Relatório Técnico: Robô Autônomo de Entrega com ROS 2

Equipe:
Glenda Santana
Kalvin Albuquerque

22 de setembro de 2025

Sumário

1 Introdução

Este relatório detalha a arquitetura de software e a lógica de controle de um robô autônomo projetado para realizar tarefas de entrega em um ambiente de armazém simulado. O sistema, desenvolvido utilizando o framework ROS 2 (Robot Operating System), é capaz de navegar, identificar visualmente, coletar e entregar um objeto (encomenda) em um local pré-definido, enquanto desvia de obstáculos.

O objetivo principal do projeto foi desenvolver uma solução modular e robusta, separando as responsabilidades de percepção visual e controle de movimento em nós independentes, o que facilita a depuração, manutenção e escalabilidade do sistema.

2 Estrutura do Projeto

O projeto foi desenvolvido em um workspace do ROS 2, utilizando Python como linguagem de programação principal. A estrutura modular visa a clareza e a reutilização de código.

2.1 Dependências e Bibliotecas

O sistema depende de bibliotecas padrão do ROS 2 e de pacotes de processamento de imagem amplamente utilizados:

- **rclpy:** Biblioteca de cliente ROS 2 para Python, fundamental para a criação de nós, publishers e subscribers.
- **OpenCV (cv2):** Biblioteca de visão computacional de código aberto, utilizada intensivamente no `vision_node` para manipulação de imagens, conversão de espaço de cores (BGR para HSV) e detecção de contornos.
- **NumPy:** Utilizada para operações numéricas eficientes, especialmente na criação e manipulação das máscaras de cores.
- **cv_bridge:** Uma ferramenta ROS essencial para converter mensagens do tipo `sensor_msgs/Image` para o formato de imagem do OpenCV e vice-versa.
- **math, time, enum:** Bibliotecas padrão do Python utilizadas para cálculos trigonométricos no retorno à base, pausas temporizadas e para a definição da máquina de estados, respectivamente.

2.2 Arquivo de Lançamento (Launch File)

Um arquivo de lançamento do ROS 2 orquestra a inicialização de todos os componentes necessários para a simulação e operação do robô. Suas principais responsabilidades são:

1. Iniciar o simulador Gazebo, carregando o servidor (`gzserver`) e o cliente gráfico (`gzclient`).
2. Carregar o mundo virtual do armazém definido em um arquivo `.world`.
3. "Spnar"(instanciar) o modelo do robô TurtleBot3 no ambiente de simulação.
4. Iniciar o nó `robot_state_publisher` para publicar as transformações (TF) da estrutura do robô.
5. Lançar os nós customizados do projeto: `vision_node`, `controller_node` e `statistics_collector_n`

3 Arquitetura ROS

A arquitetura do sistema foi projetada de forma modular, dividida em nós que se comunicam através de tópicos ROS. Essa abordagem desacopla a lógica de processamento de imagem, controle e coleta de métricas.

3.1 Nós do Sistema

- **Vision Node (`vision_node`):** Este nó é inteiramente dedicado ao processamento das imagens capturadas pela câmera do robô. Sua principal função é detectar a encomenda (um objeto vermelho) e obstáculos (objetos azuis). As informações extraídas são publicadas em tópicos específicos.
- **Controller Node (`controller_node`):** Este nó implementa a inteligência do robô. Ele contém a máquina de estados que dita o comportamento do sistema e se inscreve nos tópicos publicados pelo `vision_node` e nos sensores do robô para tomar decisões de movimentação.
- **Statistics Collector Node (`statistics_collector`):** Nó de propósito específico para a automação de testes. Ele monitora o sistema para determinar o resultado de uma missão (sucesso ou falha) e mede sua duração.

3.2 Diagrama de Comunicação

O diagrama a seguir (Figura 1) ilustra a interação entre os nós principais e os tópicos do sistema. O nó `statistics_collector` não está representado visualmente para manter a clareza, mas ele se inscreve no tópico global de logs `/rosout` para monitorar as mensagens de todos os outros nós.

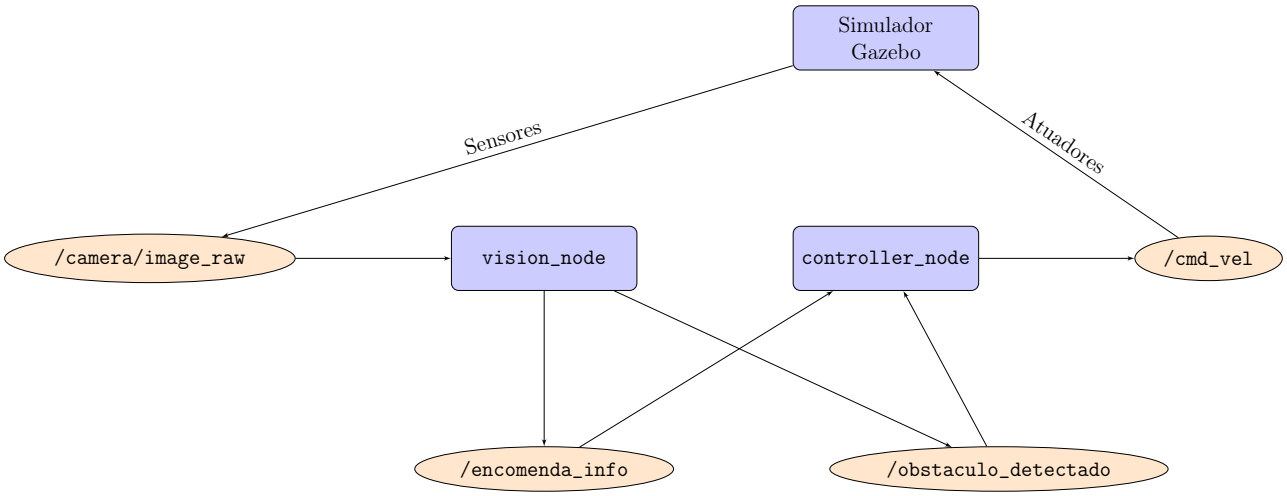

Figura 1: Arquitetura de Nós e Tópicos do ROS.

4 Máquina de Estados

O comportamento do robô é gerenciado por uma máquina de estados finitos (FSM) implementada no `controller_node`. A FSM garante que o robô execute suas tarefas de forma ordenada e reaja adequadamente aos eventos do ambiente. Cada estado possui uma lógica de ação e condições de transição bem definidas.

4.1 Descrição Detalhada dos Estados

- **BUSCANDO_ENCOMENDA:** É o estado inicial. O robô gira sobre seu eixo para varrer o ambiente.
- **APROXIMANDO_ENCOMENDA:** Após detectar a encomenda, o robô usa um controlador proporcional para se alinhar e avançar em direção a ela.
- **COLETANDO_ENCOMENDA:** Simula a ação de pegar o objeto, parando por um instante.
- **RETORNANDO_PARA_BASE:** O robô navega de volta ao ponto de origem (0,0) usando odometria e uma lógica de controle Proporcional.
- **OBSTACULO_DETECTADO:** Estado de segurança para desviar de obstáculos visuais, escolhendo o lado com mais espaço livre.
- **ENTREGANDO_ENCOMENDA:** O estado final da missão, onde o robô para e encerra suas operações.

5 Testes e Resultados

Para validar a robustez e eficiência do sistema, foi criado um framework de testes automatizados, composto por um nó coletor de estatísticas e um script orquestrador.

5.1 Automação com o Nó `statistics_collector`

Para viabilizar a execução de múltiplos testes de forma automática e a coleta de métricas precisas, foi desenvolvido o nó `statistics_collector`.

- **Finalidade:** O nó serve como um "juiz"automático para cada execução. Sua única função é observar a simulação, medir o tempo de execução e reportar o resultado final.
- **Funcionamento:**
  1. Ao ser iniciado, o nó registra o tempo atual (início da missão).
  2. Ele se inscreve no tópico `/rosout`, que centraliza todas as mensagens de log publicadas pelos outros nós do sistema.
  3. Ele monitora o fluxo de logs em busca da mensagem específica "Encomenda entregue! Missão cumprida.", que é publicada pelo `controller_node` no estado final.
  4. Ao detectar a mensagem de sucesso, o nó calcula a duração total da missão, imprime o resultado (SUCCESS) e a duração no terminal em um formato padronizado, e então se encerra.
  5. O nó também recebe um parâmetro de *timeout*. Se a mensagem de sucesso não for detectada dentro desse tempo limite, ele considera a missão como falha, imprime o resultado (FAILURE - TIMEOUT) e encerra a execução.
- **Uso:** Este nó não é utilizado na operação normal do robô, sendo ativado apenas durante a execução do script de testes automatizados, que o utiliza para capturar os resultados de cada cenário.

5.2 Cenários de Teste

1. **Sem Obstáculos, Encomenda na Frente:** O cenário mais simples, onde a encomenda é posicionada diretamente à frente do robô.

2. **Sem Obstáculos, Encomenda Atrás:** Um teste para a capacidade de busca do robô, onde a encomenda é posicionada atrás dele.

3. **Com Obstáculo na Frente:** O cenário mais complexo, com um obstáculo posicionado diretamente entre o robô e a encomenda.

5.3 Resultados

Todos os 20 testes em cada um dos três cenários foram concluídos com sucesso. A tabela abaixo resume o tempo médio de conclusão da missão para cada cenário.

Cenário de Teste	**Resultado Médio**	**Duração Média (s)**
`sem_obstaculos_encomenda_na_frente.world`	SUCCESS	17.71
`sem_obstaculos_encomenda_atras.world`	SUCCESS	28.61
`com_obstaculo_na_frente.world`	SUCCESS	56.49

Tabela 1: Resultados dos testes automatizados (média de 20 execuções).

Os resultados demonstram que a máquina de estados e os algoritmos de controle são eficazes. O aumento no tempo de duração é consistente com a complexidade de cada cenário: a busca inicial (encomenda atrás) adiciona cerca de 11 segundos, enquanto a necessidade de contornar um obstáculo adiciona aproximadamente 40 segundos à missão mais simples.