UNIVERSIDADE ESTADUAL PAULISTA - UNESP

Instituto de Biociências Letras e Ciências Exatas, São José do Rio Preto - campus de São José do Rio Preto

JOÃO VICTOR MILLANE

Plataforma web para controle e monitoramento de sistemas robóticos em ambiente ROS :
Uma solução para execução e supervisão de sistemas robóticos em tempo real

São José do Rio Preto
2025

João Victor Millane

Plataforma web para controle e monitoramento de sistemas robóticos em ambiente ROS :

Uma solução para execução e supervisão de sistemas robóticos em tempo real

Trabalho de Conclusão de Curso, apresentada à Universidade Estadual Paulista (UNESP), Instituto de Biociências Letras e Ciências Exatas, São José do Rio Preto , São José do Rio Preto, para obtenção do título de Bacharelem Ciência da Computação.

Orientador: Prof^{o} Dr. Diego Renan Bruno

São José do Rio Preto
2025

JOÃO VICTOR MILLANE

PLATAFORMA WEB PARA CONTROLE E MONITORAMENTO DE SISTEMAS ROBÓTICOS EM AMBIENTE ROS :

Uma solução para execução e supervisão de sistemas robóticos em tempo real

Trabalho de Conclusão de Curso apresentado à Universidade Estadual Paulista (UNESP), Instituto de Biociências Letras e Ciências Exatas, São José do Rio Preto , São José do Rio Preto, para obtenção do título de Bacharel em Ciência da Computação.

Data de defesa: 18/11/2025

BANCA EXAMINADORA

Profº Dr. Diego Renan Bruno
UNESP – Instituto de Biociências Letras e Ciências Exatas, São José do Rio Preto – Campus de São José do Rio Preto

---

Profº Dr. Rodrigo Capobiano Guido
UNESP – Instituto de Biociências Letras e Ciências Exatas, São José do Rio Preto – Campus de São José do Rio Preto

---

Profº Dr. Lucas Correia Ribas
UNESP – Instituto de Biociências Letras e Ciências Exatas, São José do Rio Preto – Campus de São José do Rio Preto

AGRADECIMENTOS

Agradeço à minha família, por sempre me incentivar a conquistar meus objetivos e aos meus colegas, em especial Vinicius Augusto Toreli Borgue, William D'Abruzzo Martins, Bruno Henrique Zara, Gabriel Leoni Duarte, Ana Beatriz Zerati e Luan Bonizi Guerra, pelo companheirismo e auxílio nesta jornada de aprendizado contínuo.

“Intelligence is the ability to avoid doing work, yet getting the work done“
(Torvalds, 2013)

RESUMO

O desenvolvimento de sistemas robóticos demanda soluções capazes de integrar módulos construídos com diferentes algoritmos e linguagens de programação, responsáveis por percepção, planejamento, controle e atuação, de modo a favorecer a coordenação entre componentes heterogêneos e a realização de tarefas complexas. Nesse cenário, apresenta-se uma interface voltada ao monitoramento, controle e execução de módulos em sistemas robóticos, cujo objetivo é oferecer um ambiente de alto nível que permita a interação simples e intuitiva entre os colaboradores e os módulos, ao mesmo tempo em que possibilita acompanhar a comunicação entre eles. A solução disponibiliza recursos de visualização do estado do sistema em tempo real, permitindo observar variáveis críticas de desempenho, identificar falhas e analisar registros de execução. Também oferece mecanismos de controle para iniciar e encerrar módulos de forma seletiva, conferindo maior flexibilidade ao gerenciamento das aplicações. Sua arquitetura foi desenvolvida para viabilizar a comunicação entre diferentes módulos a partir do Robot Operating System (ROS), amplamente utilizado na área. Experimentos conduzidos em cenários simulados e reais demonstraram que a interface reduz o tempo de configuração e a necessidade de conhecimento técnico aprofundado, aumentando a eficiência no desenvolvimento e teste de experimentos científicos. Assim, a proposta representa uma contribuição relevante para a robótica aplicada, ao simplificar o desenvolvimento e a operação de sistemas modulares e ao aproximar pesquisa e prática em ambientes acadêmicos e industriais.

Palavras-Chave: Sistemas robóticos; Interface de monitoramento e controle; Execução de módulos; Integração de componentes; ROS.

ABSTRACT

The development of robotic systems requires solutions capable of integrating modules built with different algorithms and programming languages, responsible for perception, planning, control, and actuation, in order to enable coordination among heterogeneous components and the execution of complex tasks. In this context, this work presents an interface for monitoring, control, and execution of modules in robotic systems, whose goal is to provide a high-level environment that allows collaborators to interact with the modules in a simple and intuitive way, while also enabling the monitoring of their communication. The solution provides real-time visualization of the system state, allowing the observation of critical performance variables, failure detection, and execution log analysis. It also offers control mechanisms to start and stop modules selectively, providing greater flexibility in application management. Its architecture was designed to enable communication between different modules through the Robot Operating System (ROS), widely adopted in the field. Experiments carried out in both simulated and real scenarios demonstrated that the interface reduces environment setup time and the need for extensive technical knowledge, increasing efficiency in the development and testing of scientific experiments. Therefore, the proposal represents a relevant contribution to applied robotics, by simplifying the development and operation of modular systems and by strengthening the integration between research and practice in academic and industrial environments.

Keywords: robotic systems; monitoring and control interface; module execution; component integration; ROS.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CAPES	Coordenação de Aperfeiçoamento de Pessoal de Nível Superior
DDS	Data Distribution Service
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IES	Instituição de Ensino Superior
MEC	Ministério da Educação
ROS	Robot Operating System
TCP	Transmission Control Protocol
URL	Uniform Resource Locator

LISTA DE SÍMBOLOS

t	Tempo
N	Número de nós no sistema
T	Tópico
M	Mensagem
C	Container

SUMÁRIO

1 INTRODUÇÃO

A robótica representa a convergência tangível entre o mundo digital e o mundo físico, consolidando-se como um dos pilares fundamentais da revolução tecnológica contemporânea. Os primeiros sistemas robóticos existentes focaram em executar tarefas repetitivas com alta precisão em ambientes controlados, especialmente na indústria e áreas médicas (Hockstein et al., 2007). Contudo, a partir de 1995, a robótica avançou para robôs de campo e serviço, com o objetivo geral de aproximar os robôs das necessidades sociais humanas e assisti-los; evoluindo de sistemas de controle de movimento clássicos para robôs industriais até técnicas de controle inteligente e paradigmas de aprendizado social (Garcia et al., 2007), de forma que a aplicação de sistemas robóticos se tornaram cada vez mais frequentes (Shukla; Shukla, 2012).

Sistemas robóticos, devido à sua ampla aplicação, envolvem a integração de uma variedade de software e hardware, exigindo a colaboração entre módulos de Inteligência Artificial, Sistemas de Controle, Redes e Aquisição de Dados, entre outros (Fernández-Madrigal et al., 2008). Em sistemas mais complexos, como veículos robóticos, há ainda a inclusão de componentes de percepção interna (informações do veículo), percepção externa (informações do ambiente), bem como sistemas de decisão e comunicação (Bruno et al., 2025).

Visando acelerar o desenvolvimento de sistemas robóticos, foram propostas soluções que permitem o funcionamento de módulos independentes, capazes de se comunicar entre si e formar sistemas complexos voltados a tarefas específicas. Um exemplo é o ROS (Robot Operating System) (Robot. . . , 2016), um *middleware* que atua de forma semelhante a um sistema operacional, fornecendo um conjunto de softwares e bibliotecas que facilitam a comunicação entre os programas em execução dentro de um sistema robótico.

Apesar de sua robustez, a interação com o ecossistema ROS ocorre predominantemente via Interface de Linha de Comando (CLI). Para realizar tarefas triviais, como verificar a saúde de um nó ou a frequência de um tópico, o desenvolvedor precisa memorizar e executar comandos textuais complexos. Essa dependência intrínseca do terminal reforça a necessidade de domínio do sistema operacional subjacente. Esse rigor técnico cria um gargalo na pesquisa em Robótica, dificultando a participação de pesquisadores de áreas correlatas essenciais — como engenharia, física e biologia — que não possuem necessariamente formação especializada em computação.

1.1 OBJETIVOS

Dada a contextualização e a problemática, o presente trabalho tem como objetivo desenvolver uma interface web que possibilite o monitoramento e a submissão de módulos, simplificando a contribuição de novos colaboradores e facilitando o teste de módulos em desenvolvimento em sistema ROS.

De maneira específica, os objetivos desse trabalho contemplam:

- **Viabilizar a submissão de módulos via web**: implementar um mecanismo que permita o *upload* e configuração de novos módulos robóticos, eliminando a necessidade de interação direta com o terminal ou arquivos de sistema;
- **Prover visualização gráfica dos nós**: desenvolver uma representação visual dinâmica da topologia do sistema ROS, facilitando o monitoramento do fluxo de dados e o status de execução sem exigir comandos de inspeção manual;
- **Gerenciar o ciclo de vida dos módulos**: permitir a edição, reinicialização e remoção de módulos existentes diretamente pela interface, centralizando operações de manutenção que tradicionalmente exigiriam conhecimento avançado em Linux.

1.2 ORGANIZAÇÃO DO TEXTO

Esta monografia está dividida em cinco capítulos. Neste capítulo, uma introdução e contextualização ao projeto foram dada, contendo a motivação, objetivos e sua justificativa. No Capítulo 2, realiza-se uma revisão bibliográfica sobre Sistemas Web e do sistema ROS, fornecendo a base teórica necessária, bem como uma revisão da literatura e trabalhos relacionados.

O Capítulo 3 apresenta a metodologia da monografia, incluindo a arquitetura utilizada, interface desenvolvida e ferramentas necessárias. No Capítulo 4, são apresentados e discutidos os testes e resultados obtidos, e, por fim, o Capítulo 5 conclui a monografia, destacando os principais resultados e sugerindo perspectivas para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os fundamentos teóricos que embasam o desenvolvimento da plataforma web proposta para controle e monitoramento de sistemas robóticos em ambiente ROS. Para isso, são discutidos os principais conceitos relacionados a tecnologias web, aos componentes e arquitetura do Robot Operating System (ROS), bem como aos protocolos que possibilitam a comunicação necessária.

2.1 SISTEMAS WEB

A *World Wide Web* (WWW), comumente chamada de Web, foi inicialmente desenvolvida como uma espécie de biblioteca virtual de páginas estáticas, permitindo que pesquisadores científicos compartilhassem documentos através da rede (Shklar; Rosen, 2009). Em seu estágio inicial, essas páginas apresentavam conteúdo fixo, sem capacidade de interação ou atualização automática. Diferentemente desse modelo, as páginas dinâmicas utilizam *scripts* e mecanismos de processamento para modificar seu conteúdo conforme a interação do usuário, possibilitando experiências mais ricas e personalizadas (AMELIA, 2023).

Com o avanço de novas tecnologias e protocolos de comunicação via internet, a Web passou por uma evolução significativa. Atualmente, ela não é utilizada apenas por pesquisadores, mas também por um grande número de pessoas em todo o mundo. O uso de páginas dinâmicas vem crescendo, substituindo gradualmente o modelo original voltado para conteúdos estáticos, como documentos e jornais.

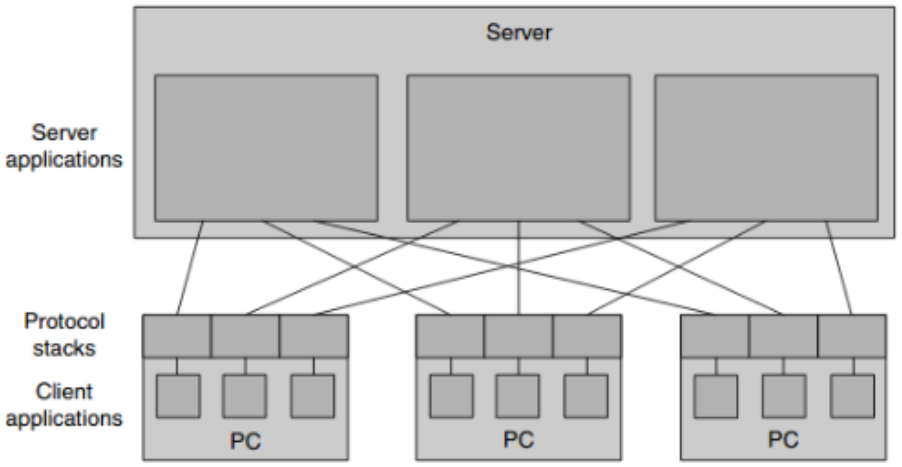
Ferramentas como motores de busca, bancos de dados, servidores e diversas aplicações permitem consultas em tempo real, retornando informações diferentes de acordo com os critérios definidos pelo usuário. Esses avanços também impulsionaram o surgimento de novas arquiteturas de aplicações, como a arquitetura cliente-servidor.

2.1.1 Arquitetura Cliente-Servidor

O modelo cliente-servidor pode ser entendido como uma organização de sistemas distribuídos, na qual diferentes computadores trocam informações utilizando protocolos de rede, conforme apresentado na Figura 1. Nesse arranjo, os servidores são responsáveis por disponibilizar serviços ou recursos, enquanto os clientes solicitam o acesso a esses elementos (Sinha, 1992). A principal vantagem desse paradigma está na divisão das tarefas de processamento entre clientes e servidores, evitando sobrecarregar um único ponto do sistema.

As aplicações web seguem exatamente essa lógica: o navegador atua como cliente, enviando pedidos ao servidor. Este, por sua vez, interpreta as solicitações recebidas e retorna uma resposta adequada. O navegador, então, transforma essa resposta em uma apresentação visual para

Figura 1 – Exemplo de aplicação cliente-servidor



Fonte: Shklar & Rosen (2003)

o usuário. Para que esse processo seja possível, três componentes essenciais precisam estar integrados, conforme destacado em (Shklar; Rosen, 2009):

- Uma linguagem de marcação que define a estrutura e a formatação das páginas exibidas.
- Um método padronizado de endereçamento que permita localizar qualquer recurso disponível na rede.
- Um protocolo de comunicação que viabiliza o transporte das mensagens entre as partes envolvidas.

Em (Jazayeri, 2007) é possível encontrar quais tecnologias desempenham esses papéis, acompanhadas de uma descrição resumida: HTML/CSS/JavaScript ,URL e HTTP.

- HTML (Hypertext Markup Language), CSS (Cascading-Style Sheets), JS (JavaScript)

  HTML trata-se da linguagem de marcação utilizada para organizar e formatar o conteúdo das páginas. Documentos em HTML podem incluir hiperlinks para outros arquivos e instruções que orientam o navegador na renderização do conteúdo. CSS trata-se da linguagem de estilização, utilizada para alterar os estilos padrões da página HTML para uma visualização mais customizada e agradável. JS trata-se da linguagem de programação utilizada para fornecer funcionalidades extras mais complexas e dinâmicas, como o carregamento automático de páginas em um site de notícias

- URL (Uniform Resource Locator)

  O URL corresponde ao padrão empregado para identificar e acessar recursos na internet. Sua estrutura normalmente contém três partes: a identificação da máquina hospedeira (via endereço IP ou DNS), a referência ao recurso solicitado e o protocolo de comunicação que será utilizado.

- HTTP (Hypertext Transfer Protocol)

  Trata-se de um protocolo que define um conjunto de regras para a troca de informações na web. Ele opera no modelo de requisição-resposta entre cliente e servidor. O HTTP também define oito métodos de requisição que podem ser utilizados: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE e CONNECT. Entre esses, os mais comuns são o GET, usado para acessar um recurso específico de um URL, e o POST, que permite enviar dados do cliente para o servidor.

2.1.1.1 Protocolo Websocket

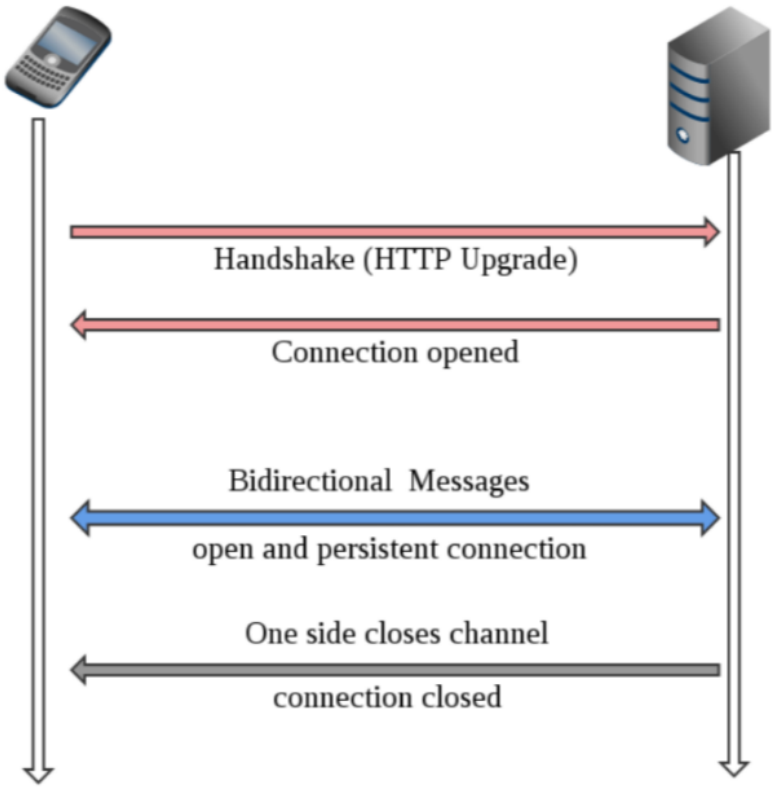
O modelo cliente-servidor, embora amplamente utilizado em sistemas web, apresenta uma grande limitação. Como discutido anteriormente, ele se baseia em um protocolo de requisição-resposta, no qual toda comunicação depende de solicitações originadas pelo cliente. Isso implica que o servidor não pode enviar informações por conta própria, sem que haja uma requisição prévia. Para contornar essa limitação e possibilitar interações mais próximas do tempo real, foram criadas algumas soluções, foram desenvolvidas algumas soluções, entre as quais se destacam o HTTP *long polling* (Pimentel; Nickerson, 2012a) e, mais recentemente, o protocolo WebSocket (Wang; Salim; Moskovits, 2013).

O HTTP *long polling* é uma técnica de comunicação empregada para simular interações quase em tempo real sobre o protocolo HTTP. Trata-se de um mecanismo *half-duplex*, no qual a troca de dados ocorre em apenas um sentido por vez. Seu funcionamento consiste em o cliente enviar uma requisição ao servidor e manter a conexão aberta até que novos dados estejam disponíveis ou até que um tempo limite seja atingido (Murley et al., 2021). Assim que a resposta é enviada, o cliente imediatamente inicia uma nova requisição, estabelecendo um ciclo contínuo que permite manter um fluxo de informações com baixa defasagem temporal, embora ainda limitado pelo modelo tradicional de requisição–resposta.

O WebSocket, por sua vez, permite comunicação *full-duplex* entre navegador e servidor, possibilitando que mensagens sejam enviadas simultaneamente em ambos os sentidos por meio de uma única conexão persistente. O estabelecimento dessa comunicação inicia-se com um processo de *handshake*, no qual o navegador solicita a abertura da conexão ao servidor. Caso a solicitação seja aceita, a conexão permanece ativa, viabilizando a troca bidirecional e em tempo real de dados. Um exemplo de utilização do WebSocket é apresentado na Figura Figura 2.

Uma das principais vantagens do WebSocket em relação ao *long polling* é referente à latência (Pimentel; Nickerson, 2012b). Como o protocolo HTTP adiciona diversos cabeçalhos às mensagens, a comunicação se torna mais lenta, já que informações extras são trocadas junto aos recursos. Além disso, o *long polling*, pela construção sobre o HTTP, é limitado pelo modelo tradicional de requisição–resposta. Essas características podem ser desfavoráveis em cenários que exigem respostas em tempo real.

Figura 2 – Modelo de comunicação WebSocket



Fonte: Mu (2016)

2.2 ROS: ROBOT OPERATING SYSTEM

O ROS é um *middleware* criado em 2007 por meio de uma colaboração entre a Universidade de Stanford, a Willow Garage e a Open Source Robotics Foundation, originalmente proposto em (Quigley et al., 2009). Ele busca permitir comunicação direta entre diferentes nós, apoiar o desenvolvimento por meio de ferramentas, ser compatível com múltiplas linguagens de programação, funcionar de forma leve e ainda ser livre e de código aberto, possibilitando o uso e a modificação sem restrições comerciais.

Sua proposta é simplificar o desenvolvimento de aplicações robóticas fornecendo bibliotecas, *drivers*, ferramentas de visualização e mecanismos de comunicação (Robot. . . , 2016).

The philosophical goals of ROS can be summarized as: • Peer-to-peer • Tools-based • Multi-lingual • Thin • Free and Open-Sourc

2.2.1 Componentes Principais

Segundo (Quigley; Gerkey; Smart, 2015), a arquitetura do ROS é composta por:

- *Drivers* para sensores e atuadores.
- Algoritmos de robótica (localização, mapeamento, planejamento de movimento).

- Uma infraestrutura de comunicação entre processos distribuídos.
- Ferramentas de visualização e depuração.
- Uma comunidade ativa e vasta documentação de suporte.

2.2.2 Funcionamento e Comunicação

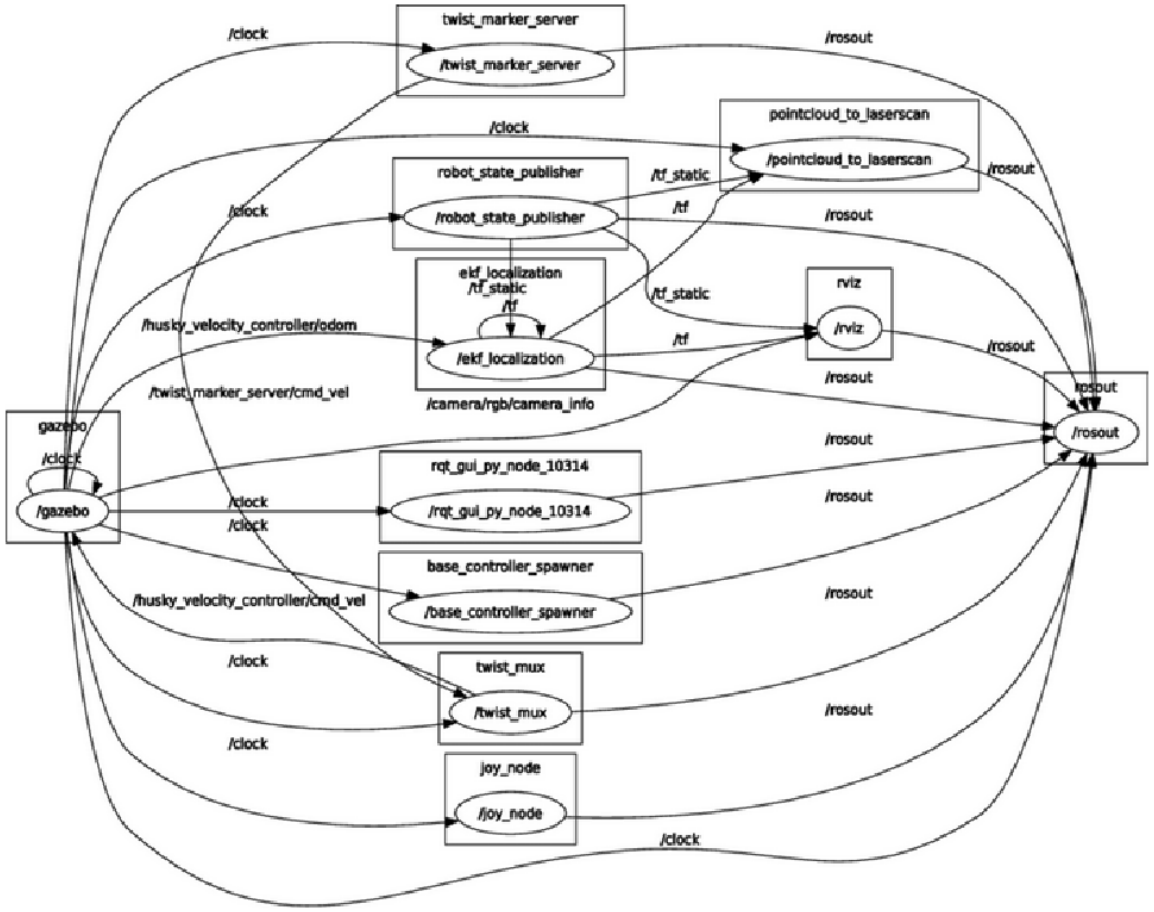
O *Robot Operating System* (ROS) adota um modelo de comunicação *peer-to-peer*, no qual os programas — denominados *nós* (*nodes*) — comunicam-se diretamente entre si, formando uma estrutura lógica em grafo. Essa arquitetura descentralizada favorece a modularidade e a escalabilidade de sistemas robóticos complexos, permitindo a integração de diversos componentes de software de forma independente.

A comunicação entre os nós pode ocorrer de três maneiras principais:

- **Tópicos (*Topics*)**: seguem o paradigma *publisher/subscriber*. Nesse modelo, um nó *publisher* envia mensagens a um tópico específico, enquanto um ou mais nós *subscribers* recebem essas mensagens de forma assíncrona. Esse mecanismo é ideal para o envio contínuo de dados, como leituras de sensores, imagens de câmeras ou comandos de controle. O ROS utiliza o sistema de mensagens para garantir a serialização e o transporte adequado entre os nós, mesmo quando executados em diferentes máquinas.
- **Serviços**: implementam um modelo de comunicação baseado em *requisição–resposta*. Um nó atua como *servidor de serviço*, disponibilizando uma funcionalidade específica, enquanto outro nó, o *cliente*, envia uma requisição e aguarda a resposta. Esse tipo de comunicação é síncrono e mais apropriado para operações pontuais, como solicitar a posição atual de um robô ou inicializar um determinado processo.
- **Ações (*Actions*)**: estendem o conceito de serviços para operações de longa duração, introduzindo mecanismos de *feedback* periódico e de cancelamento. O protocolo de ações permite que o cliente acompanhe o progresso da tarefa em execução e, se necessário, a interrompa antes da conclusão. Esse modelo é amplamente utilizado em tarefas que exigem monitoramento contínuo, como navegação autônoma, manipulação de objetos ou planejamento de trajetórias.

Em síntese, a flexibilidade desses três mecanismos de comunicação torna o ROS uma plataforma robusta e adaptável, adequada tanto para sistemas simples quanto para arquiteturas robóticas distribuídas e complexas. Um exemplo de programa usando ROS é apresentado pela Figura 3.

Figura 3 – Exemplo de programa usando ROS



Fonte: Cadavid, Ruiz & Rocha (2017)

2.2.3 ROS 2 (Kilted)

A plataforma foi desenvolvida especificamente para o ROS 2, versão *Kilted*, que representa uma evolução significativa em relação ao ROS clássico (Macenski et al., 2022). As principais vantagens do ROS 2 incluem:

- **Protocolo DDS (*Data Distribution Service*) para comunicação nativa entre nós**: o ROS 2 utiliza o padrão DDS como camada de middleware, eliminando a dependência do *ROS Master* e permitindo comunicação direta entre nós. O DDS oferece descoberta automática, suporte a múltiplos transportes de rede e um modelo de publicação/assinatura altamente escalável.
- **Suporte a tempo real e *Quality of Service* (QoS) configurável**: o ROS 2 possibilita o controle fino sobre os parâmetros de comunicação, como confiabilidade, prioridade e latência, por meio de políticas de QoS. Essa flexibilidade permite o uso em sistemas de tempo real, nos quais a previsibilidade e o desempenho determinístico são críticos.
- **Melhor suporte a ambientes de produção**: o ROS 2 foi projetado para uso industrial e

comercial, com foco em segurança, estabilidade e interoperabilidade. Inclui mecanismos de autenticação, controle de acesso e isolamento de componentes, tornando-o adequado para aplicações certificáveis e de alta confiabilidade.

- **Arquitetura distribuída aprimorada**: a nova arquitetura remove o ponto único de falha presente no ROS 1, permitindo a execução totalmente distribuída dos nós. Essa característica favorece a resiliência do sistema e possibilita o desenvolvimento de aplicações cooperativas envolvendo múltiplos robôs.
- **Comunicação via TCP/UDP e suporte a múltiplos computadores**: o ROS 2 utiliza diferentes transportes de rede, incluindo TCP e UDP, permitindo que os nós se comuniquem eficientemente em redes locais ou remotas. Essa capacidade é essencial para robôs conectados em ambientes heterogêneos ou para sistemas que exigem balanceamento de carga e processamento distribuído.

O ROS 2 utiliza DDS (Data Distribution Service) como middleware de comunicação, oferecendo descoberta automática de serviços e configuração transparente de topologia de rede. Os nós se comunicam através dos três mecanismos principais apresentados na seção anterior.

2.3 FRAMEWORK DJANGO

O Django foi escolhido como *framework* web para desenvolvimento do *backend* da aplicação devido à sua robustez, segurança incorporada, facilidade de integração com tecnologias emergentes e velocidade de desenvolvimento devido a vários módulos comuns na maioria dos projetos pre-construídos. Construído em Python, o Django fornece uma estrutura Model-Template-View (MTV) que facilita a organização do código e a manutenção de projetos (Thakur; Jadon, 2023).

Dentre as funcionalidades Django aproveitadas no desenvolvimento, destacam-se:

- Sistema de autenticação e autorização de usuários;
- Gerenciamento de sessões e segurança CSRF;
- Sistema de templates para renderização de páginas;
- Interface administrativa para gerenciamento de dados;
- Integração com bancos de dados relacionais.

O Django facilita também a integração com a Docker SDK, permitindo que o servidor web gerencie diretamente o ciclo de vida de containers Docker, incluindo criação, execução e remoção de containers para execução de código customizado.

2.4 DOCKER E CONTAINERIZAÇÃO

O Docker é uma plataforma de virtualização leve baseada em contêineres, que permite empacotar, distribuir e executar aplicações de forma isolada e portátil. Diferente das máquinas virtuais tradicionais, os contêineres do Docker compartilham o mesmo kernel do sistema operacional, o que os torna mais rápidos e eficientes em termos de recursos. Cada contêiner inclui tudo o que a aplicação precisa para funcionar — como bibliotecas, dependências e configurações — garantindo que ela seja executada de maneira idêntica em qualquer ambiente, seja em um computador local, em servidores ou em nuvens públicas e privadas. Essa abordagem facilita o desenvolvimento, a implantação e o escalonamento de sistemas complexos, além de promover maior reprodutibilidade e integração contínua em pipelines de desenvolvimento (Rad; Bhatti; Ahmadi, 2017).

No contexto desse projeto, o Docker foi utilizado para isolar e executar módulos ROS de forma independente e reproduzível. A containerização oferece várias vantagens no contexto deste projeto:

- Isolamento de processos, evitando conflitos entre módulos diferentes
- Escalabilidade horizontal, permitindo execução simultânea de múltiplos módulos
- Reprodutibilidade de ambientes, garantindo comportamento consistente
- Gestão de recursos, podendo limitar consumo de memória e CPU por container
- Portabilidade entre diferentes plataformas e servidores

O sistema implementado gera Dockerfiles dinamicamente baseado no código fornecido pelo usuário e na linguagem de programação selecionada. Para código Python, utiliza-se uma imagem base ROS Kilted e instalação automática de dependências via pip. Para JavaScript/TypeScript, além da instalação do Node.js, o sistema configura automaticamente o ambiente rclnodejs para comunicação com ROS.

Cada container é executado e integrado à rede ROS através de uma rede Docker customizada, permitindo comunicação direta entre todos os módulos ativos e com o sistema ROS central.

2.5 ROSBRIDGE E ROSLIB.JS

O Rosbridge é um pacote do ROS que fornece uma ponte de comunicação via WebSocket, permitindo que aplicações web se comuniquem com o ROS sem instalar bibliotecas nativas. Implementa um servidor WebSocket que aceita mensagens JSON e as traduz em comandos ROS, publicando e assinando tópicos em nome dos clientes conectados.

A biblioteca ROSLib.js foi desenvolvida especificamente para uso em navegadores web, oferecendo uma API JavaScript que abstrai a comunicação WebSocket com o Rosbridge. Ela fornece classes e métodos para:

- Conexão WebSocket com servidor Rosbridge;
- Publicação e assinatura de tópicos ROS;
- Chamada de serviços ROS;
- Interação com ações ROS;
- Monitoramento do estado da conexão.

A integração do ROSLib.js permite que o frontend JavaScript interaja nativamente com o ecossistema ROS através de uma API intuitiva e assíncrona, sem necessidade de bibliotecas Python ou C++ instaladas no cliente.

2.6 TRABALHOS RELACIONADOS

Diversos trabalhos na literatura buscam adaptar o ROS para um uso mais amplo e acessível. Com o objetivo de democratizar o uso do ROS entre usuários não especializados, o rosbridge foi introduzido como uma camada de abstração de middleware que trata o ROS como um back-end (Crick et al., 2016). Essa solução fornece um acesso programático simplificado, baseado em sockets, às interfaces e algoritmos do robô, convertendo services e topics do ROS em objetos JSON serializados.

No mesmo contexto, (Rajapaksha et al., 2021) apresenta uma interface web para o controle de diversas funcionalidades do robô, tanto manuais quanto autônomas, incluindo teleoperação, mapeamento (manual e autônomo utilizando GMapping), identificação de objetos, desvio de obstáculos e seguimento de rotas. A proposta facilita a execução de funcionalidades complexas que normalmente exigiriam a instalação de múltiplas bibliotecas e a interação com um ambiente Linux via terminal. Entretanto, o projeto limita-se a um ambiente de simulação controlado, não permitindo o controle direto sobre componentes do ROS em um sistema real.

Da mesma forma, visando mitigar a complexidade de uso do ROS, (Velamala; Patil; Ming, 2017) descreve o desenvolvimento e a implementação de uma Interface Gráfica do Usuário baseada em Qt para o controle de um Veículo de Superfície Autônomo (ASV) WAM-V. A interface foi projetada para oferecer uma experiência simples e intuitiva aos operadores, permitindo inicializar e finalizar processos, exibir dados do veículo (como leituras de IMU e GPS) e realizar teleoperação.

Por fim, (Fresnillo et al., 2024) apresenta uma GUI inovadora baseada na web, altamente adaptável e reconfigurável, destinada ao controle, monitoramento e configuração intuitiva de sistemas robóticos complexos baseados em ROS. A proposta aborda diretamente a acentuada curva de aprendizado do ROS e as dificuldades de gerenciar sistemas extensos via interface de linha de comando, mesmo para usuários experientes. Sua principal característica é a modularidade: a plataforma permite a incorporação seletiva de componentes independentes, oferecendo funcionalidades como lançamento e monitoramento dinâmico de módulos, controle manual

e automático do robô, visualização de dados de sensores e alarmes, além da configuração de parâmetros do sistema.

3 METODOLOGIA

Este capítulo apresenta a metodologia utilizada para o desenvolvimento da plataforma web de monitoramento e controle de sistemas robóticos. Serão descritas as ferramentas, tecnologias e estratégias adotadas para viabilizar a comunicação entre a interface web e os nós ROS, além das funcionalidades implementadas.

3.1 VISÃO GERAL DA ARQUITETURA

A arquitetura do sistema desenvolvido é composta por múltiplas camadas que interagem de forma integrada. O sistema opera em um ambiente híbrido, combinando elementos distribuídos do ROS com uma camada centralizada de comunicação via WebSocket que permite o acesso a partir de navegadores web sem necessidade de instalação de dependências ROS no cliente.

O principal desafio arquitetural enfrentado foi a conversão entre dois paradigmas de comunicação distintos: o modelo *peer-to-peer* do ROS, baseado em DDS (*Data Distribution Service*) para comunicação entre nós, e o modelo cliente-servidor centralizado através de WebSocket, necessário para interfaces web. Essa ponte é estabelecida pelo componente Rosbridge, que atua como *gateway* de protocolo, traduzindo mensagens JSON (*JavaScript Object Language*) recebidas via WebSocket em comandos e dados nativos do ROS.

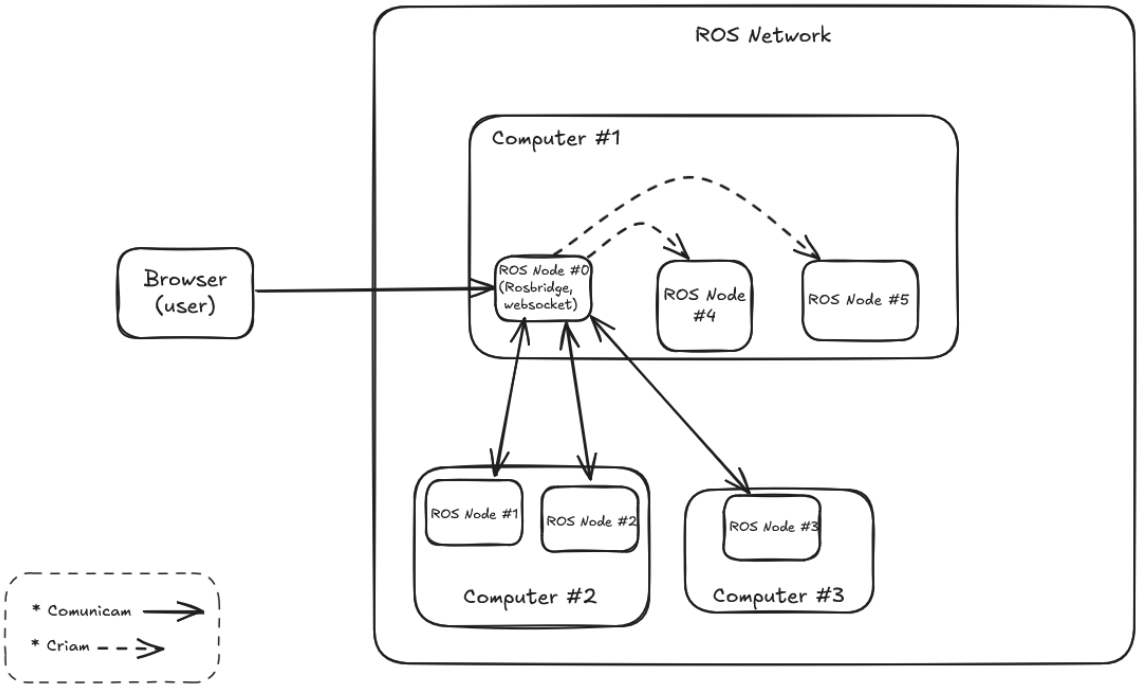
A arquitetura implementada, representada na Figura 4, contempla:

- **Camada de apresentação**: Interface web responsiva acessível via navegador.
- **Camada de aplicação**: Servidor Django gerenciando usuários, sessões e orquestração de contêineres.
- **Camada de comunicações**: Rosbridge como ponte entre WebSocket e ROS nativo.
- **Camada de execução**: Containers Docker isolados executando módulos ROS.
- **Camada de infraestrutura**: PostgreSQL para persistência, Redis para cache e filas.

3.2 FERRAMENTAS E TECNOLOGIAS UTILIZADAS

O desenvolvimento da plataforma demandou a integração de diversas tecnologias e ferramentas, cada uma desempenhando um papel específico na arquitetura do sistema. Esta seção detalha as principais tecnologias adotadas, suas justificativas de escolha e como foram integradas no projeto.

Figura 4 – Diagrama da arquitetura do sistema com as camadas de comunicação.



Fonte: Elaborada pelo autor.

3.2.1 Framework Django

O Django foi escolhido como *framework* web para o desenvolvimento do *backend* devido a suas características que se alinham perfeitamente com os requisitos do projeto. Como *framework* de alto nível construído em Python, o Django oferece uma arquitetura Model-Template-View (MTV) que facilita a organização do código e a manutenção do projeto.

Dentre as funcionalidades Django aproveitadas no desenvolvimento, destacam-se:

- **Sistema de autenticação e autorização**: O Django fornece um sistema robusto de gerenciamento de usuários, incluindo autenticação, autorização e controle de sessões. Isso permite que a plataforma gerencie múltiplos usuários simultaneamente, cada um com seus próprios containers e recursos isolados.

- **Integração com Docker SDK**: A biblioteca `docker` do Python permite que o servidor Django gerencie diretamente o ciclo de vida de containers Docker. O servidor web pode criar, iniciar, parar e remover containers programaticamente, utilizando a API do Docker através do socket Unix localizado em `/var/run/docker.sock`.

- **Sistema de templates**: O Django Template Language (DTL) permite a renderização dinâmica de páginas HTML, facilitando a criação de interfaces que se adaptam aos dados do sistema.

- **Interface administrativa**: O Django Admin fornece uma interface pronta para gerenciamento de dados, útil durante o desenvolvimento e para administração do sistema.

A integração com o Docker SDK é particularmente relevante, pois permite que o servidor Django execute operações como construção de imagens, criação de containers e gerenciamento de redes Docker de forma programática, sem necessidade de comandos de terminal ou scripts externos.

3.2.2 Docker e Orquestração de Containers

O Docker foi fundamental para o isolamento e execução de módulos ROS. A plataforma utiliza Docker Compose para orquestração de todos os serviços, garantindo que componentes como PostgreSQL, Redis, servidor web Django e Rosbridge sejam iniciados de forma coordenada e compartilhem a mesma rede Docker.

A rede Docker customizada, denominada `mrfi_ros`, é essencial para permitir comunicação entre todos os containers. Quando um container é criado para executar código customizado, ele é automaticamente conectado a esta rede, permitindo que o módulo ROS se comunique com outros nós através do protocolo DDS nativo do ROS 2.

O processo de geração dinâmica de Dockerfiles é implementado no servidor Django. Quando um usuário faz *upload* de código, o sistema:

1. Cria um diretório temporário para armazenar os arquivos enviados;
2. Gera um Dockerfile baseado na linguagem selecionada (Python ou JavaScript);
3. Para Python: utiliza imagem base `osrf/ros:kilted-desktop` e instala dependências via pip;
4. Para JavaScript: instala Node.js, TypeScript e a biblioteca `rclnodejs` para comunicação com ROS;
5. Constrói a imagem Docker utilizando a API do Docker;
6. Inicia o container com limitação de memória por máquina e conecta-o à rede ROS.

Cada container é rotulado com informações do usuário proprietário e uma descrição fornecida pelo usuário, permitindo rastreamento e gerenciamento individualizado de recursos.

3.2.3 Rosbridge e Integração WebSocket

O Rosbridge é um pacote ROS que implementa um servidor WebSocket, permitindo que aplicações web se comuniquem com o ROS sem instalar bibliotecas nativas. No projeto, o Rosbridge é executado em um container Docker separado, expondo a porta 9090 para conexões WebSocket.

A configuração do Rosbridge inclui o nó `rosapi_node`, que fornece serviços ROS adicionais para consulta de metadados do sistema, como listagem de nós ativos, identificação de tópicos e obtenção de detalhes sobre tipos de mensagens. Esses serviços são essenciais para a funcionalidade de visualização do grafo de comunicação.

O Rosbridge atua como um *proxy* bidirecional: mensagens JSON recebidas via WebSocket são convertidas para mensagens ROS nativas e publicadas nos tópicos apropriados; mensagens ROS publicadas por outros nós são capturadas pelo Rosbridge, convertidas para JSON e enviadas aos clientes WebSocket conectados.

3.2.4 Biblioteca ROSLib.js e Frontend

A biblioteca ROSLib.js foi utilizada no frontend para abstrair a comunicação WebSocket com o Rosbridge. Ela fornece uma API JavaScript orientada a objetos que simplifica a interação com o ROS, permitindo que o código JavaScript trabalhe com conceitos ROS (tópicos, serviços, ações) sem lidar diretamente com os detalhes do protocolo WebSocket.

O frontend mantém uma instância global do objeto `ROSLIB.Ros`, que gerencia a conexão WebSocket. Eventos de conexão, erro e desconexão são tratados para atualizar a interface do usuário e garantir que o sistema tente reconectar automaticamente em caso de falha.

Para a visualização do grafo, a biblioteca Cytoscape.js foi escolhida devido à sua capacidade de renderizar grafos complexos com algoritmos de layout automático. O algoritmo COSE (*Compound Spring Embedder*) foi configurado com parâmetros específicos para otimizar a visualização de topologias ROS, agrupando nós relacionados e minimizando sobreposições.

3.3 FLUXO DE DADOS E COMUNICAÇÃO

3.3.1 Comunicação ROS Nativa

O ROS 2 opera em um modelo distribuído e *peer-to-peer*, onde cada nó estabelece conexões diretas com seus pares para troca de mensagens. A descoberta de nós e tópicos é automática através do DDS (*Data Distribution Service*), eliminando a necessidade de configuração manual de rede ou de um nó central coordenador (como o *ROS Master* do ROS 1).

O processo de descoberta DDS funciona da seguinte forma:

1. Quando um nó ROS é iniciado, ele anuncia sua presença na rede através de mensagens de descoberta DDS;

2. Outros nós na mesma rede recebem essas mensagens e identificam os tópicos, serviços e ações disponíveis;

3. Quando um nó deseja publicar em um tópico, ele anuncia sua intenção através da infraestrutura DDS;

4. Nós que desejam assinar aquele tópico são notificados automaticamente e estabelecem conexões diretas TCP ou UDP com o publicador;

5. A comunicação subsequente ocorre diretamente entre os nós, sem intermediários.

Essa arquitetura oferece baixa latência e alta taxa de transferência, ideal para sistemas robóticos com requisitos de tempo real. A ausência de um ponto único de falha (como o *ROS Master*) torna o sistema mais robusto e escalável.

No contexto deste projeto, todos os containers Docker são conectados à mesma rede Docker (`mrfi_ros`), permitindo que o DDS descubra automaticamente todos os nós ROS em execução, independentemente do container em que estão hospedados.

3.3.2 Conversão para WebSocket

O Rosbridge atua como um *gateway* centralizado, convertendo o modelo distribuído do ROS para um modelo cliente-servidor compatível com navegadores web. Esta conversão é fundamental para permitir que aplicações web interajam com o ROS sem necessidade de bibliotecas nativas instaladas no cliente.

O fluxo de comunicação bidirecional segue os seguintes passos detalhados:

1. **Estabelecimento da conexão**: O navegador estabelece uma conexão WebSocket com o servidor Rosbridge na porta 9090. O handshake inicial segue o protocolo WebSocket padrão, iniciando como uma requisição HTTP que é atualizada para WebSocket.

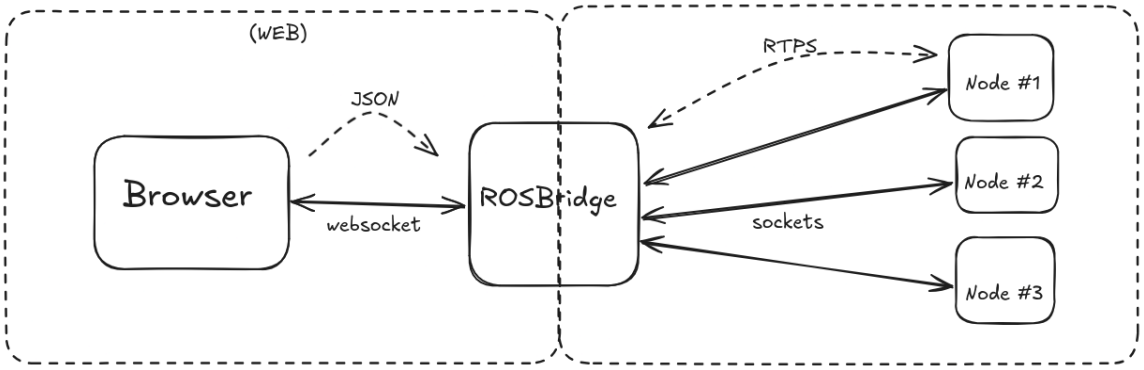
2. **Publicação de mensagens (cliente para ROS)**: Quando o cliente JavaScript deseja publicar uma mensagem em um tópico ROS:

    - O código JavaScript utiliza a biblioteca ROSLib.js para criar um objeto do tipo `ROSLIB.Topic`;
    - Uma mensagem JSON é construída contendo o nome do tópico, tipo de mensagem e dados;
    - A mensagem JSON é enviada via WebSocket para o Rosbridge;
    - O Rosbridge recebe a mensagem JSON e a converte para o formato de mensagem ROS nativo;
    - O Rosbridge atua como um nó ROS, publicando a mensagem no tópico solicitado;
    - Outros nós ROS que assinam aquele tópico recebem a mensagem através do DDS.

3. **Assinatura de tópicos (ROS para cliente)**: Quando o cliente JavaScript deseja receber mensagens de um tópico ROS:

    - O código JavaScript cria um objeto `ROSLIB.Topic` e registra um callback;

Figura 5 – Diagrama de sequência mostrando conversão de protocolo entre WebSocket e ROS



Fonte: Elaborado pelo autor.

- Uma mensagem JSON de assinatura é enviada via WebSocket para o Rosbridge;
- O Rosbridge recebe a solicitação e se inscreve no tópico ROS solicitado;
- Quando outros nós ROS publicam mensagens naquele tópico, o Rosbridge as recebe através do DDS;
- O Rosbridge converte cada mensagem ROS recebida para JSON;
- As mensagens JSON são enviadas via WebSocket para o cliente;
- O callback JavaScript é executado com os dados recebidos, atualizando a interface.

4. **Chamada de serviços**: Para chamadas de serviços ROS (comunicação síncrona request/reply):
   - O cliente JavaScript cria um objeto `ROSLIB.Service` e especifica o nome e tipo do serviço;
   - Uma requisição JSON é enviada via WebSocket;
   - O Rosbridge converte a requisição e chama o serviço ROS;
   - A resposta do serviço é convertida para JSON e retornada ao cliente.

Essa conversão permite que a interface web visualize e interaja com todos os componentes do sistema ROS, mantendo sincronização em tempo real através de uma única conexão WebSocket persistente. A latência introduzida pela conversão JSON é aceitável para aplicações de monitoramento, embora não seja adequada para controle de tempo crítico.

3.3.3 Interface Web

A interface web foi desenvolvida utilizando HTML5, JavaScript e a biblioteca Cytoscape.js para visualização de grafos. O frontend mantém uma conexão WebSocket persistente com o servidor Rosbridge e monitora ativamente o estado do sistema ROS.

3.3.3.1 Visualização do Grafo de Comunicação

Para visualizar a topologia de comunicação, a interface implementa um algoritmo que consulta periodicamente os serviços ROS expostos pelo ROSAPI. O processo de construção do grafo segue os seguintes passos:

1. **Listagem de nós**: O frontend chama o serviço `/rosapi/nodes` para obter uma lista de todos os nós ROS ativos no sistema.

2. **Filtragem de nós do sistema**: Nós relacionados à infraestrutura do Rosbridge (como `/rosapi_node` e `/rosbridge_websocket`) são filtrados, focando apenas nos módulos de aplicação do usuário.

3. **Obtenção de detalhes dos nós**: Para cada nó identificado, o frontend chama o serviço `/rosapi/node_details`, passando o nome do nó como parâmetro. Este serviço retorna informações sobre:

    - Tópicos publicados pelo nó;
    - Tópicos assinados pelo nó;
    - Serviços oferecidos pelo nó;
    - Tipos de mensagens utilizados.

4. **Construção do grafo**: Com as informações coletadas, o frontend constrói uma representação em grafo onde:

    - Nós ROS são representados como vértices verdes;
    - Tópicos são representados como vértices retangulares laranjas;
    - Arestas direcionadas conectam publicadores aos tópicos (nó $\rightarrow$ tópico);
    - Arestas direcionadas conectam tópicos aos assinantes (tópico $\rightarrow$ nó).

5. **Renderização visual**: A biblioteca Cytoscape.js renderiza o grafo utilizando o algoritmo de layout COSE, que posiciona os elementos de forma hierárquica e legível, minimizando sobreposições e agrupando nós relacionados.

O sistema atualiza automaticamente a visualização quando detecta mudanças na topologia, como novos nós conectados ou desconectados. Tópicos do sistema, como `/parameter_events` e `/rosout`, são filtrados. Isso mantém a visualização focada nos módulos de aplicação do usuário.

3.3.3.2 Gerenciamento de Estado da Conexão

A interface monitora continuamente o estado da conexão WebSocket através de event listeners:

- **Evento `connection`**: Quando a conexão é estabelecida, o status é atualizado para "Connected"e a função de atualização do grafo é chamada;
- **Evento `error`**: Em caso de erro na conexão, o status é atualizado para "Error"e a interface indica o problema ao usuário;
- **Evento `close`**: Quando a conexão é fechada, o status é atualizado para "Disconnected"e o sistema pode tentar reconectar automaticamente.

Esta monitoração permite que o usuário tenha feedback imediato sobre o estado da comunicação com o sistema ROS, facilitando a identificação de problemas de conectividade.

3.4 FUNCIONALIDADES IMPLEMENTADAS

3.4.1 Visualização de Grafo de Comunicação

A funcionalidade principal da interface permite visualizar em tempo real a topologia de comunicação do sistema ROS como um grafo interativo. Os nós são representados graficamente e conectados aos tópicos por arestas que indicam a direção do fluxo de dados.

O sistema filtra automaticamente nós e tópicos relacionados ao próprio Rosbridge, focando apenas nos módulos de aplicação do usuário. A visualização atualiza-se dinamicamente conforme novos nós se conectam ou desconectam da rede ROS.

3.4.2 Submissão de Código Personalizado

O sistema permite que usuários enviem código Python ou JavaScript para execução como nó ROS. Esta funcionalidade é implementada através de uma *view* Django que processa o *upload* de arquivos e gerencia o ciclo de vida dos containers Docker.

3.4.2.1 Processo de Upload e Validação

Quando um usuário submete código através da interface web, o servidor Django recebe os seguintes dados:

- Arquivo de código principal (obrigatório);
- Arquivo de dependências opcional (requirements.txt para Python ou package.json para JavaScript);
- Linguagem de programação selecionada;

- Descrição do módulo (máximo de 200 caracteres).

O sistema valida os dados recebidos, verificando se o arquivo de código foi fornecido e se a descrição não excede o limite de caracteres. Em caso de erro, mensagens apropriadas são retornadas ao usuário através do sistema de mensagens do Django.

3.4.2.2 Geração Dinâmica de Dockerfile

Após a validação, o sistema gera um Dockerfile dinamicamente baseado na linguagem selecionada. O processo utiliza um diretório temporário para armazenar os arquivos antes da construção da imagem Docker.

Para código Python:

- Utiliza a imagem base `osrf/ros:kilted-desktop`, que já contém o ROS 2 Kilted instalado;
- Copia o arquivo de código para o diretório `/app` do container;
- Se um arquivo de dependências (requirements.txt) foi fornecido, instala as dependências via pip;
- Define o comando de execução como `python3 nome_do_arquivo.py`.

Para código JavaScript/TypeScript:

- Utiliza a mesma imagem base ROS;
- Instala Node.js versão 20.x através do repositório oficial;
- Instala TypeScript e ts-node globalmente;
- Configura o ambiente ROS e instala a biblioteca `rclnodejs` para comunicação com ROS;
- Gera um arquivo `tsconfig.json` com configurações apropriadas;
- Se um arquivo package.json foi fornecido, instala as dependências via npm;
- Define o comando de execução como `npx ts-node nome_do_arquivo.ts`.

3.4.2.3 Construção e Execução do Container

Após a geração do Dockerfile, o sistema utiliza a API do Docker para construir a imagem:

1. A imagem Docker é construída a partir do diretório temporário contendo o Dockerfile e os arquivos do usuário;

2. Os logs de construção são capturados para possível análise de erros;
3. Se a construção for bem-sucedida, o sistema verifica se existem containers anteriores criados a partir da mesma imagem e os remove para evitar conflitos;
4. Um novo container é iniciado com as seguintes configurações:
    - Modo *detached* (execução em segundo plano);
    - Limitação de memória por máquina;
    - Conexão à rede Docker `mrfi_ros`;
    - Labels Docker contendo o nome de usuário do proprietário e a descrição fornecida;
    - Captura de stdout e stderr para possível visualização de logs.

Uma vez iniciado, o container executa o código fornecido como um nó ROS. O código deve implementar a lógica ROS apropriada (publicação/assinatura de tópicos, chamada de serviços, etc.) utilizando as bibliotecas ROS da linguagem selecionada.

3.4.2.4 Integração Automática com a Rede ROS

Como todos os containers são conectados à mesma rede Docker (`mrfi_ros`), o DDS do ROS 2 descobre automaticamente o novo nó assim que ele é iniciado. O nó aparece imediatamente na visualização do grafo da interface web, permitindo que o usuário monitore sua execução em tempo real.

Esta integração transparente elimina a necessidade de configuração manual de variáveis de ambiente ROS, descoberta de serviços ou configuração de rede, simplificando significativamente o processo de desenvolvimento e teste de módulos ROS.

3.4.3 Gerenciamento de Containers

A interface fornece uma área de gerenciamento onde usuários podem visualizar e controlar todos os containers em execução associados à sua conta. Esta funcionalidade é implementada através de uma *view* Django que consulta a API do Docker para obter informações sobre containers.

3.4.3.1 Listagem de Containers

A listagem de containers é implementada através de uma *ListView* do Django que:

1. Consulta a API do Docker para obter todos os containers (incluindo os parados);
2. Filtra os containers utilizando labels Docker, retornando apenas aqueles cujo label `owner` corresponde ao nome de usuário do usuário autenticado;

3. Para cada container, extrai informações relevantes:
    - Identificador curto do container (12 primeiros caracteres);
    - Identificador curto da imagem utilizada;
    - Status atual (running, exited, etc.);
    - Descrição fornecida pelo usuário (armazenada no label `description`).
4. Renderiza as informações em uma tabela HTML para visualização pelo usuário.

Esta abordagem baseada em labels Docker permite que múltiplos usuários utilizem o sistema simultaneamente, cada um vendo apenas seus próprios containers, garantindo isolamento de recursos e privacidade.

3.4.3.2 Remoção de Containers

O sistema permite que usuários removam containers específicos através de um formulário POST. O processo de remoção:

1. Recebe o identificador do container a ser removido;
2. Obtém o container através da API do Docker;
3. Remove o container forçadamente (utilizando a flag `force=True`), garantindo que o container seja parado e removido mesmo se estiver em execução;
4. Redireciona o usuário de volta à lista de containers.

A remoção de um container interrompe imediatamente a execução do nó ROS correspondente, desconectando-o da rede ROS. O nó desaparece automaticamente da visualização do grafo, refletindo a mudança na topologia do sistema.

O tratamento de erros garante que falhas na remoção (por exemplo, se o container já foi removido) não interrompam a experiência do usuário, redirecionando-o silenciosamente de volta à lista.

3.5 PROCESSO DE DESENVOLVIMENTO E DECISÕES DE DESIGN

Esta seção apresenta as principais decisões de design tomadas durante o desenvolvimento da plataforma e os processos adotados para garantir a qualidade e manutenibilidade do código.

3.5.1 Arquitetura de Software

A arquitetura do sistema foi projetada seguindo princípios de separação de responsabilidades e modularidade. O backend Django é organizado em aplicações (*apps*), cada uma responsável por uma funcionalidade específica:

- **Aplicação `pages`**: Contém as *views*, modelos e templates relacionados às páginas principais da interface (home, código, logs, configurações);
- **Aplicação `up`**: Fornece um endpoint de saúde (*healthcheck*) para monitoramento do sistema;
- **Configuração centralizada**: As configurações do Django, incluindo banco de dados, cache e middleware, são centralizadas no módulo `config`.

3.5.2 Segurança e Isolamento

O sistema implementa várias camadas de segurança e isolamento:

- **Autenticação obrigatória**: Todas as funcionalidades que interagem com containers ou o sistema ROS requerem autenticação através do sistema de autenticação do Django;
- **Isolamento por usuário**: Containers são associados a usuários através de labels Docker, garantindo que cada usuário veja e gerencie apenas seus próprios recursos;
- **Limitação de recursos**: Cada container é iniciado com limitação de memória por máquina, prevenindo que um container consuma recursos excessivos;
- **Validação de entrada**: O sistema valida todos os dados de entrada do usuário, incluindo tamanho de arquivos e comprimento de descrições, prevenindo ataques de injeção ou sobrecarga do sistema.

3.5.3 Orquestração com Docker Compose

A orquestração de todos os serviços é gerenciada através do Docker Compose, que define:

- **Serviços principais**: Web (Django), PostgreSQL, Redis, Rosbridge;
- **Serviços auxiliares**: Worker (Celery) para processamento assíncrono, serviços de build de assets (JavaScript e CSS);
- **Rede compartilhada**: Todos os serviços são conectados à rede `ros`, permitindo comunicação entre componentes;
- **Volumes persistentes**: Dados do PostgreSQL e Redis são armazenados em volumes Docker, garantindo persistência entre reinicializações;

- **Healthchecks**: O serviço web possui um healthcheck configurado para monitoramento automático da saúde do sistema.

Esta abordagem simplifica significativamente o processo de implantação e manutenção do sistema, permitindo que todos os componentes sejam iniciados com um único comando e garantindo que as dependências sejam resolvidas corretamente.

4 RESULTADOS

Este capítulo apresenta os resultados obtidos com a implementação da plataforma de monitoramento e controle de sistemas robóticos, incluindo descrição dos testes realizados, comportamento observado da interface e análise do impacto na facilidade de uso.

4.1 AMBIENTE DE TESTES

Os testes foram realizados em um ambiente de desenvolvimento local utilizando Docker Compose para orquestração de todos os serviços. O ambiente compreende:

- Servidor Django com PostgreSQL e Redis
- Servidor Rosbridge expondo porta 9090 para conexões WebSocket
- Nós de demonstração (talker e listener) do ROS 2
- Navegador web moderno para acesso à interface

A infraestrutura Docker Compose garante que todos os componentes ROS compartilhem a mesma rede, permitindo comunicação transparente entre nós distribuídos e o gateway Rosbridge. Os testes foram conduzidos em ambiente simulado utilizando os pacotes de demonstração do ROS 2.

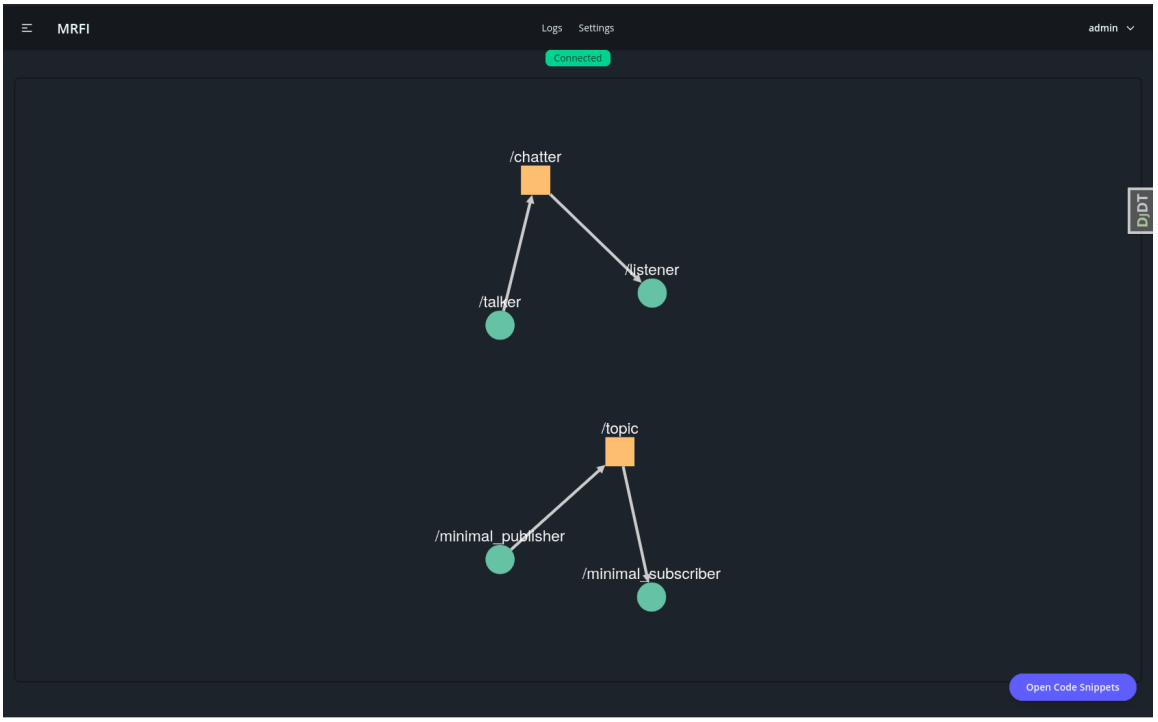
4.2 VISUALIZAÇÃO EM TEMPO REAL

A funcionalidade de visualização do grafo de comunicação demonstrou eficácia na representação da topologia do sistema ROS. Durante os testes, observou-se que:

- A conexão WebSocket estabelece-se automaticamente ao carregar a página;
- O sistema consulta periodicamente o estado do ROS através dos serviços ROSAPI;
- Novos nós aparecem dinamicamente no grafo quando conectados ao sistema;
- A remoção de nós remove automaticamente suas representações visuais;
- Os tópicos são representados como nós intermediários conectando publicadores e assinantes.

O algoritmo de layout COSE redistribui automaticamente os elementos do grafo sempre que a topologia muda, mantendo uma visualização organizada e legível. Os nós representando componentes do usuário são destacados em verde, enquanto tópicos são representados em laranja, facilitando a distinção visual, como representado na Figura 6.

Figura 6 – Interface web mostrando visualização do grafo de comunicação ROS em tempo real



Fonte: Elaborado pelo autor

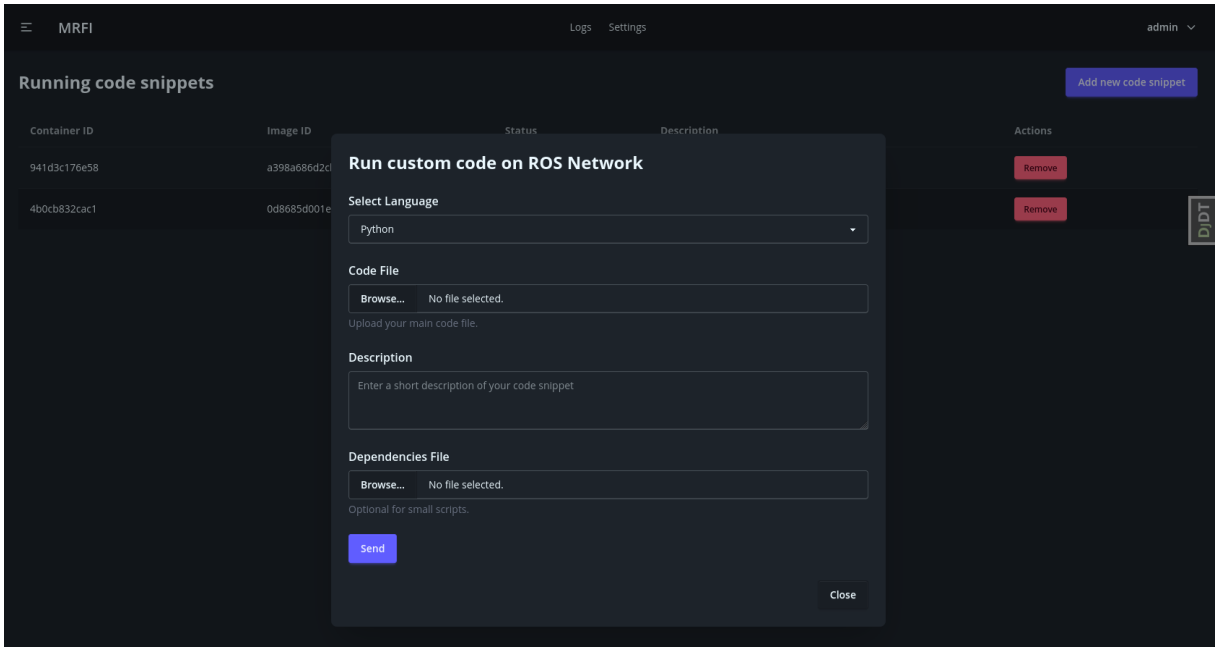
4.3 EXECUÇÃO DE CÓDIGO CUSTOMIZADO

Os testes de submissão de código customizado foram realizados criando um nó *publisher* simples que publica mensagens em um tópico e outro que se inscreve nas mensagens publicadas. O processo completo de submissão, construção e execução pode ser feito para diversas linguagens de programação, facilitando o uso, como pode ser visto na Figura 7.

1. Upload do arquivo Python contendo código ROS publisher
2. Geração automática do Dockerfile pela aplicação
3. Build da imagem Docker completada em aproximadamente 30-60 segundos
4. Container iniciado e conectado automaticamente à rede ROS
5. Novo nó aparecendo automaticamente na visualização do grafo
6. Publicações do nó customizado sendo recebidas pelos listeners existentes

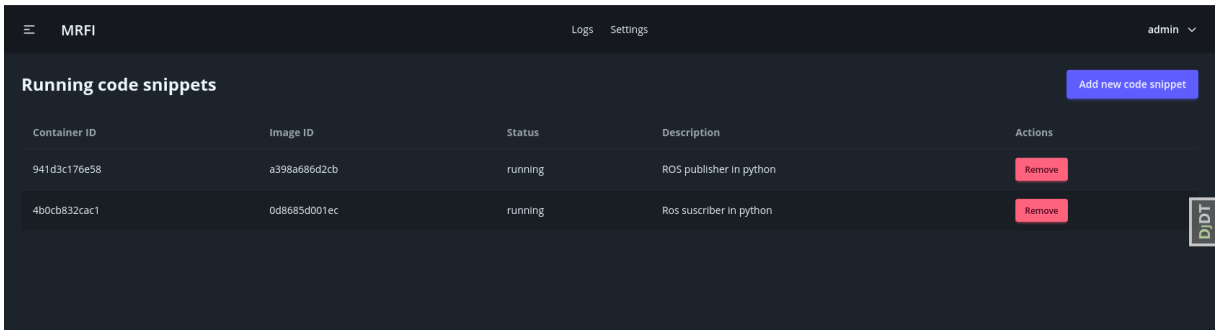
A interface web permite verificar que o container foi criado com sucesso, como pode ser visto na Figura 8, mostrando seu identificador, status e descrição. A remoção do contêiner também funciona, desconectando o nó da rede ROS e removendo-o do grafo.

Figura 7 – Tabela mostrando tela de criação de containers



Fonte: Elaborado pelo autor.

Figura 8 – Tabela mostrando containers em execução



Fonte: Elaboração própria

4.4 REDUÇÃO DE BARREIRAS TÉCNICAS

Uma análise comparativa entre o processo tradicional de desenvolvimento em ROS e a utilização da plataforma web proposta evidencia uma redução significativa nas barreiras de entrada para novos usuários e colaboradores. Enquanto o fluxo convencional demanda configuração manual e conhecimento técnico avançado, a abordagem baseada em uma interface web automatiza etapas críticas e oferece uma experiência mais acessível e integrada.

Processo Tradicional:

- Requer a instalação completa do ROS no sistema local do usuário, incluindo dependências específicas de versão e pacotes adicionais;
- Exige familiaridade com o ambiente Linux e com o uso de ferramentas de linha de

comando (CLI);

- Necessita da configuração manual de *workspaces* e pacotes, além da definição de variáveis de ambiente;
- Envolve processos de compilação e *build* manuais, suscetíveis a falhas de dependências;
- Pode demandar o uso de Docker ou outras ferramentas de virtualização para isolamento de ambientes e prevenção de conflitos de bibliotecas;
- Implica configuração manual de rede, tópicos e descoberta de serviços para comunicação entre nós distribuídos.

Com a Plataforma Proposta:

- Permite acesso direto via navegador, eliminando a necessidade de instalação local de ROS ou dependências adicionais;
- Oferece uma interface gráfica intuitiva, dispensando o uso de comandos de terminal;
- Realiza a configuração automática de ambientes e dependências por meio de contêineres Docker gerenciados pelo sistema;
- Automatiza os processos de compilação, *build* e execução, reduzindo erros e tempo de configuração;
- Proporciona integração transparente com a rede ROS, permitindo comunicação entre nós locais e remotos;
- Inclui visualização gráfica em tempo real da topologia do sistema, facilitando o monitoramento e a depuração.

Essa simplificação do fluxo de desenvolvimento reduz drasticamente a complexidade técnica envolvida, viabilizando a participação de colaboradores de diferentes áreas — mesmo sem formação em engenharia de software ou robótica —, que podem concentrar seus esforços na implementação da lógica de negócio e na funcionalidade dos módulos.

4.5 DESEMPENHO E LATÊNCIA

Observações sobre o desempenho do sistema durante os testes:

- A latência entre publicações ROS e atualização na interface web é aceitável para monitoramento (inferior a 1 segundo);
- A conexão WebSocket mantém-se estável durante sessões longas de uso;

- O build de containers Docker para código Python é rápido, com resultados entre 30 a 60 segundos;
- A visualização do grafo é fluida mesmo com múltiplos nós simultâneos.

Limitações identificadas:

- A construção dinâmica de imagens Docker adiciona overhead inicial à execução de código;
- A escalabilidade horizontal está limitada pelos recursos do servidor;
- Não há suporte a depuração interativa dos nós executados;
- A interface não fornece *logs* em tempo real da execução dos *containers*

5 CONCLUSÃO

O presente trabalho apresentou o desenvolvimento de uma plataforma web para monitoramento e controle de sistemas robóticos utilizando ROS (Robot Operating System), com foco em reduzir barreiras técnicas e facilitar a colaboração em projetos robóticos. A solução implementada combina com sucesso o modelo distribuído do ROS com uma camada centralizada de comunicação via WebSocket, permitindo acesso intuitivo a partir de navegadores web sem necessidade de instalação de dependências ROS no cliente. A integração do Django como framework backend, Docker para conteinerização de módulos, e Rosbridge como ponte de protocolo, resultou em uma arquitetura robusta que abstrai a complexidade de configuração de ambientes ROS, reduzindo significativamente o tempo necessário para iniciar desenvolvimento de módulos robóticos.

As principais contribuições deste trabalho incluem: (i) uma interface web que visualiza dinamicamente a topologia de comunicação ROS em tempo real, facilitando a compreensão da arquitetura do sistema; (ii) mecanismo automatizado de submissão, build e execução de código customizado em contêineres isolados; (iii) simplificação do processo de desenvolvimento para colaboradores sem formação especializada em robótica ou engenharia de software; e (iv) abstração completa da complexidade de redes distribuídas através de uma interface visual intuitiva. A plataforma demonstrou eficácia em cenários de testes, permitindo a execução simultânea de múltiplos módulos ROS com integração transparente e visualização imediata de mudanças na topologia do sistema.

Quanto aos trabalhos futuros, identificam-se diversas oportunidades de expansão e aprimoramento da plataforma. Destacam-se a implementação de autenticação robusta com controle de acesso granular baseado em roles, permitindo que diferentes usuários tenham permissões específicas sobre módulos e recursos do sistema. O suporte a mais linguagens de programação, como C++ e Rust, ampliaria a flexibilidade de desenvolvimento. Ferramentas de depuração avançadas, incluindo suporte a *breakpoints* e visualização de variáveis em tempo real, facilitariam o desenvolvimento de módulos mais complexos. A integração de métricas de desempenho e *profiling* em tempo real auxiliaria na otimização de algoritmos robóticos. Por fim, a integração com simuladores populares como Gazebo e Webots permitiria a validação de algoritmos em ambientes virtuais antes da execução em hardware real, aprimorando significativamente o ciclo de desenvolvimento e testes.

REFERÊNCIAS

AMELIA. **Static vs Dynamic Website: What Is the Difference?** Jovana Smoljanovic Tucakov, 2023. Disponível em: https://wpamelia.com/static-vs-dynamic-website/. Acesso em: 21 set. 2025.

BRUNO, D. et al. Robotic visual attention architecture for adas in critical embedded systems for smart vehicles. In: **Proceedings of the 20th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications**. SCITEPRESS - Science and Technology Publications, 2025. p. 871–878. Disponível em: http://dx.doi.org/10.5220/0013362600003912.

CADAVID, H.; RUIZ, A. P.; ROCHA, C. Reliable control architecture with plexil and ros for autonomous wheeled robots. **Software Engineering Research**, p. 611–626, Agosto 2017.

CRICK, C. et al. Rosbridge: Ros for non-ros users. In: ____. **Robotics Research**. Springer International Publishing, 2016. p. 493–504. ISBN 9783319293639. Disponível em: http://dx.doi.org/10.1007/978-3-319-29363-9_28.

FERNáNDEZ-MADRIGAL, J.-A. et al. A software engineering approach for the development of heterogeneous robotic applications. **Robotics and Computer-Integrated Manufacturing**, Elsevier BV, v. 24, n. 1, p. 150–166, fev. 2008. ISSN 0736-5845. Disponível em: http://dx.doi.org/10.1016/j.rcim.2006.10.002.

FRESNILLO, P. M. et al. An open and reconfigurable user interface to manage complex ros-based robotic systems. **IEEE Access**, v. 12, p. 114601–114617, 2024.

GARCIA, E. et al. The evolution of robotics research. **IEEE Robotics Automation Magazine**, v. 14, n. 1, p. 90–103, 2007.

HOCKSTEIN, N. G. et al. A history of robots: from science fiction to surgical robotics. **Journal of Robotic Surgery**, Springer Science and Business Media LLC, v. 1, n. 2, p. 113–118, mar. 2007. ISSN 1863-2491. Disponível em: http://dx.doi.org/10.1007/s11701-007-0021-2.

JAZAYERI, M. Some trends in web application development. **IEEE: Future of Software Engineering**, p. 199–213, 2007.

MACENSKI, S. et al. Robot operating system 2: Design, architecture, and uses in the wild. **Science Robotics**, American Association for the Advancement of Science (AAAS), v. 7, n. 66, maio 2022. ISSN 2470-9476. Disponível em: http://dx.doi.org/10.1126/scirobotics.abm6074.

MU, L. Multi-purpose web framework design based on websocket over http gateway. 2016. 09 2016. 17.

MURLEY, P. et al. Websocket adoption and the landscape of the real-time web. In: **Proceedings of the Web Conference 2021**. New York, NY, USA: Association for Computing Machinery, 2021. (WWW '21), p. 1192–1203. ISBN 9781450383127. Disponível em: https://doi.org/10.1145/3442381.3450063.

PIMENTEL, V.; NICKERSON, B. G. Communicating and displaying real-time data with websocket. **IEEE Internet Computing**, v. 16, n. 4, p. 45–53, 2012.

PIMENTEL, V.; NICKERSON, B. G. Communicating and displaying real-time data with websocket. p. 45–53, 2012. 17.

QUIGLEY, M. et al. Ros: an open-source robot operating system. In: KOBE. **ICRA workshop on open source software**. [S.l.], 2009. v. 3, n. 3.2, p. 5.

QUIGLEY, M.; GERKEY, B.; SMART, W. D. **Programming Robots with ROS: A Practical Introduction to the Robot Operating System**. 1st. ed. Sebastopol, CA: O'Reilly Media, 2015.

RAD, B. B.; BHATTI, H. J.; AHMADI, M. An introduction to docker and analysis of its performance. **International Journal of Computer Science and Network Security (IJCSNS)**, International Journal of Computer Science and Network Security, v. 17, n. 3, p. 228, 2017.

RAJAPAKSHA, D. D. et al. Web based user-friendly graphical interface to control robots with ros environment. In: **2021 6th International Conference on Information Technology Research (ICITR)**. [S.l.: s.n.], 2021. p. 1–6.

ROBOT Operating System (ROS). Springer International Publishing, 2016. ISSN 1860-9503. ISBN 9783319260549. Disponível em: http://dx.doi.org/10.1007/978-3-319-26054-9.

SHKLAR, L.; ROSEN, R. Web application architecture. 2003. 14, 15.

SHKLAR, L.; ROSEN, R. **Web Application Architecture: Principles, Protocols, and Patterns**. Segunda. [S.l.]: Wiley, 2009.

SHUKLA, M.; SHUKLA, A. N. Growth of robotics industry early in 21st century. **International Journal Of Computational Engineering Research**, v. 2, n. 5, p. 1554–1558, 2012.

SINHA, A. Client-server computing. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 35, n. 7, p. 77–98, jul. 1992. ISSN 0001-0782. Disponível em: https://doi.org/10.1145/129902.129908.

THAKUR, P.; JADON, P. Django: Developing web using python. In: **2023 3rd International Conference on Advance Computing and Innovative Technologies in Engineering (ICACITE)**. [S.l.: s.n.], 2023. p. 303–306.

TORVALDS, L. **Intelligence is the ability to avoid doing work, yet getting the work done**. 2013. Twitter/X post. Disponível em: https://x.com/Linus__Torvalds/status/296333706828849153. Acesso em: 14 set. 2025.

VELAMALA, S. S.; PATIL, D.; MING, X. Development of ros-based gui for control of an autonomous surface vehicle. In: **2017 IEEE International Conference on Robotics and Biomimetics (ROBIO)**. [S.l.: s.n.], 2017. p. 628–633.

WANG, V.; SALIM, F.; MOSKOVITS, P. The websocket protocol. In: ____. **The Definitive Guide to HTML5 WebSocket**. Apress, 2013. p. 33–60. ISBN 9781430247418. Disponível em: http://dx.doi.org/10.1007/978-1-4302-4741-8_3.