e ações dentro da arquitetura ROS.

No modelo de tópicos, abstrações nomeadas de *data streams* transmitem dados em um fluxo contínuo, permitindo que nós publicadores (*publishers*) enviem mensagens para um canal, enquanto nós assinantes (*subscribers*) escutam esse canal e recebem as mensagens conforme são publicadas. Essa arquitetura do tipo *publish/subscribe* é assíncrona e altamente escalável, visto que múltiplos nós podem assinar ou publicar em um mesmo tópico sem a necessidade de conhecerem uns aos outros diretamente. Por exemplo, um nó responsável pela leitura de um sensor LiDAR pode publicar os dados em um tópico /scan, enquanto outro nó encarregado da navegação pode assinar esse mesmo tópico e processar os dados em tempo real para desviar de obstáculos (KOUBAA, 2015).

Ademais, no modelo de serviços é utilizado para comunicações do tipo request/reply, onde um nó atua como servidor que disponibiliza um determinado serviço, e outro atua como cliente, que realiza uma requisição e aguarda pela resposta. Essa forma de comunicação é síncrona e mais adequada para operações pontuais, como solicitar uma leitura específica ou executar uma ação única. Cada serviço é definido por um par de mensagens: uma para o pedido (request) e outra para a resposta (reply) (KOUBAA, 2015).

O ROS, inicialmente desenvolvido pela Willow Garage em 2007, uma empresa de robótica e pesquisa, e atualmente mantido pela Open Robotics, é amplamente utilizado nas aplicações industriais, e por isso, é uma ferramenta de importante uso para pesquisa acadêmica, pois está diretamente relacionada ao contexto real do mercado de trabalho. Com uma arquitetura modular, com nós de controle (nodes) distribuídos e fracamente acoplados, possibilita a construção de projetos em formato de módulos independentes. Essa abordagem, segundo Mehlber (2024), aumenta a taxa de reutilização de código, desse modo contribuindo para uma maior eficiência no desenvolvimento da aplicação.

Atualmente, o ROS, está em constante evolução e encontra-se em sua segunda versão, o ROS 2, o termo "ROS", neste trabalho fará referência a comunidade, ao projeto de forma geral e ao ROS em sua segunda versão.

2.3.5 Rosbridge Suite

Segundo Koubaa (2015), com o crescimento da internet das coisas (IoT) e a computação em nuvem, um interesse contínuo em conectar robôs com a internet tem sido desenvolvido. Nesse sentido, surgiu a *Robotics Web Tools*, um projeto de código aberto que entrega bibliotecas com meios para integração entre os mundos dos clientes web e sistemas robóticos (TOOLS, 2024).

Nessa perspectiva, a Robotics Web Tools criou, entre outros projetos, a Rosbridge suite, uma meta-pacote que contém funcionalidades focadas em fornecer uma API JSON, ou seja, uma interface de comunicação para acesso dos dados de programas construídos com ROS, criando a possibilidade de uso dessas informações mesmo para desenvolvedores sem conhecimento prévio do ROS (MACE, 2024d).

Sendo assim, o projeto foi responsável por criar o protocolo chamado de rosbridge protocol, que cria especificações para implementações web agnósticas, de maneira que somente é necessário utilizar uma linguagem de programação com suporte a JSON para efetuar a comunicação. Esse protocolo fornece ao cliente web a capacidade de inscrição e publicação de tópicos, chamadas de serviços, obtenção e definição de parâmetros, e compressão de mensagens. Para tal, o projeto fornece um conjunto de bibliotecas que implementam o rosbridge protocol para intermediar a interação entre o sistema web e a o ROS. Essas bibliotecas são a Rosbridge library, a

Rosbridge server e a Rosapi (MACE, 2024d).

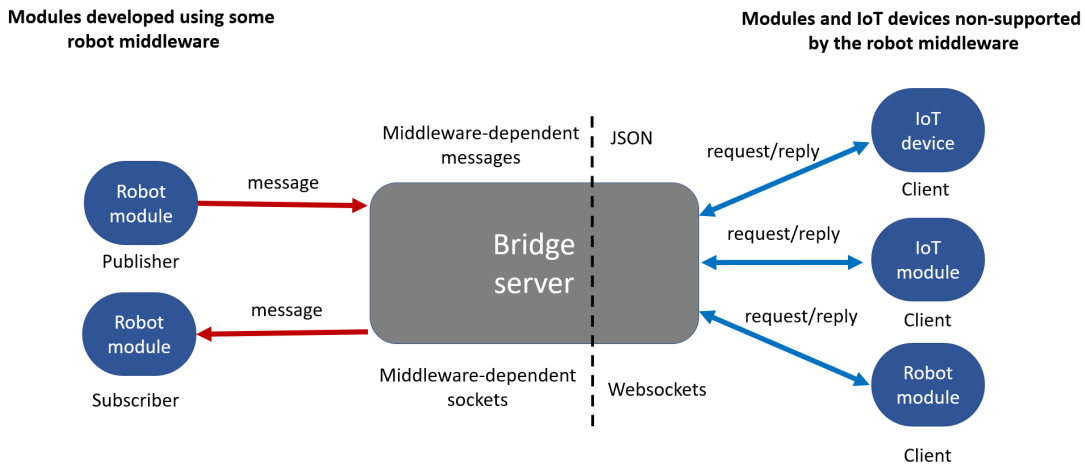
2.3.5.1 Rosbridge Library

A biblioteca Rosbridge é uma ferramenta desenvolvida em Python que facilita a comunicação entre sistemas web e o ROS (Robot Operating System). Essa biblioteca é responsável por interpretar e converter strings em formato JSON para mensagens ROS e vice-versa, permitindo a interação com tópicos, serviços e parâmetros do ROS. Diante disso, a funcionalidade principal da biblioteca é transporte, pois ela foi criada para ser usada na Rosbridge server como uma tradutora para os pacotes compartilhados dentro do servidor WebSocket (MACE, 2024b).

2.3.5.2 Rosbridge Server

A Rosbridge Server, uma implementação do protocolo rosbridge protocol, é responsável por fornecer ao cliente web um servidor WebSocket para que páginas web possam se comunicar com os softwares de robôs construídos com ROS. Desse modo, como mostrado no fluxograma da figura 2, a rosbridge server converte todo pacote intermediado publicado pela abstração de aplicação cliente ou publicados via ROS com o uso de um tópico, respectivamente, para uma chamada chamada ROS, no formato de uma mensagem para os seus tópicos ou para JSON. Com isso os dados transmitidos podem ser recebidos por qualquer um dos dois lados com a linguagem desejada. Assim, com esse fluxo é possível criar uma comunicação eficiente e em tempo real entre a web e sistemas ROS (MACE, 2024c).

Figura 2 – Fluxograma básico de comunicação com a Rosbridge Server



Fonte: (CORONADO; VENTURE, 2020)

2.3.5.3 Rosapi

A Rosapi, a última parte do pacote Rosbridge Suite, se responsabiliza por expor funcionalidades internas do ROS, como obtenção e definição de parâmetros, além da listagem de tópicos. Essas funcionalidades, que normalmente não poderiam ser acessadas por um cliente externamente pois são liberadas somente dentro do ecossistema do ROS, podem ser acessadas pela rosbridge server através do uso da biblioteca do Rosapi (MACE, 2024a).

Com isso, operações comuns, como a obtenção de listas de tópicos ou a configuração de parâmetros, podem ser realizadas por meio de chamadas de serviço para a Rosbridge Server. Através dessa camada adicional de acessibilidade, a capacidade de interação entre o ROS e outros sistemas é simplificada, o que facilita a criação de aplicações que podem controlar e monitorar sistemas robóticos remotamente (MACE, 2024a).

2.4 IROBOT ROOMBA® 614

Para execução deste projeto foi utilizado um robô iRobot Roomba® 614, para futuras referências ele será chamado abaixo somente como roomba.

O robô roomba foi adquirido para a execução do projeto Robótica Aplicada para Engenharias, aprovado por meio do EDITAL 3/2021 - PROEX/REI/IFPI, de 29 de abril de 2021. Após a aquisição a máquina foi acrescida de modificações dentro do ambiente do laboratório de inovação e pesquisa aplicada, chamado de LABIRAS e tornou-se uma plataforma robótica voltada para pesquisa científica (DANTAS, 2022). Nas sessões criadas abaixo, serão apresentadas características do hardware do robô e as respectivas modificações que ele sofreu.

2.4.1 Modificações Realizadas

Para tornar o roomba em uma plataforma robótica, alterações foram feitas, entre elas um ESP32 Dev Module foi adicionado para intermediar a comunicação entre o robô e o sistema externo. Esse microcontrolador foi escolhido por sua compatibilidade com redes Wi-Fi, facilitando a integração com o ROS 2 via rede local.

De acordo com Dantas (2022), para controle das funções do robô foi utilizada a biblioteca *iRobot-Create2-Arduino*, porém devido às diferenças da arquitetura da placa ESP32 com o Arduino. Ainda, houve a reformulação da função de coleta de dados dos sensores para permitir a leitura de mais de um sensor por vez. Apesar dessa melhoria, ainda persiste a limitação física do Roomba, que não permite a leitura simultânea de todos os sensores, dado que a comunicação ocorre via serial com taxa máxima de 115200 baud. Nessa configuração, é possível transmitir até 172 bytes a cada 15 ms, o que define a frequência de amostragem viável.

Para a comunicação com o ROS, foi utilizado as mensagens do pacote create_msgs, estendido para incluir novos tipos mensagens contendo os dados de interesse agrupados em índices específicos mostrados abaixo e listados na figura 1:

- *Cliff* - Índices 2 ao 5
- *Battery* - Índices 7 ao 11
- *Wheels* - Índices 12 ao 15

Também, Dantas (2022) descreve que a coleta dos dados sensoriais no Roomba é realizada por meio da comunicação serial entre o ESP32 e o robô. A integração com o ROS 2 ocorre via rede Wi-Fi local, onde o ESP32, ao ser energizado, inicializa um ponto de acesso que deve ser configurado para acessar a rede Wi-Fi desejada da máquina que executa *Agent* do ROS 2, além do namespace a ser utilizado pelo robô (sendo este último opcional). Após a configuração, o ESP32 estabelece conexão com o *Agent* e passa a enviar e receber mensagens nos tópicos previamente definidos.

Tabela 1 – Lista com sensores que tiveram os dados coletados

Índice	**Nome do pacote**	**Número do pacote**	**Nº de bytes**
0	Bumps and Wheel Drops	7	1
1	Light Bumper	45	1
2	Cliff Left	9	1
3	Cliff Front Left	10	1
4	Cliff Front Right	11	1
5	Cliff Right	12	1
6	Charging State	21	1
7	Voltage	22	2
8	Current	23	2
9	Temperature	24	1
10	Battery Charge	25	2
11	Battery Capacity	26	2
12	Requested Right Velocity	41	2
13	Requested Left Velocity	42	2
14	Left Encoder Counts	43	2
15	Right Encoder Counts	44	2

Fonte: (DANTAS, 2022)

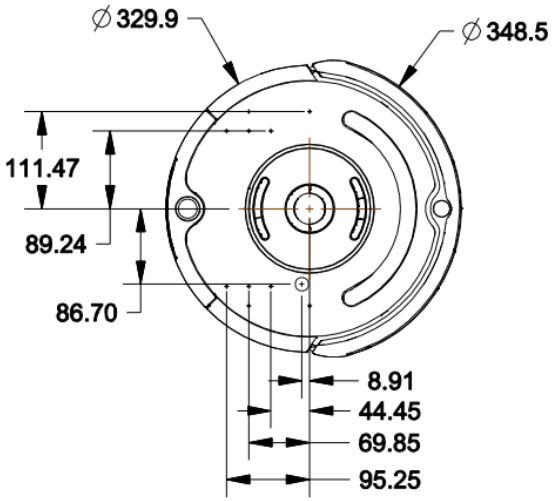
2.4.2 Hardware

O Roomba utilizado neste projeto apresenta estrutura compacta e resistente, com massa aproximada de 3,5 kg e capacidade de suportar até 15 kg de carga útil (DANTAS, 2022). Suas dimensões físicas estão representadas nas figuras 4 e 3, nas quais também se observa a área superior do robô que foi adaptada com uma base fixa para fixação futuras de módulos físicos complementares.

Em relação aos sensores e atuadores embarcados, o robô conta com um conjunto diverso que inclui sensores de beirada, contadores de roda, sensores de colisão, infravermelho, queda, nível de bateria, temperatura, corrente e voltagem elétrica. Já suas saídas são compostas por dois motores de passo independentes, alto-falantes e LEDs de sinalização, o que possibilita controle fino de navegação e comunicação básica com o usuário (DANTAS, 2022).

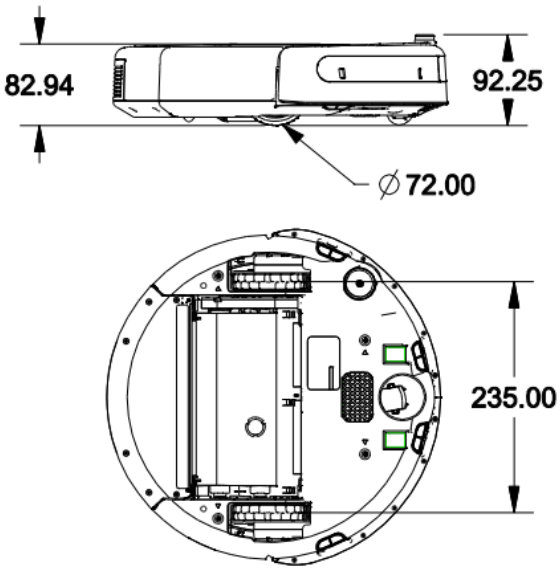
Por fim, o conector mini-din do roomba que foi utilizado para realizar a conexão serial pode ser visto da figura 5.

Figura 3 – Vista superior



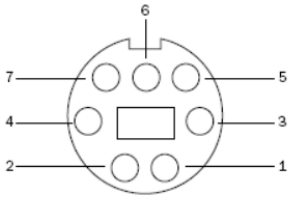
Fonte: (DANTAS, 2022)

Figura 4 – Vista lateral e inferior



Fonte: (DANTAS, 2022)

Figura 5 – Conector mini-din do Roomba



Pin	Name	Description
1	Vpwr	Roomba battery + (unregulated)
2	Vpwr	Roomba battery + (unregulated)
3	RXD	0 – 5V Serial input to Roomba
4	TXD	0 – 5V Serial output from Roomba
5	BRC	Baud Rate Change
6	GND	Roomba battery ground
7	GND	Roomba battery ground

Fonte: (DANTAS, 2022)

2.5 TECNOLOGIAS DE PROTOTIPAÇÃO

De acordo com Zacarioto, Silva e Fernandes (2022), a prototipação se faz importante, quando no início de um projeto, é necessário fundamentar ou visualizar uma ideia, pois através de um protótipo de uma interface, gastos elevados podem ser evitados nessa etapa anterior ao desenvolvimento. Com isso, ao começar a estruturar as funcionalidades e casos de uso no *design* do sistema, novas visões passam a ser vistas, auxiliando o desenvolvedor ou idealizador a ajustar os requisitos do sistema de acordo com as novas demandas. Além disso, com um protótipo, é possível adquirir *feedbacks* de usuários ainda na etapa de modelagem do sistema, garantindo que o projeto atenda melhor às suas necessidades (ZACARIOTO; SILVA; FERNANDES, 2022).

Outrossim, com a prototipação um produto final mesmo que inexistente já está em mãos. Desse modo, é possível responder dúvidas do usuário e realizar testes simples da interação dele com a interface dependendo da ferramenta utilizada para construir essa interface e com essa interação, validar a sua ideia. Esse processo facilita a identificação de problemas e a adaptação do design, além de documentar todo o processo de melhorias e alterações da interface prototipada ao longo do desenvolvimento (OLIVEIRA, 2022).

2.5.1 Figma

O Figma, lançado em 2016, é uma conceituada ferramenta utilizada para prototipação, com versões web e *desktop*. A ferramenta se destaca por garantir alta acessibilidade, pois, com uma versão web agnóstica de sistema, pode ser utilizada em qualquer sistema operacional, mesmo que sua versão *desktop* possua limitações de sistema operacional (OLIVEIRA, 2022; ZACARIOTO; SILVA; FERNANDES, 2022).

Em complemento, esse construtor de interfaces se mostra como uma poderosa ferramenta para a criação de protótipos em pequenas ou grandes empresas. O Figma conta com um ambiente para colaboração em tempo real e compartilhamento de arquivos entre usuários. Além disso, integra a capacidade de versionar o protótipo criado, garantindo um histórico de alterações e diminuindo o risco de perda de funcionalidades no decorrer do projeto (ZACARIOTO; SILVA; FERNANDES, 2022).

Com uma comunidade bastante ativa, o Figma conta com uma variedade de *plugins* (extensões responsáveis por adicionar novas funcionalidades à aplicação). Com essas funcionalidades, novos processos passam a poder ser executados no desenvolvimento do protótipo, como adicionar informações às interfaces ou melhorar a experiência de uso do próprio Figma, o que tende a aumentar a produtividade durante o desenvolvimento do protótipo (OLIVEIRA, 2022).

3 MODELAGEM DO PROJETO

Neste capítulo, são descritos os procedimentos metodológicos utilizados no desenvolvimento do software, bem como a modelagem do sistema por meio de requisitos, diagramas e protótipos de interface. Essa abordagem metodológica orienta o planejamento, a organização e a documentação do projeto, permitindo que o processo de desenvolvimento seja conduzido de forma estruturada e alinhada aos objetivos propostos.

A modelagem de software é uma etapa essencial no desenvolvimento de sistemas, pois ela oferece uma visão abstrata e organizada dos diferentes componentes e processos que fazem parte de um projeto. Através da utilização de diagramas e linguagens de modelagem, como a UML (Unified Modeling Language), é possível visualizar e discutir de forma clara a estrutura e o comportamento do sistema (TEAM, 2025b). Isso facilita a comunicação entre a equipe de desenvolvimento, *stakeholders* e clientes, garantindo que todos compartilhem uma compreensão comum do produto final e das etapas necessárias para sua construção (TEAM, 2025b).

Além de aprimorar a comunicação, a modelagem de software desempenha um papel na antecipação e mitigação de possíveis riscos durante as primeiras fases do desenvolvimento. Ao elaborar modelos detalhados, torna-se mais fácil identificar inconsistências ou problemas de integração antes mesmo de iniciar qualquer tipo de programação, economizando tempo e recursos dos envolvidos. Além disso, aspectos como desempenho, escalabilidade e segurança, podem ser visualizados ainda nessa etapa de modelagem.

Nas próximas seções deste capítulo, serão detalhadas as diferentes etapas e artefatos envolvidos no processo de modelagem de software. A começar pelo Levantamento de Requisitos, nessa etapa são definidos os padrões de aceitação de um produto, estabelecendo todas as características necessárias de um sistema e alinhado-as com as expectativas de usuário e *stakeholders* (todos os interessados no sistema), proporcionando a base para o desenvolvimento da aplicação (CAROSIA; OLIVEIRA, 2023).

Depois, serão apresentado os Diagramas de Casos de Uso, que ilustram as interações entre os atores e o sistema, fornecendo uma visão clara das funcionalidades esperadas. Em seguida, o diagrama de implementação do sistema, ilustrando como as diferentes abstrações do sistema se relacionam fisicamente, exibindo inclusive servidores e dispositivos necessários para execução do sistema.

Ainda, um capítulo contendo a arquitetura modular do sistema que semelhante ao diagrama de implementação, mostra as diferentes partes do sistema, porém com um detalhamento maior em cada módulo com foco em exibir relações internas do

sistema e detalhar o design geral juntamente com a organização dos componentes do software. Por fim, será mostrada a prototipação das interfaces, responsáveis por mostrar visualmente o que objetiva-se como resultado visual final do projeto.

3.1 LEVANTAMENTO DE REQUISITOS

3.1.1 Requisitos Funcionais

[RF1] O sistema deve ser capaz de se comunicar com todos os sensores do robô aspirador iRobot Roomba® 614 utilizando o ROS.

[RF2] O sistema deve coletar dados dos sensores em tempo real, incluindo informações sobre posição, status dos motores e dados da bateria.

[RF3] O sistema deve fornecer uma interface para a visualização dos dados dos sensores do robô em tempo real.

[RF4] O sistema deve fornecer uma interface para listagem dos tópicos ROS.

[RF5] O sistema deve fornecer uma interface visualizar unitariamente as mensagens recebidas de um tópico ROS.

[RF6] O sistema deve fornecer uma interface que exiba os dados recebidos do robô em um painel com os dados estruturados, em gráficos e cores sincronizadas.

[RF7] O sistema deve implementar um mecanismo de comunicação bidirecional (WebSocket) entre o servidor e o cliente.

[RF8] O sistema deve permitir que o usuário conecte-se ou desconecte-se do servidor WebSocket.

[RF9] O sistema deve fornecer uma interface para que o usuário possa visualizar se está ou não conectado com o servidor WebSocket.

[RF10] O sistema deve exibir notificações visuais na interface quando erros de conexão com o servidor acontecerem.

[RF11] O sistema deve conseguir exibir os dados de um banco de dados para exibição do registro de atividades do robô.

[RF12] O sistema deve salvar em um banco de dados um registro de atividade toda vez que uma conexão com o robô for efetuada.

3.1.2 Requisitos Não Funcionais

[RF1] O sistema deve garantir baixa latência na comunicação entre o servidor e a interface gráfica, assegurando que as atualizações dos dados sejam apresentadas em tempo real.

[RF2] A interface gráfica deve ser responsiva e operar suavemente em dispositivos utilizados pelos usuários.

[RF3] O sistema deve ser escalável e projetado para suportar futuros aprimoramentos para integração com diferentes sensores ou dispositivos.

[RF4] A arquitetura do sistema deve permitir fácil atualização e manutenção dos pacotes ROS, da interface gráfica e do servidor web.

[RF5] A interface gráfica deve ser fácil de usar, e proporcionar uma experiência amigável de entendimento.

[RF6] O sistema deve incluir ferramentas para monitoramento e diagnóstico, facilitando a identificação e resolução de problemas.

3.2 DIAGRAMA DE CASO DE USOS

O levantamento dos requisitos da aplicação voltou-se para a necessidade de visualização dos dados de um robô, algo que é imperativo para facilitar ambos o acesso à esses dados e a sua interpretação, no caso desse trabalho os dados a serem analisados eram de um robô aspirador de pó iRobot Roomba® 614. Nesse sentido, segue abaixo os casos uso que atenderam o levantamento de requisitos e que também pode ser vistos na figura 6.

- **Caso de Uso 01 - Conectar com o servidor WebSocket:** O usuário atualmente pode visualizar o estado de conexão com o robô, e conectar-se ou desconectar-se.
- **Caso de Uso 02 - Visualizar os dados do robô**: O usuário pode ver os dados em tempo real do robô, exibidos através de gráficos e ferramentas para fácil entendimento dos dados.
- **Caso de Uso 03 - Registrar *Log* de Conexão**: O sistema deve realizar o registro de um *Log*, ou seja, um registro de atividade toda vez que a aplicação conectar-se ao robô.
- **Caso de Uso 04 - Visualizar todos os tópicos ativos do ROS**: O usuário deve conseguir visualizar, em tempo real, a listagem de todos os tópicos existentes na infraestrutura do ROS.

- **Caso de Uso 05 - Visualizar as mensagens do tópico selecionado**: A partir da listagem dos tópicos ROS, o usuário pode selecionar qualquer um deles e, ao clicar, visualizar em tempo real todas as mensagens que estão sendo publicadas naquele canal específico.
- **Caso de Uso 06 - Acessar controles comando do robô**: Quando a conexão com o robô via WebSocket estiver ativa, o usuário poderá acessar uma interface dedicada à controlar o movimento do robô.

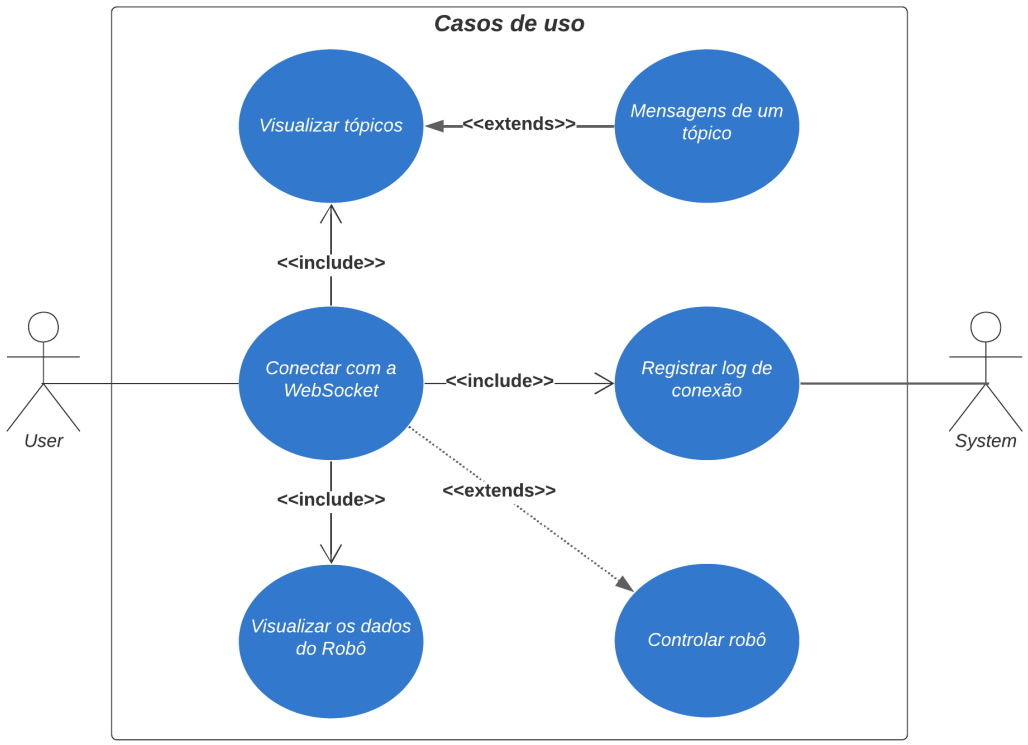
O caso de uso 01 trata da efetuação da conexão da interface gráfica com o servidor WebSocket, em que o usuário pode efetuar as ações de conectar-se ou desconectar-se do servidor. Além dele, o caso de uso 02, relacionado ao caso de uso 01, estabelece que o usuário conectado com o servidor é capaz de acessar uma página para visualização dos dados do robô através de gráficos e um conjunto de outros recursos visuais. O uso 03, também ligado ao primeiro caso de uso 01, define que toda conexão ao robô deve manter armazenado um registro da data e hora.

O caso de uso 04 trata da visualização em tempo real de todos dos tópicos ROS expostos pelo robô. Com esta funcionalidade, o usuário tem acesso a uma listagem completa dos canais de comunicação utilizados pelos nós ROS, o que facilita o entendimento da estrutura do sistema e sua dinâmica de funcionamento.

O caso de uso 05 estende a funcionalidade do caso anterior, permitindo que, a partir da listagem dos tópicos, o usuário clique em qualquer um deles e seja automaticamente inscrito naquele tópico, possibilitando a visualização das mensagens publicadas no canal, que são transmitidas no formato de objetos JSON. Trata-se de um recurso essencial de monitoramento, no qual o usuário pode, de forma dinâmica, acompanhar o conteúdo publicado em tempo real no tópico.

Por fim, o caso de uso 06 explora a interatividade direta com o robô, permitindo ao usuário, quando conectado à WebSocket, acessar uma tela de controle dedicada. Por meio desses comandos, é possível publicar instruções de movimentação, seja para o robô realizar giros em torno do próprio eixo, seja para deslocar-se linearmente no ambiente físico.

Figura 6 – Diagrama de casos de uso interface gráfica



Fonte: Criado pelo autor

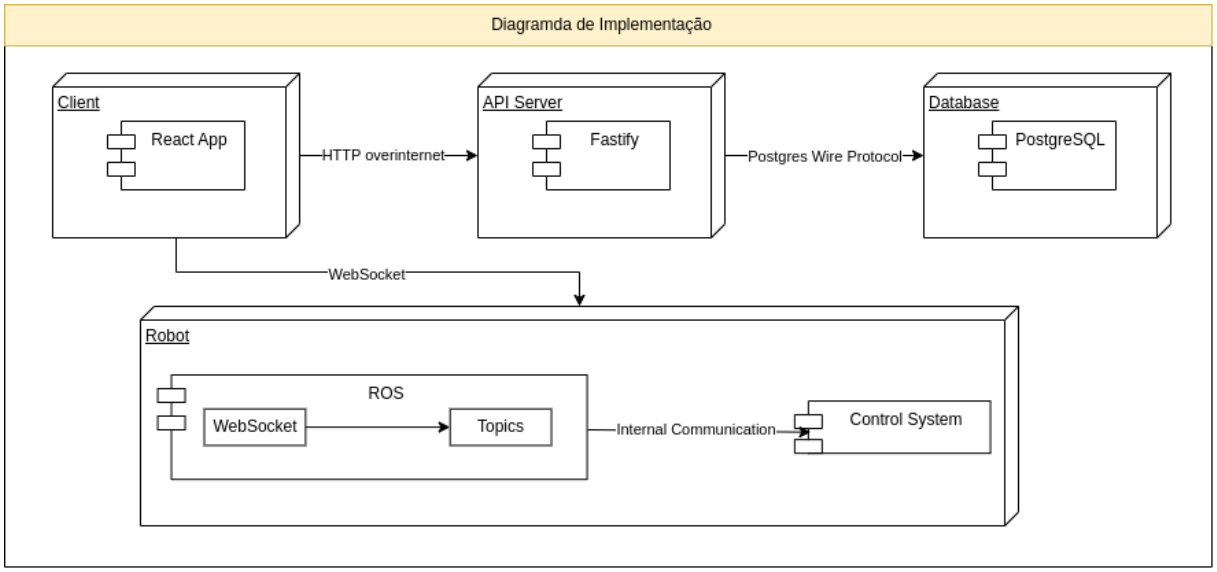
3.3 DIAGRAMA DE IMPLEMENTAÇÃO

Os diagramas de implementação são estruturas que exibem como os diferentes nós do sistema (abstrações do hardware) interagem entre si.Nesse tipo de diagrama, são indicadas as partes de software implantadas em cada nó e as relações de comunicação estabelecidas entre elas, sejam conexões físicas ou lógicas. Além disso, o diagrama é também capaz de exibir as diferentes conexões realizadas por cada nó, as portas em que se comunicam e como essas conexões são efetuadas, podendo ser utilizado para mostrar, inclusive, o tempo execução para estabelecimento de uma comunicação ou de resposta médio para uma requisição (IBM, 2025).

Outrossim, nesse projeto o diagrama é utilizado com foco em transparecer quais estruturas serão necessárias para construir a aplicação. Nesse sentido, o diagrama traz os tipos de conexão realizadas entre os componentes do sistema, sem detalhar exaustivamente os números de porta, uma vez que essas informações se encontram dentro do domínio de portas padrão definido pela IANA.

No diagrama **??**, que pode ser visto abaixo na arquitetura de implementação proposta, foi realizado a divisão em quatro principais elementos: o cliente web, o servidor da aplicação, o banco de dados e o robô.

Figura 7 – Diagrama de implementação



Fonte: Criado pelo autor

O cliente, desenvolvido com a biblioteca React, realiza a exibição dos dados para o usuário do robô, para isso, ele interage com o resto da aplicação de duas maneiras. Na primeira maneira ele realiza requisições HTTP para um servidor, com uma aplicação Fastify utilizada para recuperação e gerenciamento de algum tipo de dado. E na segunda maneira, ele se comunica com o uso do protocolo WebSocket presente no robô para receber em tempo real os seus dados.

No servidor, o serviço web construído com Fastify, por sua vez, atua como o *backend* central do sistema, sendo responsável por persistir dados do cliente referentes a sua atividade de utilização da aplicação. Para que haja essa persistência, por meio do banco de dados, armazenadas informações essenciais ao funcionamento da aplicação, uma comunicação entre o servidor da aplicação e o banco de dados é realizada utilizando o PostgreSQL Wire Protocol, protocolo nativo de interação com os clientes de seus dados.

Finalmente, o robô que opera de maneira autônoma dentro de sua própria arquitetura, possui a especificidade de ter um ecossistema ROS acoplado à sua estrutura. Nesse sentido, o ROS como comunicante com o sistema interno do robô, captura os seus dados e de seus sensores, e os expõe para as estruturas externas a ele, através de um servidor WebSocket criado pelo arranjo de bibliotecas do ROS.

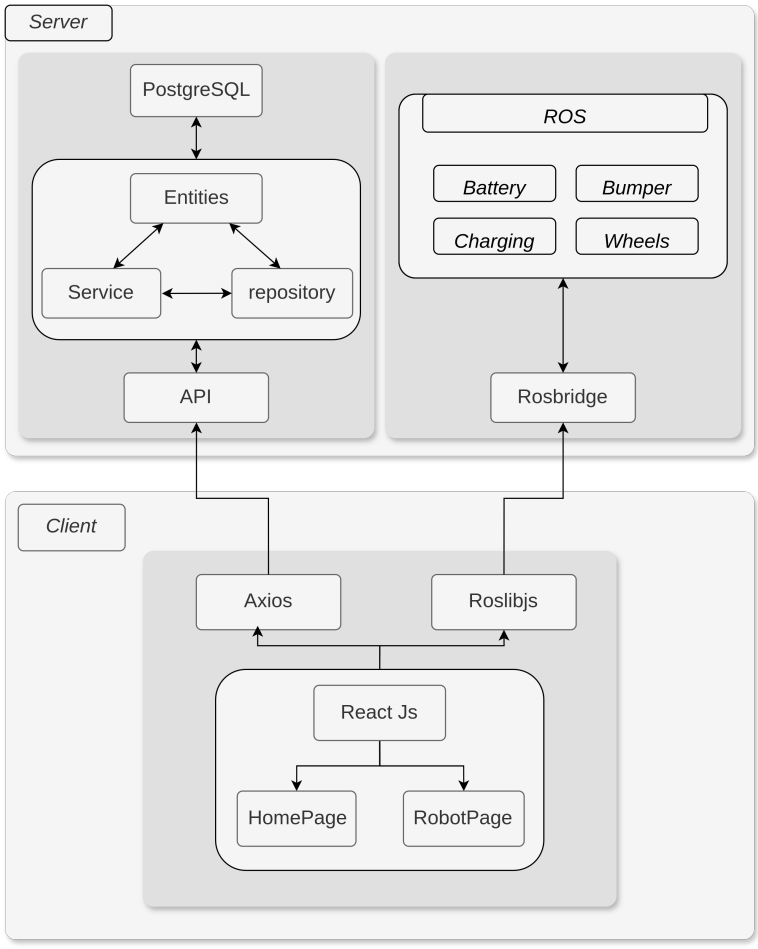
Essa comunicação entre os componentes do sistema ocorre de maneira estruturada para garantir a eficiência operacional de todo o sistema. A arquitetura foi projetada para proporcionar garantindo que os usuários possam interagir com o robô em tempo real sem comprometer a integridade e escalabilidade da aplicação. Além disso, objetiva-se que o sistema possa suportar futuramente novos dispositivos e funcionalidades adicionais, mantendo sua estrutura preparada para crescimento.

3.4 ARQUITETURA DO SISTEMA

Na arquitetura do sistema utilizada, mostrada na figura 8, a aplicação foi dividida em duas camadas servidor e cliente. Esse tipo de divisão é utilizada para representar os recursos da aplicação e como serão organizados. Com isso, o desenvolvedor tem mais facilidade para orientar-se durante o desenvolvimento da aplicação, por fim, acelerando o seu processo de desenvolvimento.

Dessa maneira, na camada do cliente, como mostrado no diagrama, é a camada na qual o usuário pode interagir com os recursos gerados, a partir dos dados retornados pelo servidor da aplicação. Nessa camada, uma interface web React irá utilizar de dois serviços para captura de dados, um desses serviços é uma API, que através da biblioteca Axios, terá seus *endpoints* requisitados com o protocolo HTTP para manipulação dos seus dados pela aplicação cliente, por outro lado um servidor WebSocket será acessado para recuperar dados de um robô em tempo real, e para tal, a Roslibjs será utilizada tanto para implementar o *Handshake* com o servidor, quanto para facilitar a leitura dos dados retornados em fluxo contínuo.

Figura 8 – Arquitetura do sistema



Fonte: Produzido pelo autor

Ademais, no lado do servidor, o sistema robótico é responsável por coletar dados de sensores presentes no robô Roomba, tratando-os como *nodes* dentro do sistema. Esses dados são publicados em diferentes tópicos, cada um representando informações capturadas por sensores específicos. Após isso, a aplicação faz uso da rosbridge, para converter os dados do Roomba em mensagens no formato JSON, facilitando a sua leitura para o cliente quando enviado via WebSocket.

Além disso, como visto na figura 8, os sensores capturados pelo robô Roomba, publicam nos seus respectivos tópicos dados dos sensores de, *bumper*, *charging* e *wheels*, utilizados para mostrar em tempo real para o usuário os valores de cada um desses sensores do robô.

Ainda no lado do servidor, um serviço web construído com Fastify será responsável por disponibilizar os dados armazenados no banco de dados por meio de endpoints. Esses endpoints serão desenvolvidos seguindo os princípios da arquitetura RESTful (Representational State Transfer), também conhecido como API REST, é um modelo amplamente utilizado para a construção de APIs escaláveis. Nesse modelo de arquitetura, um conjunto de boas práticas são definidas para comunicação entre sistemas distribuídos, nele operações do protocolo HTTP, como GET, POST, PUT e DELETE são padronizadas para facilitar a sua leitura e criação de novos recursos.

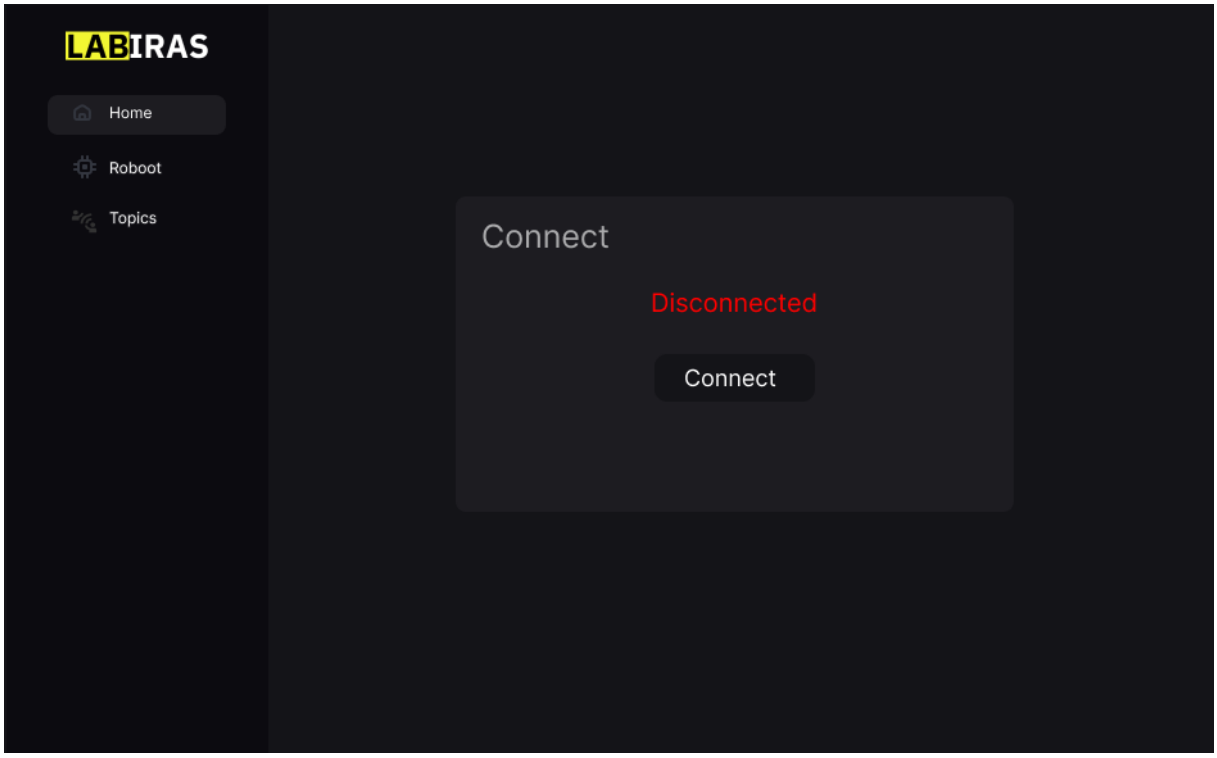
3.5 INTERFACE PROTOTIPADAS

No software serão desenvolvidas cinco interfaces interfaces gráficas, elas foram pensadas visando futuros reajustes e adição de novas funcionalidades, pois observa-se que o projeto tem grande capacidade de crescimento e de receber novas funcionalidades e recursos visuais. As interfaces foram planejadas para serem o mais objetivas possível, destacando somente apenas os principais pontos que o usuário precisa observar sobre o robô no sistema.

Além disso, as cores utilizadas foram pensadas para o maior conforto estético e facilitar o entendimento das funcionalidades do sistema. Nos parágrafos seguintes são exibidas as interfaces prototipadas:

Para começar, a tela inicial na figura 9, primeira interface do sistema, foi projetada para permitir ao usuário estabelecer uma conexão com o servidor WebSocket, essencial para que os dados dos sensores do robô possam ser recuperados e visualizados posteriormente na página *RobotPage*. Com esta interface, o usuário tem acesso ao status de conexão com o servidor WebSocket, podendo também mudá-lo se necessário. Essa funcionalidade garante que o usuário tenha controle sobre o estado da conexão antes de prosseguir para a visualização dos dados capturados pelo robô.

Figura 9 – Tela inicial (HomePage)



Fonte: Produzido pelo autor

Em seguida, denominada de *RobotPage* na figura 10, a interface objetiva exibir os dados coletados pelos sensores do robô em tempo real. Nessa página, os dados são apresentados de maneira visualmente amigável, foram utilizados gráficos, esquemas de cores e ferramentas de visualização que facilitam a interpretação das informações. Nesse sentido, com a atualização dos gráficos continuamente é possível acompanhar o comportamento do robô e a mudança de seus sensores em tempo real. Além disso, a simplicidade dos recursos disponíveis na imagem permitem que um público com menor conhecimento técnico possa facilmente compreender os dados do Roomba em operação.

Figura 10 – Interface dos dados do Roomba (RobotPage)

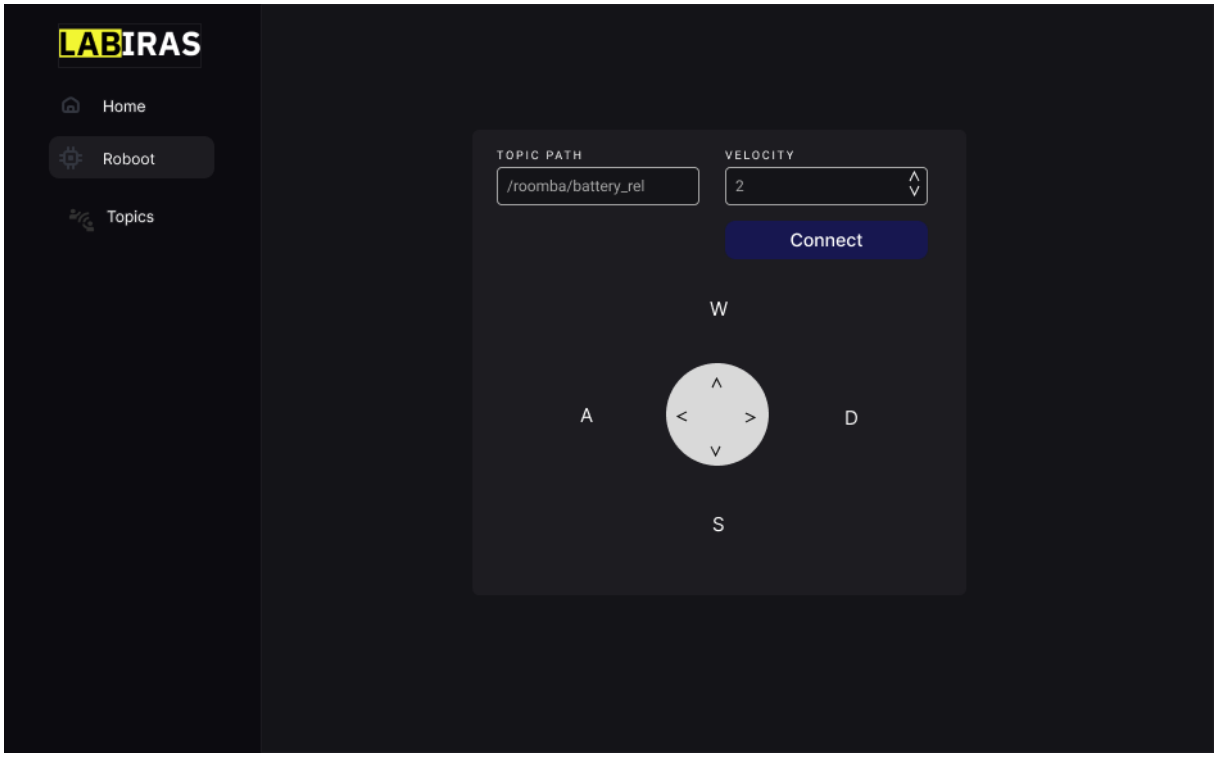
Fonte: Produzido pelo autor

Outro ponto importante é a forma como as informações das páginas são priorizadas e destacadas. Dados críticos ou mudanças bruscas nos valores dos sensores podem ser facilmente notados na interface, alertando o usuário para comportamentos inesperados do Roomba. Desse modo, as interfaces pensadas no sistema se consolidam como poderosas ferramentas de visualização, proporcionando uma experiência rica e intuitiva para os usuários.

A próxima página do sistema é responsável por oferecer ao usuário recursos para o controle físico do robô no ambiente real. Na interface mostrada na figura 17, é possível observar um botão destinado à funcionalidade de controle do robô, esse botão deve direcionar o usuário para a página ilustrada na figura 17, onde poderá efetuar o controle direto do robô. Para isso, o usuário vai precisar realizar a subscrição no tópico responsável pela movimentação do robô, preenchendo os *inputs* mostrados na imagem de nome do tópico e a velocidade desejada.

A interface também disponibiliza os comandos de controle nas teclas “W”, “A”, “S” e “D”, permitindo o movimento intuitivo do robô com o teclado do usuário, em diferentes direções, como avanço, recuo e rotações em torno de seu próprio eixo.

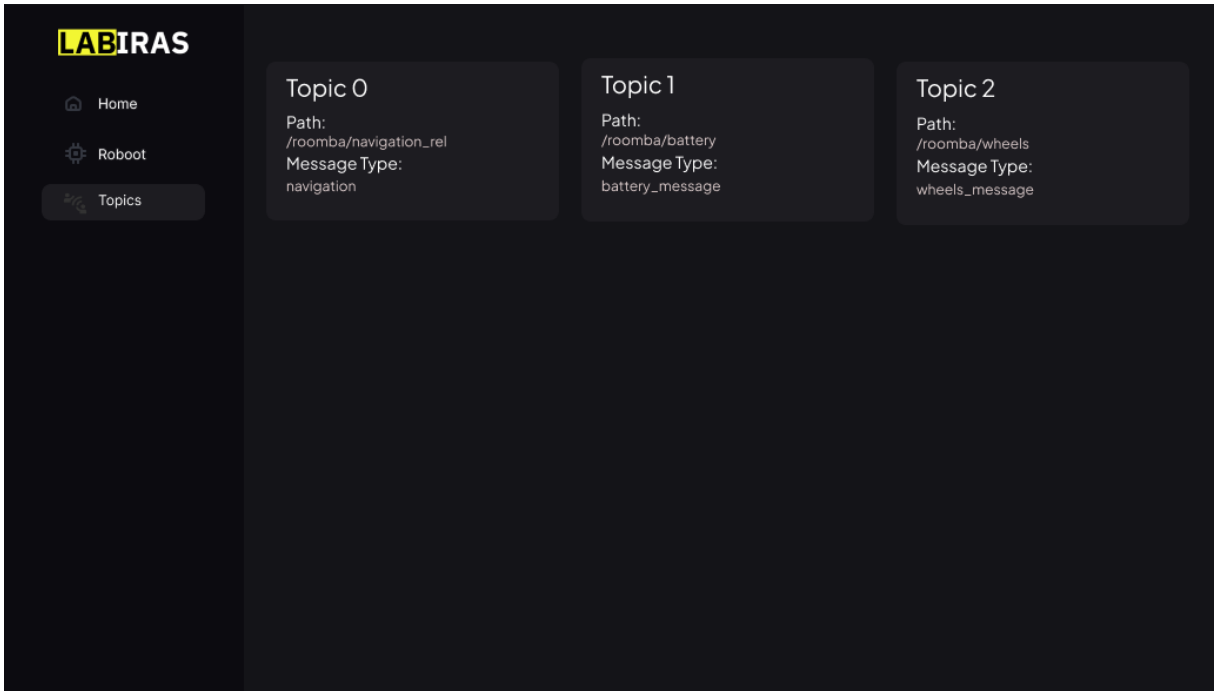
Figura 11 – Página de controle do Roomba (ControlPage)



Fonte: Produzido pelo autor

A sequência de navegação do sistema apresenta também uma página dedicada à listagem dos tópicos disponíveis do ROS. Essa tela, ilustrada na figura 18, exibe de forma organizada todos os tópicos criados na infraestrutura do ROS de comunicação entre nós.

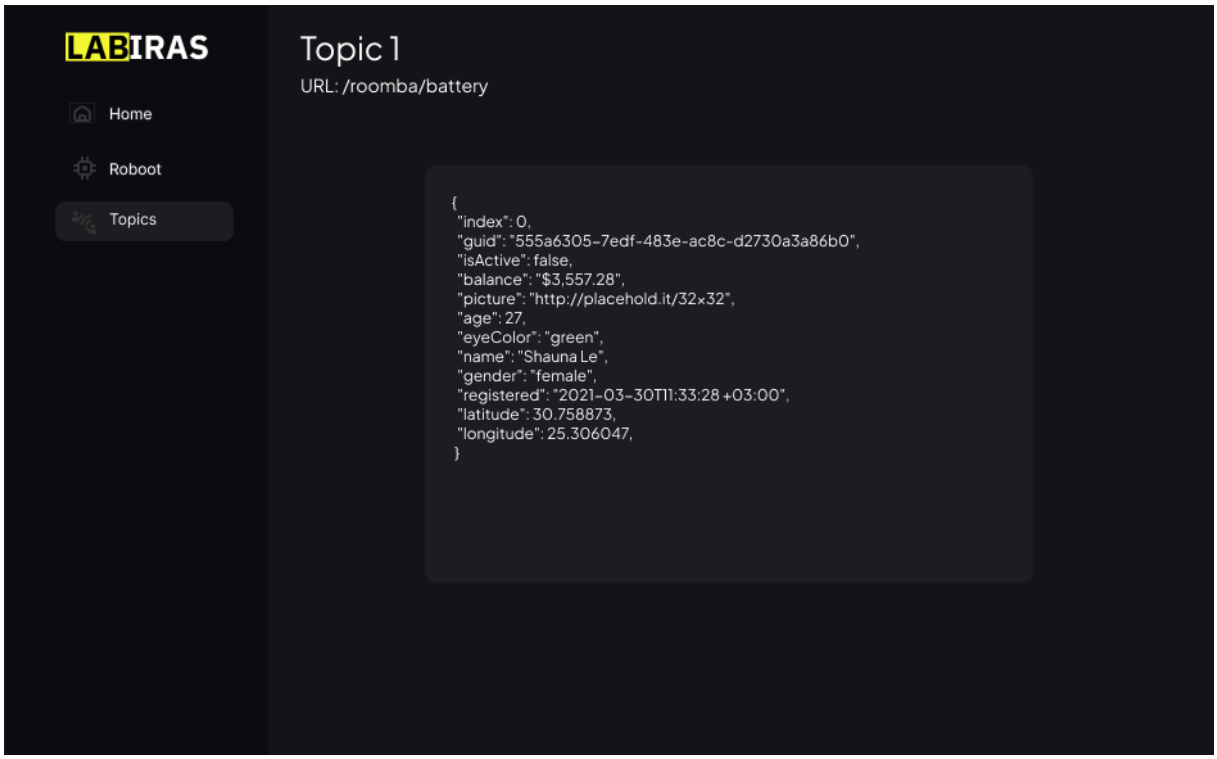
Figura 12 – Página de listagem de tópicos (TopicsPage)



Fonte: Produzido pelo autor

A tela final do sistema apresenta é voltada à visualização detalhada dos dados publicados em um tópico específico. Ao selecionar um dos tópicos exibidos na figura 18, o usuário é direcionado para a página ilustrada na figura 19, onde pode acompanhar, em tempo real, as mensagens publicadas naquele canal.

Figura 13 – Exibição dos dados do tópico (TopicViewPage)



Fonte: Produzido pelo autor

4 SOFTWARE

Neste capítulo, são apresentados os resultados obtidos com o desenvolvimento do trabalho discutido nesse artigo de título "Desenvolvimento de uma interface web para visualização dos dados de um robô aspirador". Os resultados do trabalho dispostos aqui vão refletir a evolução do processo de construção do sistema, destacando as principais funcionalidades implementadas. Além disso, nesse tópico é destacado como o software foi desenvolvido e se atendeu aos requisitos do trabalho.

4.1 APRESENTAÇÃO DO TRABALHO

A interface web para visualização dos dados de um robô aspirador descrita neste trabalho busca desenvolver um sistema que é executado dentro de navegadores de internet para visualização dos dados de um robô, permitindo o monitoramento de seus valores, que devem ser atualizados imediatamente logo após uma mudança em seus valores ser detectada. Para isso, foram abordadas diversas etapas de desenvolvimento, desde a criação das infraestruturas de comunicação, implementação de interfaces gráficas voltadas ao usuário final, até a intercomunicação entre os serviços de entrega de dados e o sistema de visualização desses dados.

No processo de desenvolvimento, a primeira etapa do projeto consistiu na construção de um sistema robótico utilizando o ROS e que atua como uma interface central de comunicação entre os diversos sensores embarcados no robô iRobot Roomba® 614. Posterior a isso, surgiu o desafio "Como expor esses dados para consumo de outra aplicação?". Apesar de encontradas algumas possibilidades, escolheu-se seguir no caminho das bibliotecas da Robot Web Tools, que, sem o uso de intermediários, como algum servidor na internet para armazenamento e posterior consumo, publica-se os dados do robô diretamente pra rede específica com o protocolo de conexão WebSocket.

Em seguida, foi desenvolvido um serviço web responsável pela comunicação entre a interface web e o banco de dados, garantindo que pudesse ser registrado dados da conexão entre a interface de usuário e o robô em toas as conexões. Para isso, foram implementadas rotas de APIs que possibilitam a integração entre o servidor e a interface do usuário, permitindo consultas e manipulação dos dados de forma dinâmica.

Por último, do lado cliente, parte onde o usuário irá interagir, foram criadas as interfaces gráficas, projetadas para possibilitar a visualização dos dados da infraestrutura do sistema robótico, de maneira organizada em gráficos e recursos visuais para entendimento dos seus dados. Essas interfaces permitem que o usuário acompanhe, em tempo real, informações relevantes sobre o funcionamento do robô, como status dos sensores das rodas, nível de bateria e estado de carregamento, ou ainda

observabilidade sobre outras partes internas ao sistema operacional robótico que o usuário deseje visualizar. Com essa abordagem de desenvolvimento, o trabalho busca fornecer um sistema de monitoramento desses dados, mantendo-se aberto a melhorias e expansões futuras, bem como à melhoria de funcionalidades antigas e à criação de novas.

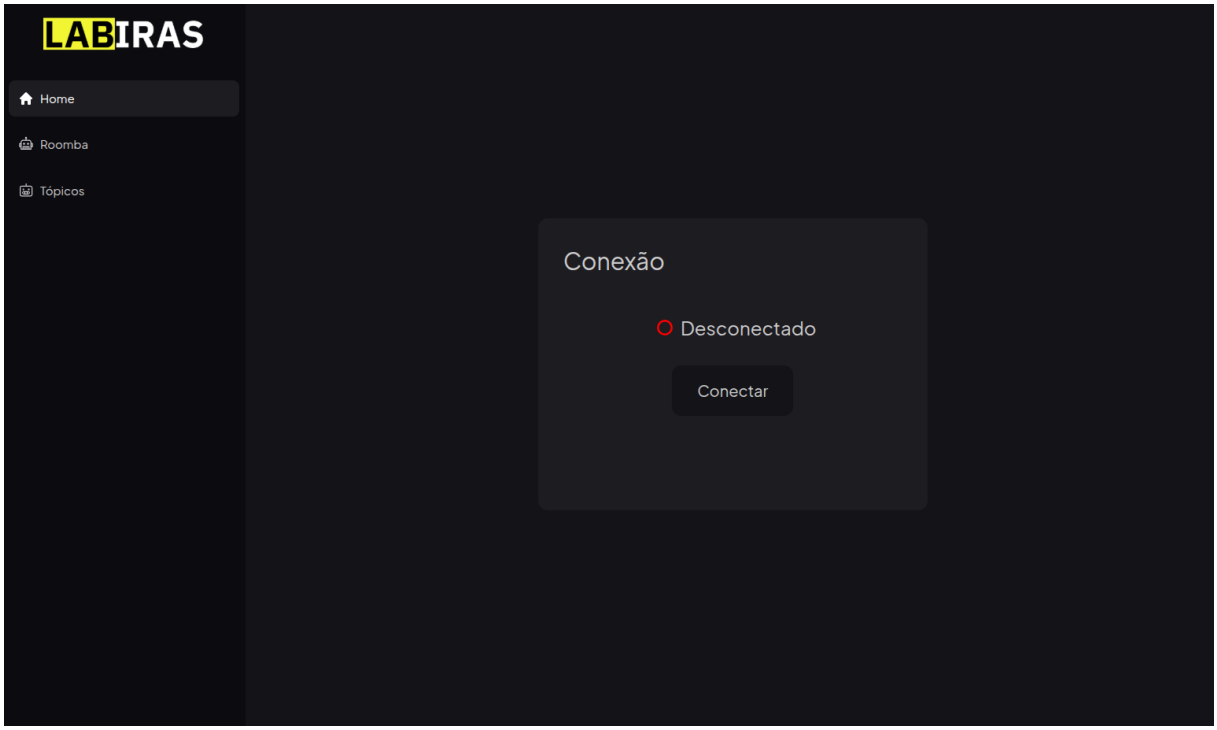
4.2 TELAS FINAIS

Nesta seção, serão apresentadas as interfaces reais desenvolvidas para a aplicação de monitoramento e visualização dos dados do robô aspirador. Nesse sentido, essas interfaces foram projetadas para garantir uma interação intuitiva e eficiente, permitindo que os usuários acompanhem, em tempo real, as informações captadas pelos sensores do robô.

O design final tentou seguir à risca a prototipação das interfaces descritas na etapa de modelagem desse projeto citadas neste trabalho, porém com algumas pequenas modificações, esse design incorpora todos os elementos visuais e funcionais definidos no projeto, incluindo esquema de cores, tipografia, ícones e componentes interativos. Essa disposição de elementos foi pensada para oferecer facilidade de navegação, destacando as informações essenciais dos sensores do robô. Assim, essas interfaces apresentadas refletem a versão final do sistema.

Assim, a interface inicial do sistema, mostrada na figura 14, também chamada de HomePage, é responsável pela conexão com o servidor WebSocket, que é essencial para a recepção e exibição dos dados dos sensores do robô aspirador. Nesta tela, são apresentadas informações sobre o status da conexão, além de um botão de controle que possibilita ao usuário ativar ou desativar a comunicação com o servidor. Ainda, pode ser observado na imagem que a barra de navegação foi implementada com muita semelhança à do design definido.

Figura 14 – Tela inicial (HomePage)



Fonte: Produzido pelo autor

A segunda interface, RobotPage, na figura 15, exibe em tempo real os dados capturados pelos sensores do robô. Na tela, as informações são apresentadas de maneira visual e intuitiva, procurando ser o mais amigável possível, como descrito na interface prototipada na modelagem do projeto, com gráficos dinâmicos e esquemas de cores diferenciados, o que facilita a interpretação dos dados por parte do usuário. Os dados da página são atualizados em tempo real de acordo com as mudanças nos sensores do robô, e objetiva-se que qualquer público possa interpretar esses dados assim que observados.

Dentre os elementos definidos na página, o gráfico de atividade do robô, que exibe por meses a quantidade de dias e horas em que a aplicação foi utilizada, facilita o acompanhamento do uso do robô ao longo do mês, com isso o usuário pode analisar seus padrões de uso ou períodos de inatividade. O gráfico está sendo apresentado com dados correspondentes à um ano de atividade, para alterar o ano selecionado existe um componente de seleção no canto superior direito do gráfico de linhas de atividade do robô.

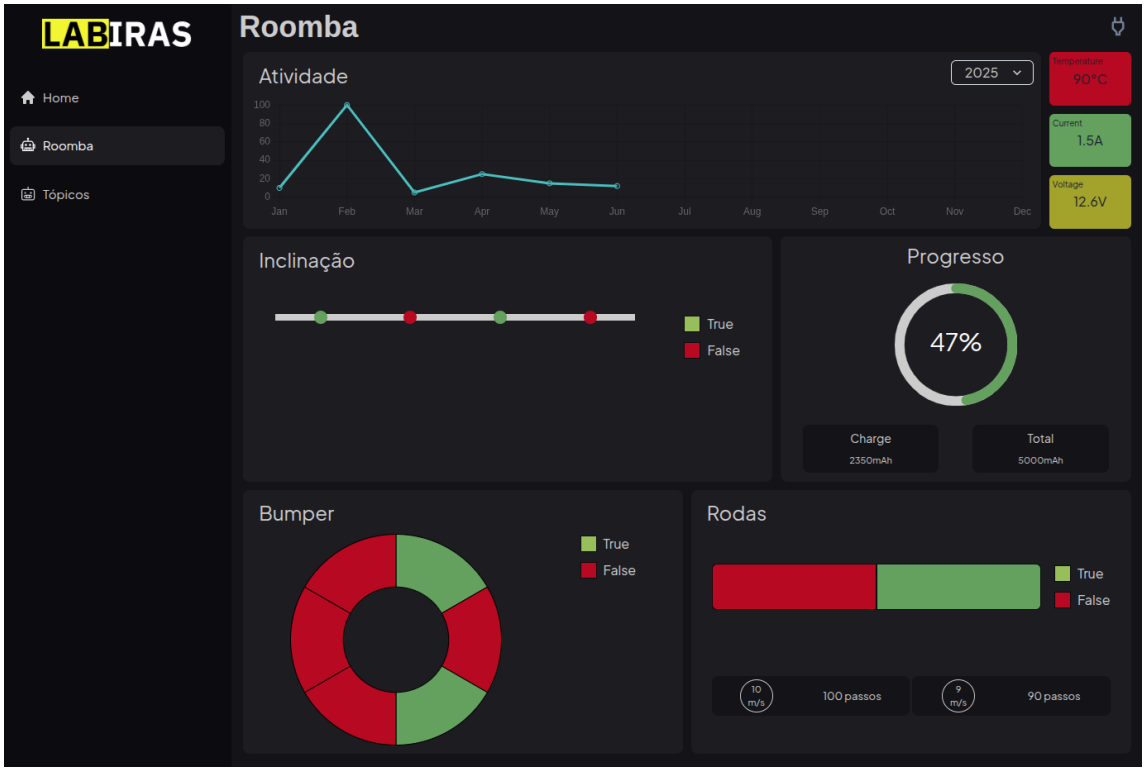
Além disso, acima do gráfico foi implementado um botão que redireciona o usuário ao painel de controle do robô, esse funcionalidade somente é exibia quando a aplicação estiver conectada com o servidor do robô, com o objetivo garantir que a comunicação com o robô esteja devidamente estabelecida antes da execução de qualquer comando. Na figura 16 pode ser visto o botão sendo exibido diante da efetuação da conexão e na falta dela o botão escondido na tela 15.

Ao lado do gráfico de atividade, três *cards* dedicados aos sensores de temperatura, corrente e voltagem são mostrados. O *card* de temperatura, exibe não só o dado do valor do sensor, como também tenta alertar o usuário para mudanças no estado térmico do robô, utilizando um sistema de cores: vermelho quando a temperatura ultrapassa 70℃ e azul quando está abaixo desse valor. Depois, os gráficos de corrente e voltagem mostram a energia que o robô está recebendo no caso de estar conectado ao seu sistema de carregamento (*Dock* de carregamento).

Outro componente importante da interface é o sensor de *cliff*, responsável por detectar a inclinação do robô em relação ao solo. Esse sensor indica inclinações do robô, podendo estar inclinado para frente esquerda, esquerda, direita ou frente direita, o que permite a identificação de obstáculos no deslocamento do dispositivo robótico. O recurso de detalhamento dos valores da bateria mostra a porcentagem de carga do dispositivo, a quantidade de energia disponível e o total suportado pela bateria, todos representados em mAh (miliampère-hora).

Outrossim, a interface também conta com a visualização do sensor de batida (Bumper), que monitora seis pontos de impacto no robô. Esse recurso permite que o robô determine qual a próxima direção que ele pode seguir quando encontrado uma parede. Por fim, há o sensor das rodas, que desempenha duas funções principais, a primeira é identificar se ambas as rodas estão em contato com o chão, e a segunda, exibe métricas detalhadas como quantidade de passos contabilizados pelo motor de passos de cada roda e velocidade de deslocamento do robô.

Figura 15 – Interface dos dados do Roomba (RobotPage)



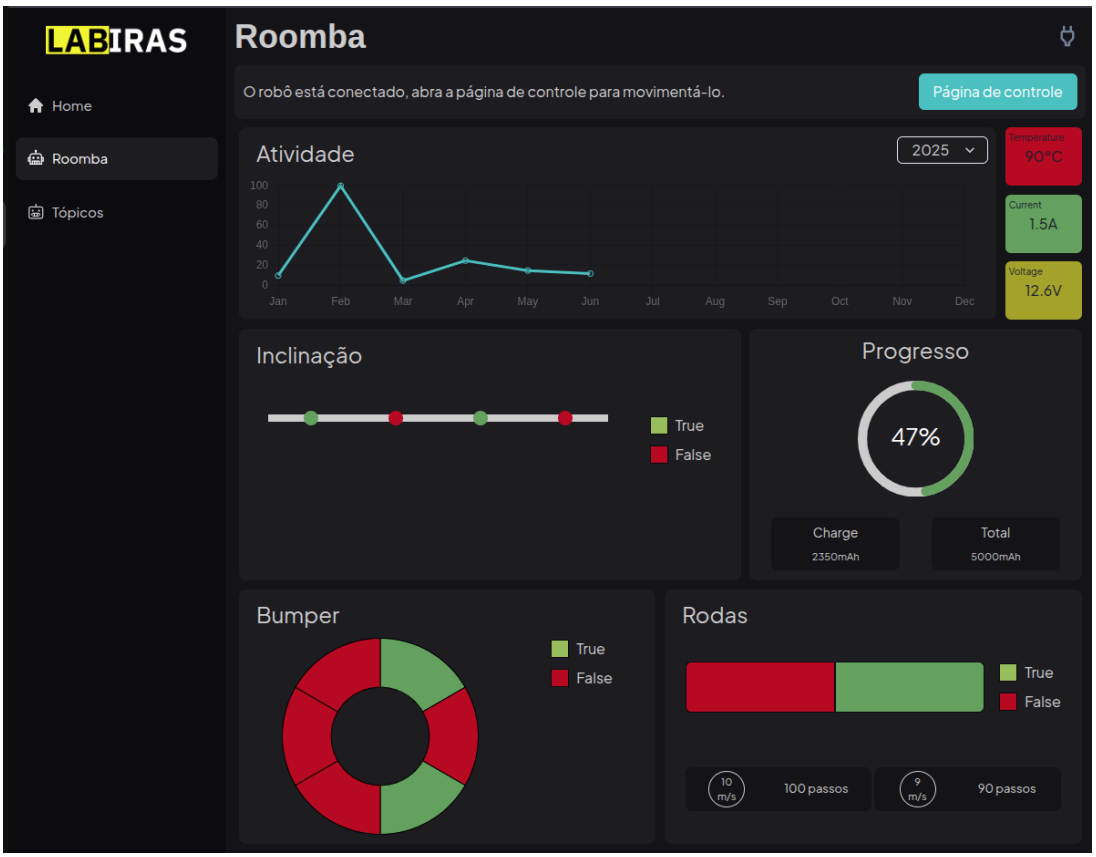
Fonte: Produzido pelo autor

A próxima interface do sistema tem como objetivo fornecer ao usuário recursos para o controle físico do robô em seu ambiente de operação. Na página mostrada na figura 17, pode-se observar a tela que o botão na figura 16 redireciona, nela o usuário poderá manipular o deslocamento do robô por meio dos controles visuais e comandos definidos na interface.

Para realizar o controle efetivo, o usuário deve subscrever-se ao tópico responsável pela movimentação do robô no sistema ROS. No campo de nome do tópico, o sistema já irá trazer preenchido como padrão "/diff_cont/cmd_vel", esse tópico é responsável receber a publicação dos dados de movimentação do roomba, caso o nome tenha sido alterado no robô ou o usuário esteja conectado a uma máquina diferente, será necessário modificar o nome do tópico manualmente. Antes da subscrição definitiva, o usuário também deve definir a velocidade linear de deslocamento, que determina a intensidade dos movimentos executados.

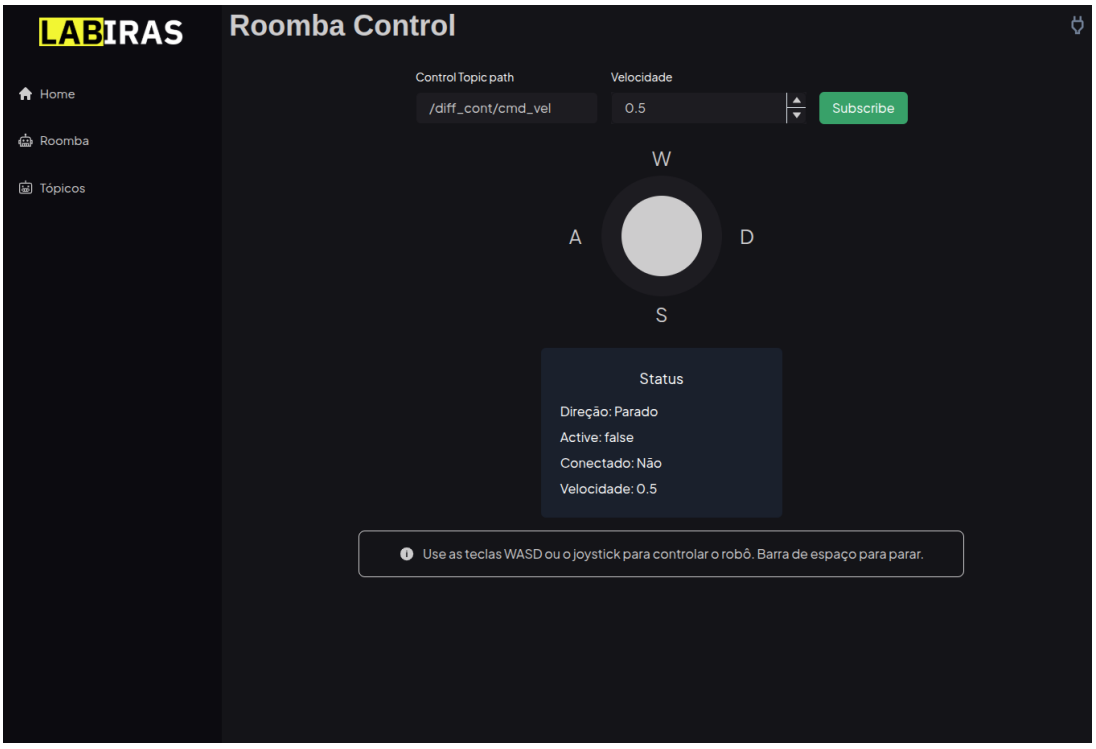
Ademais, a tela disponibiliza controles os quais permitem direcionar o robô, avançando, recuando ou rotacionando sobre o próprio eixo. As teclas mapeados “W” e “S” manipulam respectivamente o movimento de deslocar-se linearmente pela frente ou por trás. As teclas “A” e “D” movimentam o robô em torno de si mesmo, a tecla "A"para a esquerda e a "D"para a direita. Nesse contexto, com objetivo de oferecer visibilidade das ações do usuário na interface de controle, a figura 17 apresenta um painel de status que mostra a última direção designada, o status de conexão com o tópico de movimentação e a velocidade configurada.

Figura 16 – Interface dos dados do Roomba (RobotPage) com botão para módulo de controle



Fonte: Produzido pelo autor

Figura 17 – Página de controle do Roomba (ControlPage)



Fonte: Produzido pelo autor

Em continuidade, a figura 18 apresenta a tela dedicada à listagem de todos os tópicos que compõem a infraestrutura de comunicação do ROS. A listagem é carregada automaticamente sempre que o robô está conectado à aplicação, permitindo ao usuário uma visão ampla da estrutura de comunicação entre os componentes do sistema. Vale ressaltar que nem todos os tópicos exibidos correspondem necessariamente a dados de sensores, os outros são destinados à troca de diferentes variedades de mensagens internas entre os nós ROS, como comandos de controle ou variados estados de hardware.

Figura 18 – Página de listagem de tópicos (TopicsPage)

Fonte: Produzido pelo autor

A última tela do sistema é voltada à exibição detalhada das mensagens publicadas em um tópico, ao selecionar um dos tópicos exibidos na figura 18, o usuário é redirecionado para a página apresentada na figura 19. Nessa visualização, as mensagens transmitidas em tempo real pelo tópico selecionado são mostradas em formato de objetos JSON, convertido internamente no servidor através do conjunto de bibliotecas da Robot Web Tools. Essa funcionalidade é essencial para depuração e análise de dados, com ela o usuário consegue rapidamente assistir um tópico e validar se seus dados, por exemplo se estão de acordo com o que foi configurado para ser publicado no tópico.

Figura 19 – Exibição dos dados do tópico selecionado

Fonte: Produzido pelo autor

Portanto, como era objetivado na etapa de modelagem do projeto, as interfaces finais priorizam a entrega das informações dos sensores do robô, de modo que dados críticos e variações inesperadas nos sensores sejam facilmente identificáveis pelo usuário. Além disso, alertas visuais, em caso de erros da conectividade da aplicação com o servidor, informam sobre o estado de conexão. Desse modo, o propósito dessas interfaces desenvolvidas é garantir uma experiência fluida e informativa para o usuário.

4.3 IMPLANTAÇÃO

A implementação do sistema descrito durante todo esse trabalho para visualização dos dados de um robô aspirador de pó, mais especificamente o iRobot Roomba® 614, necessitou a integração de diversas tecnologias para que esses dados fossem capturados, processados e exibidos pelos sensores do robô.

4.3.1 Front-End

Essa camada do sistema utiliza as bibliotecas React JS e Chakra UI, que proporcionam uma experiência visual moderna e responsiva ao usuário. A construção da interface seguiu uma abordagem baseada em componentes, na qual cada tela foi organizada em blocos funcionais independentes. Com o uso dos componentes semi-customizáveis do Chakra UI, diversas partes da aplicação foram desenvolvidas com

estados próprios e bem definidos, permitindo o reaproveitamento desses elementos em diferentes seções da interface e facilitando a manutenção do código.

Desse modo, com a intenção de interagir com o serviço WebSocket estabelecido, a biblioteca RoslibJS foi utilizada para criação de funções personalizadas para lidar com os estados da aplicação. Essa biblioteca atua como um SDK de comunicação com o ROS, permitindo que a aplicação cliente, escrita em JavaScript, estabeleça uma conexão direta com o servidor WebSocket exposto pelo meta-sistema operacional robótico. Dessa forma, tornou-se possível processar o grande fluxo de dados provenientes do servidor e exibi-los em tempo real para o usuário por meio dos componentes da interface.

No projeto, a RoslibJS, também chamada de Roslib, foi fundamental para a criação de um serviço de gerenciamento da conexão com o servidor e tratamento dos dados recebidos. Para isso, as funcionalidades relacionadas à conexão e ao estado atual da comunicação foram encapsuladas em um *provider* e em um *hook* customizado dentro da aplicação React. Com essa abordagem, qualquer componente da interface pode acessar a instância da Roslib, bem como funções auxiliares para manipular o SDK e interagir com os tópicos ROS de forma simplificada. As principais funções e atributos expostos por esse serviço são apresentados na Tabela **??**, que lista os elementos centrais utilizados na implementação do front-end.

Tabela 2 – Lista das funções e atributos principais da instância *Ros*

Nome da função ou atributo	**Descrição**
`ros: Ros`	Instância da biblioteca *Roslib*, responsável pela comunicação entre a aplicação e o servidor ROS. Todas as funções e atributos listados abaixo são implementações de fácil acesso e utilização, criadas a partir desta biblioteca.
`isConnected: boolean`	Indica o estado atual de conexão com o servidor WebSocket dentro da aplicação. Quando verdadeiro, significa que há comunicação ativa entre o cliente e o robô.
`isConnecting: boolean`	Representa o estado de tentativa de conexão. É utilizado para exibir estados intermediários de conexão e evitar múltiplas requisições simultâneas.
`error: string`	Armazena mensagens de erro geradas durante o processo de conexão, permitindo que sejam posteriormente exibidas como notificações informativas ao usuário.
`url: string`	Define o endereço de conexão com o servidor WebSocket. Para os fins deste trabalho, a aplicação utiliza o padrão `localhost:8080`.
`toggleConnection()`	Função responsável por conectar ou desconectar a aplicação do servidor WebSocket. É utilizada principalmente na implementação da *HomePage*, conforme mostrado na figura 14.
`createListener(topic: Topic)`	Função utilizada para realizar a subscrição nos tópicos. Por exemplo, ela foi utilizada para acessar os dados de um tópico exibido na figura19.
`topics: Topic[]`	Atributo que mantém o estado atualizado de todos os tópicos ROS detectados e registrados pela conexão com o servidor.
`listeners: Listeners[]`	Esse atributo, armazena qualquer variável criada a partir de uma subscrição em um tópico, essa variável é atualizada continuamente enquanto a conexão persistir.

Fonte: Produzido pelo autor

Ainda, após a construção dos serviços customizados de acesso a Roslib, para tratamento dos dados recebidos dos tópicos foram desenvolvido classes de manipulação desses dados. A partir do acesso do usuário na tela 15, o sistema realiza dinamicamente a subscrição nos tópicos dos sensores, todos os tópicos são percorridos e para cada tópico desejado, um listener é criado através da função createListener,

esse listener é armazenado globalmente na variável listeners, descrita na tabela 2. Após isso, para cada mensagem ouvida, o sistema utiliza as classes criadas para gerar instâncias das respectivas mensagens de cada tópico.

Com essa abordagem, esses objetos são devidamente formatados, além de seus dados se tornarem tipados pelo construtor da classe, qualquer dado indesejado vindo da mensagem não poderá ser utilizado pois o construtor não usará o dado, isso facilita o uso dos objetos na aplicação e diminui a possibilidade de erros durante a sua implementação nos devidos componentes. Assim, esses objetos poderiam de imediato ser utilizados para modificação dos estados da aplicação, ou seja, os dados exibidos para o usuário final, a exemplo as informações dos sensores mostradas na RobotPage.

Ademais, o gráfico de atividade do robô, criado para registrar todas as conexões do usuário e o seu tempo de uso da aplicação web com o robô dividido em meses, necessitou da utilização de um bando de dados. Nesse sentido, a utilização de um banco de dado relacional em detrimento à alternativas não relacionais é visando a escalabilidade da aplicação, em que futuramente dados como de usuários, robôs e tópicos por robôs precisaram ser registrados, e sendo esses dados altamente relacionados entre si, optou-se nesse projeto pelo armazenamento de dados em um banco relacional PostgreSQL.

Para viabilizar o acesso e o envio desses dados à aplicação, foi desenvolvida uma API RESTful utilizando o framework Fastify, responsável pela comunicação com o banco de dados. Além disso, para consumo da API no lado do cliente (frontend), foi utilizado a biblioteca Axios para realização das requisições HTTP aos endpoints, permitindo a criação, recuperação e análise de logs de conexão e desconexão.

Nesse contexto, diante da conexão via *WebSocket* com o servidor, a aplicação executa uma requisição *POST* para o *endpoint /logs*, enviando no corpo da requisição as seguintes informações:

- *timestamp*: data e hora atual do evento;
- *event*: tipo de evento, podendo ser *"connection"* ou *"desconnection"*.

Como resposta, o servidor retorna um session_id, que é armazenado na aplicação enquanto a sessão estiver ativa, através desse identificador os eventos de conexão e desconexão são relacionado.

A aplicação trata diferentes casos de desconexão, incluindo:

- Desconexão manual, realizada pelo usuário na interface da aplicação;
- Fechamento inesperado da aba do navegador;
- Quedas de conexão por instabilidade de rede.

Em todos os casos, a lógica implementada garante que, ao ocorrer uma desconexão, o navegador execute uma função final que registra um novo log no servidor, enviando o *timestamp*, o tipo de evento *"desconnection"* e o session_id previamente armazenado.

Para alimentar o gráfico de atividades, a aplicação realiza uma requisição *GET* ao *endpoint* /logs/stats, passando como parâmetro o ano desejado. O servidor retorna os dados estruturados por mês, contendo a quantidade total de acessos e o tempo total de utilização do robô, permitindo a exibição das informações de forma visual e organizada no gráfico da interface web.

4.3.2 Back-End

No serviço web da aplicação, uma API para acesso de um banco de dados PostgreSQL foi criada, nela todo o processamento dos dados enviados via *endpoints* é realizado, deixando o lado do cliente livre desse tipo de lógica de negócio. Além disso, o serviço web foi estruturado em *entities*, *services* e *repository*, de modo que cada um se responsabiliza por uma parte do processo de retorno de dados de uma requisição. As *entities*, ou entidades, são estruturas que represam os objetos fundamentais do domínio da aplicação. Os *services*, ou casos de uso, são os que conhecem a regras da aplicação, eles encapsulam a lógica de negócio, garantindo a execução das normas definidas no sistema. Em conclusão, a interação com o banco de dados é gerenciada pelos *repositories*, realizados as operações de acesso e persistência das informações.

Os *endpoints* implementados na API têm como principal finalidade permitir o registro, consulta e análise dos dados de uso da aplicação. Cada um deles foi desenvolvido conforme apresentado na tabela 3.

Tabela 3 – Endpoints da API, métodos HTTP e suas funcionalidades.

Método	**Endpoint**	**Descrição**
POST	/logs	Cria um novo registro de conexão ou desconexão.
GET	/logs/:id	Retorna um log específico pelo seu identificador.
GET	/logs/ session/:session_id	Retorna todos os logs de uma mesma sessão.
GET	/logs	Retorna todos os logs cadastrados na aplicação.
GET	/logs/stats	Retorna estatísticas de uso por mês e ano.

O *endpoint* /logs é responsável pela criação de novos registros. O corpo da requisição deve conter um *timestamp*, o tipo de evento (*connection* ou *desconnection*) e, no caso de eventos de desconexão, um session_id. O serviço realiza validações que

incluem garantir que o evento informado seja válido e verificar se a sessão correspondente já possui dois registros (um de conexão e um de desconexão), essas medidas são fundamentais para evitar inconsistências na contagem de tempo de uso.

Os *endpoints* /logs/:id, /logs/session/:session_id e /logs permitem a consulta dos registros de diferentes formas. O primeiro busca por identificador único, o segundo retorna todos os eventos associados a um mesmo session_id, o que é útil para validações internas e o último retorna todos os registros salvos na aplicação, podendo ser usado para fins de auditoria e monitoramento.

O *endpoint* /logs/stats agrega os registros por mês, com base em um parâmetro de ano informado na requisição. A resposta traz a quantidade de sessões registradas e o total de horas de utilização do robô em cada mês.

Outrossim, para realizar a comunicação eficiente com os sensores dos dispositivo iRobot Roomba® 614 foi essencial a utilização da biblioteca de código aberto ROS. Através de sua utilização, foi possível abstrair toda a estrutura e *drivers* da arquitetura do robô. Nesse sentido, dentro do ecossistema ROS, os sensores do robô são tratados como nós, e estes realizam a exposição de seus dados através de estrutura denominadas tópicos, que podem ser publicados ou sobrescritos. Assim, na aplicação robótica montada, os sensores responsáveis pelas informações de carregamento, bateria, batida e rodas, publicam seus dados nos respectivos tópicos, battery, bumper, charging_state, cliff e wheels.

Após a configuração estrutural do ROS com a publicação em tópicos, a execução da biblioteca Rosbridge, que é parte de um conjunto de bibliotecas chamado de Rosbridge Suite, é o componente principal que levou à possibilidade de criação desse projeto. Através da execução das bibliotecas dentro desse meta-pacote ROS, é possível criar uma interface de comunicação para acesso dos dados internos de um sistema ROS. Desse modo, as bibliotecas, Rosbridge Library, Rosbridge Server e Rosapi, executadas dentro do ecossistema ROS, conseguem expor os tópicos publicados com os dados do robô e ainda convertê-los na estrutura JSON, principal formato utilizado para compartilhamento de dados na web. Para tal exposição, o serviço de comunicação aberto pelas bibliotecas é o do protocolo WebSocket, permitindo que uma vez efetuado o *handshake* com o servidor, o cliente tenha acesso total a todo o fluxo de dados internos ao robô aspirador.

Cabe ressaltar que, conforme apresentado no referencial teórico, a configuração do Agent do ROS 2 no ESP32 permitiu que o robô iRobot Roomba® se integrasse ao mesmo ambiente de execução do Rosbridge. Essa integração garantiu que os tópicos publicados pelo robô estivessem acessíveis dentro do ecossistema ROS, possibilitando a comunicação contínua entre o hardware e os serviços web criados aqui nesse projeto.

Desse modo, a implementação de um sistema com os requisitos definidos

no projeto foi efetivada, e o sistema alcançou uma solução para monitoramento e análise dos dados do robô aspirador em tempo real. Para esse fim, foi realizada a integração de diversas bibliotecas em diferentes módulos, estes detalhados aqui na implementação, algumas das palavras chave que se destacam nesses módulos são ROS, React, Rosbridge, Roslib Js, Fastify e PostgreSQL.

4.4 VALIDAÇÃO E TESTES

4.4.1 Ambiente de Testes

Os testes foram conduzidos em laboratório, com o iRobot Roomba® 614 em operação, o ESP32 conectado à mesma rede local do computador executando o *micro-ROS Agent*, e a aplicação web acessada via navegador. A máquina utilizada para realização dos testes possuía Ubuntu 24.04 LTS como sistema operacional, ROS 2 Rolling compilado a partir do código-fonte, *rosbridge_server* e *rosapi* também compilados localmente e executados na mesma instância do ROS 2. O *rosbridge-server* disponibilizou o serviço WebSocket, e a *rosapi* forneceu introspecção do sistema ROS, incluindo a listagem e a consulta de tópicos.

4.5 PROCEDIMENTO DE TESTES

Com o ambiente configurado, o robô foi energizado, o ESP32 foi conectado à rede local e apontado para endereço da máquina que executava o Agent. Em seguida, iniciou-se o rosbridge_server e o rosapi na instância do ROS 2, confirmando a disponibilidade do serviço WebSocket. A aplicação web foi aberta no navegador e estabeleceu a sessão com o rosbridge.

A partir desse ponto, realizou-se a descoberta dos tópicos disponíveis e a subscrição aos tópicos de sensores relevantes através das páginas de listagem de tópicos e de acesso individual das mensagens recebidas de um tópico. Posterior a isso, para validação do caminho inverso de comunicação, foram publicados comandos de movimentação no tópico /diff_cont/cmd_vel, com observação do deslocamento do robô. Por fim, validou-se a camada de persistência por meio do registro de sessões no endpoint /logs e da consulta ao endpoint /logs/stats com o parâmetro de ano, verificando o retorno das respostas.

Antes da listagem a seguir, destaca-se que cada caso de teste foi executado pelo menos uma vez em sessão controlada, com repetição quando necessário para confirmação do comportamento observado. A lista resume os casos contemplados e o objetivo de cada um.

- **Conectividade e sessão**: encerrar e estabelecer conexões por WebSocket.

- **Descoberta de tópicos**: listar tópicos do ROS, atualizar a listagem quando nós são iniciados ou finalizados e garantir consistência entre o que a interface apresenta e o que o ROS expõe.
- **Subscrição e exibição**: subscrever tópicos de sensores, como battery, bumper, charging_state, cliff e wheels, e apresentar mensagens em tempo real na interface.
- **Publicação de comandos**: publicar no tópico /diff_cont/cmd_vel para acionar movimentação básica, observando resposta física do robô em deslocamentos curtos.
- **Persistência e estatísticas**: registrar pares de conexão e desconexão por sessão, por meio do consumo dos *endpoints* da api

4.6 RESULTADOS

Para validação das funcionalidades, utilizou-se da integridade como critério principal, em cada subscrição, foram conferidos a presença e o significado dos campos recebidos, a atualização contínua das leituras na interface e a correspondência entre o que a aplicação exibiu e o que foi observado por comandos do ROS, como ros2 topic list para descoberta e ros2 topic echo para leitura direta dos valores do tópico.

No caminho de envio de dados, a publicação de mensagens em /diff_cont/cmd_vel resultou em movimentos controlados do robô, confirmando o efeito de comando de ponta a ponta, do navegador ao robô, mostrado na figura 20. Na camada de persistência, a criação de registros de sessão e a consulta agregada por mês no /logs/stats retornaram dados compatíveis com as operações realizadas, sem duplicidades de sessão que pudessem comprometer a leitura de horas de uso.

Figura 20 – Imagem do Roomba sendo controlado

Fonte: Produzido pelo autor

Entretanto, apesar dos testes realizados, vale ressaltar que não foram coletados valores quantitativos de desempenho, como latência da conexão entre a publicação no ROS e a renderização no navegador, tempo de resposta dos serviços HTTP, vazão de mensagens ou uso de recursos. Porém, essa ausência não impede a conclusão funcional da aplicação, pois o objetivo dos testes foi a validação das funcionalidades propostas na solução desse projeto, contemplando os casos de testes descritos anteriormente. Ainda assim, essa limitação é reconhecida e foram feitas na conclusão recomendações para trabalhos futuros.

5 CONCLUSÃO E TRABALHOS FUTUROS

Desse modo, a implementação do sistema "Desenvolvimento de uma Interface Web para Visualização dos Dados de um Robô Aspirador"alcançou os objetivos estabelecidos, resultando em uma plataforma eficiente para o monitoramento dos sensores do iRobot Roomba® 614. Nesse sentido, a interface web desenvolvida proporciona uma experiência de usuário intuitiva e acessível, além de apresentar informações de maneira clara e organizada. Com o uso do React foi gerado um sistema modularizado e componentizado, atendendo as necessidades de visualização do robô, sem abrir mão de realizar futuras modificações.

A comunicação entre o Robotic Operating System (ROS) e a aplicação web foi implementada com sucesso, utilizando a Rosbridge para converter os dados dos sensores do robô para o formato JSON e transmiti-los via WebSocket. Também foi realizada a implementação de um banco de dados relacional PostgreSQL, com uma API desenvolvida em Fastify, possibilitando o armazenamento estruturado dos registros de uso da aplicação e a geração de estatísticas de utilização por mês.

Atualmente, o sistema permite que múltiplos usuários acessem simultaneamente os dados do robô, cada instância aberta da aplicação cliente estabelece sua própria conexão com o servidor WebSocket disponibilizado pelo Rosbridge, o que viabiliza sessões paralelas de visualização sem conflito. Essa abordagem, no entanto, ainda está limitada ao contexto de redes locais, dado que o ROS 2, por padrão, não opera diretamente sobre redes externas sem a configuração de túneis ou proxies de comunicação.

Ademais, a aplicação foi desenvolvida com foco específico no Roomba, com componentes da interface voltados à leitura e exibição dos dados desse robô em particular. Entretanto, a infraestrutura adotada, baseada na comunicação via Rosbridge e tópicos ROS, é suficientemente genérica para suportar outros robôs ou implementações realizadas com ROS, desde que estes também estejam executando o pacote da Rosbridge. Com isso, partes da aplicação como a tela de conexão WebSocket e a listagem dinâmica de tópicos podem ser reutilizadas em outro projetos robóticos robóticos.

Além disso, a aplicação se mostra especialmente útil no contexto laboratorial, a exemplo do ambiente LABIRAS, onde alunos e pesquisadores frequentemente desenvolvem novas soluções ou modificações sobre o funcionamento do robô. A interface desenvolvida permite uma rápida verificação do estado atual dos tópicos publicados no sistema ROS, bem como a visualização dos dados em tempo real. Em alternativa as ferramentas padrões oferecidas pelo ROS, que geralmente exigem o uso de comandos em terminal para tarefas como escutar ou listar tópicos, esta aplicação oferece uma

alternativa visual e imediata. Isso facilita a depuração e os testes durante o desenvolvimento de novos nós ou integrações com o robô, reduzindo uma parte da complexidade dessas interações.

Em síntese, a implementação realizada comprova a viabilidade de integração entre sistemas robóticos baseados em ROS e aplicações web modernas, utilizando tecnologias amplamente adotadas no mercado. O sistema desenvolvido serve como base sólida para futuras extensões, como, suporte a múltiplos dispositivos, persistência avançada de dados e acesso remoto fora do ambiente local.

Outrossim, a validação do sistema foi realizada por meio de testes com o iRobot Roomba® 614, confirmando que a comunicação entre o ROS e a interface React ocorre de forma eficiente. A exibição dos dados do robô na interface web foi precisa e responsiva, garantindo que o usuário pudesse monitorar as informações dos sensores do robô, como status da bateria, temperatura, sensores de colisão e inclinação, além da atividade do robô ao longo do tempo.

Apesar da consolidação bem efetuada da aplicação, é importante ressaltar que melhorias podem ser exploradas no futuro para tornar a aplicação mais robusta e com um escopo maior de aplicabilidade. Um dos caminhos mais promissores é a adição do suporte ao gerenciamento de múltiplos robôs simultaneamente.

Nesse cenário, um sistema de permissões poderia ser utilizado para limitar as atividades de um usuário, por exemplo para definir quem poderia visualizar os dados ou controlar remotamente cada robô. Através dessa abordagem, não apenas a aplicação seria ainda mais flexível e segura, como tornaria possível situações como dois usuários poderia interagir com robôs específicos dentro de um mesmo ambiente com a mesma aplicação web.

Além disso, como mencionado na sessão de testes da aplicação, para consolidação da aplicação, métricas com foco em mensurar a latência ponta a ponta entre a publicação no ROS e a renderização no navegador ou do tempo de resposta dos serviços HTTP envolvidos nos *endpoints* de uso devem ser adicionadas. Assim, medições simples pode ser incluídas, como registrar registrar o horário em que a mensagem é enviada no ROS e o horário em que ela aparece na página para calcular a latência, medir o tempo de resposta das requisições HTTP aos *endpoints* e definir metas de latência média na rede local.

Portanto, o sistema desenvolvido estabelece uma base sólida para futuras evoluções, permitindo que novas funcionalidades sejam incorporadas de forma progressiva. Com uma arquitetura modular e a integração eficiente das tecnologias empregadas, futuramente a aplicação pode crescer e se adaptar a novas demandas no campo da robótica e automação, consolidando-se como uma solução versátil e escalável para o monitoramento não só de um robô aspirador, mas de uma diversidade deles.

REFERÊNCIAS

ALURA. *Node.js para frameworks front-end*. 2015. Disponível em: https://alura.com.br/artigos/nodejs-para-frameworks-front-end?srsltid=AfmBOopa-2vt5Z431xSATrs4h3hwWUZ7pi67B9QxAtRkmQOACjFhdBYF. Acesso em: 15 set. 2024.

CAROSIA, J.; OLIVEIRA, J. da S. Uso da modelagem para evantamento de requisitos na proposta de tecnologia de uma empresa. In: *Congresso de Tecnologia-Fatec Mococa*. [S.l.: s.n.], 2023. v. 7, n. 1.

CORONADO, E.; VENTURE, G. Towards iot-aided human–robot interaction using nep and ros: A platform-independent, accessible and distributed approach. *Sensors*, MDPI, v. 20, n. 5, p. 1500, 2020.

DANTAS, M. A. Aplicação de um robô aspirador como plataforma robótica do instituto federal do piauí. *Trabalho de Conclusão de Curso (Bacharelado em Engenharia Mecânica) - Instituto Federal de Educação, Ciência e Tecnologia do Piauí, Campus Teresina Central*, 2022. Orientador: Francisco Marcelino Almeida de Araújo. 50f. Disponível em: http://bia.ifpi.edu.br:8080/jspui/handle/123456789/4975.

DEMASHOV, D.; GOSUDAREV, I. Efficiency evaluation of node. js web-server frameworks. In: *MICSECS*. [S.l.: s.n.], 2019.

DOSUNMU, O. F. et al. A comparative study of data visualisation techniques for effective decision-making in business intelligence. *Path of Science*, v. 11, n. 8, p. 1058–1068, 2025.

GIBB, R. et al. Solving package management via hypergraph dependency resolution. *arXiv preprint arXiv:2506.10803*, 2025.

GURUNG, B. A comparative analysis of create-react-app (cra) and vite for react. js projects. 2024.

HUANG, X. Research and application of node. js core technology. In: IEEE. *2020 International Conference on Intelligent Computing and Human-Computer Interaction (ICHCI)*. [S.l.], 2020. p. 1–4.

IBM. *Diagramas de Implementação*. 2025. Disponível em: https://www.ibm.com/docs/pt-br/rsas/7.5.0?topic=topologies-deployment-diagrams. Acesso em: Acesso em: 01 jan. 2025.

International Organization for Standardization. *ISO 8373:2021 – Robots and robotic devices – Vocabulary*. 2024. Disponível em: https://www.iso.org/obp/ui/#iso:std:iso:8373:ed-3:v1:en. Acesso em: Acesso em: 20 oct. 2025.

ISLAM, M. T. Building an end-to-end e-commerce solution: designing and implementing a full-stack e-commerce website. *Tampere University of Applied Sciences*, 2023.

ISRAEL, O.; TOBILOBA, A. User experience (ux) design for upi-based digital payment apps: A study of user interface (ui) and user journey mapping. 10 2025. Disponível em: https://www.researchgate.net/publication/396497177.

JUNIOR, F. E. F. *Uma introdução ao Robot Operating System*. 2024. Disponível em: https://embarcados.com.br/uma-introducao-ao-robot-operating-system-ros/. Acesso em: 15 set. 2024.

KOUBAA, A. Ros as a service: web services for robot operating system. *Journal of Software Engineering for Robotics*, v. 6, n. 1, p. 1–14, 2015.

LALANCETTE, C.; POUBEL, L. *ROS Index*. 2022. Disponível em: https://index.ros.org/stats/. Acesso em: 15 set. 2024.

LIU, Y.; OSVALDER, A.-L.; KARLSSON, M. Considering the importance of user profiles in interface design. In: *User interfaces*. [S.l.]: IntechOpen, 2010.

MACE, J. *Rosapi*. 2024. Disponível em: https://wiki.ros.org/rosapi. Acesso em: 15 set. 2024.

MACE, J. *Rosbrdge Library*. 2024. Disponível em: https://wiki.ros.org/rosbridge_library. Acesso em: 15 set. 2024.

MACE, J. *Rosbridge Server*. 2024. Disponível em: https://wiki.ros.org/rosbridge_server. Acesso em: 15 set. 2024.

MACE, J. *Rosbridge Suite*. 2024. Disponível em: https://wiki.ros.org/rosbridge_suite. Acesso em: 15 set. 2024.

MARQUES, P. V. F. Comunicação e controle via web de um sistema embarcado utilizando o protocolo websocket e a infraestrutura serverless. Universidade Federal de Uberlândia, 2023. Trabalho de Conclusão de Curso.

MEHLBER, D. *Distributable software architecture for synthetic environment generation in real-time missile simulations and validations*. Tese (Doutorado) — Technische Hochschule Ingolstadt, 2024.

MICROSFT. *JavaScript With Syntax For Types. — typescriptlang.org*. 2024. Disponível em: https://www.typescriptlang.org/. Acesso em: 09 set. 2024.

NAIK, A. A.; KHARE, M. R. Study of “websocket protocol for real-time data transfer”. *International Reasearch Journal of Engineering and Technology*, 2020.

NILSSON, E.; DEMIR, D. *Performance comparison of REST vs GraphQL in different web environments: Node. js and Python*. 2023.

OLIVEIRA, G. M. d. Desenvolvimento e avaliação do plugin para o figma para documentação de acessibilidade para interfaces-dai. Universidade Federal de Uberlândia, 2022. Trabalho de Conclusão de Curso.

RAWAT, P.; MAHAJAN, A. N. Reactjs: a modern web development framework. *International Journal of Innovative Science and Research Technology*, v. 5, n. 11, p. 698–702, 2020.

ROLLIN, R. et al. Reducing the negative effects of cognitive overload through visual aesthetic design. 2025. Disponível em: https://aisel.aisnet.org/icis2025/hti/hti/24/.

SHAH, S. *The Untold Tale: How Facebook Engineered ReactJS — dhiwise.com*. 2024. Disponível em: https://www.dhiwise.com/post/decoding-the-evolution-of-react-facebook-innovation-journe. Acesso em: 8 set. 2024.

SULTANA, R. Artificial intelligence in data visualization: Reviewing dashboard design and interactive analytics for enterprise decision-making. *International Journal of Business and Economics Insights*, v. 5, n. 3, p. 01–29, 2025.

TEAM, F. *Fast and low overhead web framework, for Node.js | Fastify — fastify.dev*. 2025. Disponível em: https://fastify.dev/. Acesso em: 03 fev. 2025.

TEAM, M. *The ultimate guide to UML diagrams*. 2025. Disponível em: https://miro.com/diagramming/what-is-a-uml-diagram/. Acesso em: Acesso em: 21 oct. 2025.

TOOLS, R. W. *Robot Web Tools — robotwebtools.github.io*. 2024. Disponível em: https://robotwebtools.github.io/. Acesso em: 09 set. 2024.

TORIS, R. *Roslib Js*. 2023. Disponível em: https://wiki.ros.org/roslibjs. Acesso em: 15 set. 2024.

TORIS, R. *GitHub - RobotWebTools/rosbridge_suite: Server Implementations of the rosbridge v2 Protocol — github.com*. 2024. Disponível em: https://github.com/RobotWebTools/rosbridge_suite?tab=readme-ov-file. Acesso em: 15 set. 2024.

UI, C. *Chakra UI - A simple, modular and accessible component library that gives you the building blocks you need to build your React applications. — v2.chakra-ui.com*. 2024. Disponível em: https://v2.chakra-ui.com/. Acesso em: 09 set. 2024.

VERMA VISHAL SHRIVASTAVA, A. P. A.; SHRIVASTAVA, V. Introduction on nodejs and its benefits and analysis. *International Journal of Research Publication and Reviews*, v. 5, n. 3, p. 1327–1330, 2024. Disponível em: https://ijrpr.com/uploads/V5ISSUE3/IJRPR23457.pdf.

WENG, Y.; WU, J. Database management systems for artificial intelligence: Comparative analysis of postgresql and mongodb. 2024.

ZACARIOTO, J.; SILVA, G. A. da; FERNANDES, N. M. M. C. Prototipação de um software para gerenciamento de clientes e produtos de uma empresa do setor de varejo. In: *Congresso de Tecnologia-Fatec Mococa*. [S.l.: s.n.], 2022. v. 6, n. 2.

Anexos

ANEXO A – DECLARAÇÃO DE ENTREGA DO CÓDIGO FONTE DO SOFTWARE A COORDENAÇÃO DO CURSO