# 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 dois nós customizados do projeto: `vision_node.py` e `controller_node.py`.

# 3 Arquitetura ROS

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

## 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, como a posição da encomenda e a presença de obstáculos, 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. O `controller_node` se inscreve nos tópicos publicados pelo `vision_node` e nos sensores do robô (odometria e laser) para tomar decisões sobre a movimentação, como girar para procurar a encomenda, aproximar-se para coleta ou navegar de volta à base.

## 3.2 Diagrama de Comunicação

O diagrama a seguir (Figura 1) ilustra a interação entre os nós e os tópicos do sistema.

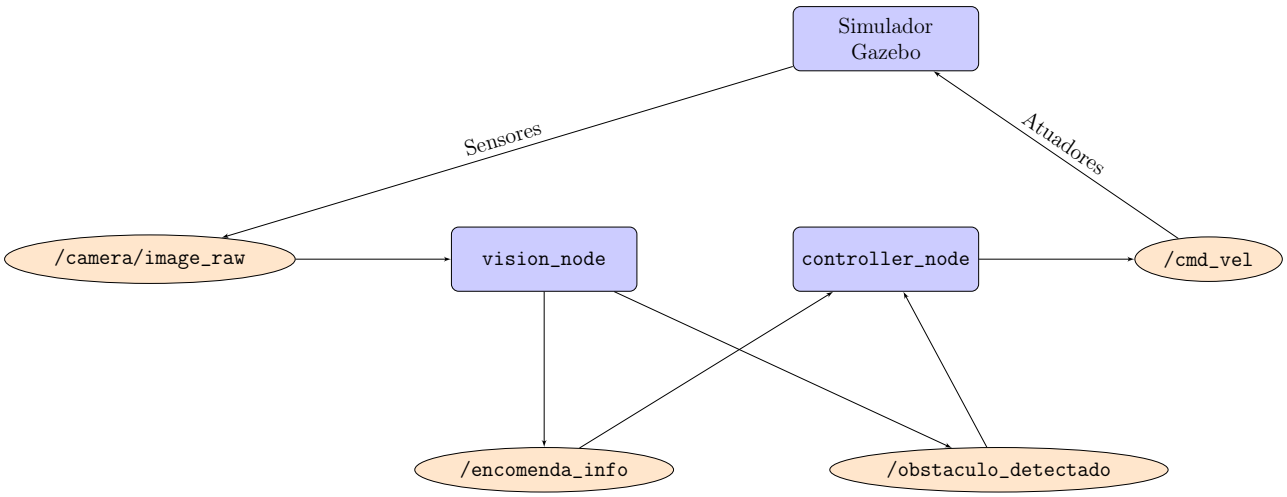


Figura 1: Arquitetura de Nós e Tópicos do ROS.

# 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.

- **Ação:** O robô executa um movimento de rotação constante sobre o próprio eixo (`angular.z = 0.3`) para varrer o ambiente visualmente.
- **Transição para APROXIMANDO_ENCOMENDA:** Ocorre assim que o `vision_node` detecta a encomenda e a flag `encomenda_detectada` se torna `True`.
- **Transição para OBSTACULO_DETECTADO:** Ocorre se um obstáculo visual for detectado e a leitura do sensor a laser frontal indicar um objeto a menos de 0.4 metros.

- **APROXIMANDO_ENCOMENDA:** Neste estado, o robô utiliza um controlador proporcional para se alinhar e avançar em direção à encomenda.
  - **Ação:**
    1. Calcula o erro de alinhamento subtraindo a posição X do centro da encomenda da posição central da câmera (320 pixels).
    2. A velocidade angular (`angular.z`) é definida proporcionalmente a este erro (`-0.002 * erro`), fazendo o robô girar para centralizar a encomenda em seu campo de visão.
    3. A velocidade linear (`linear.x`) é ajustada dinamicamente: se o obstáculo mais próximo estiver a 0.8 metros ou mais, ele avança a 0.3 m/s; caso contrário, desacelera para 0.1 m/s para uma aproximação segura.
  - **Transição para COLETANDO_ENCOMENDA:** Ocorre quando o sensor a laser frontal detecta que o robô está a menos de 0.5 metros da encomenda.
  - **Transição para BUSCANDO_ENCOMENDA:** Se, por algum motivo, o robô perder a encomenda de vista (`encomenda_detectada` se torna `False`).

- **COLETANDO_ENCOMENDA:** Simula a ação de pegar o objeto.
  - **Ação:** O robô para completamente (`Twist()`), define a flag `encomenda_coletada` como `True` e aguarda por 3 segundos.
  - **Transição para RETORNANDO_PARA_BASE:** Transição automática após a pausa de 3 segundos.

- **RETORNANDO_PARA_BASE:** O robô navega de volta ao ponto de origem (0,0) usando odometria e uma lógica de controle aprimorada.
  - **Ação:** Implementa uma lógica de controle Proporcional que permite ao robô girar e avançar simultaneamente, resultando em uma trajetória mais suave e eficiente.
    1. O ângulo para a base é calculado continuamente usando `atan2(-pos_y, -pos_x)`.
    2. O erro angular (diferença entre a orientação atual e o ângulo alvo) é normalizado para garantir sempre o menor caminho de rotação.
    3. A velocidade angular (`angular.z`) é proporcional ao erro, garantindo que o robô gire mais rápido quando está mais desalinhado.
    4. A velocidade linear (`linear.x`) é inversamente proporcional ao erro angular. O robô atinge a velocidade máxima (0.7 m/s) quando perfeitamente alinhado e desacelera à medida que o erro aumenta.
    5. Como medida de segurança, se o erro angular for muito grande (maior que 60 graus), a velocidade linear é zerada, e o robô apenas gira para se realinhar antes de prosseguir.

- **Transição para ENTREGANDO_ENCOMENDA:** Ocorre quando a distância até a base (origem) é menor que 0.2 metros.

- **OBSTACULO_DETECTADO:** Estado de segurança para desvio de obstáculos.
  - **Ação:** Aciona a função `contornar_obstaculo`, que implementa uma estratégia mais robusta:
    1. Inicialmente, o robô avalia o espaço livre à sua esquerda e à sua direita (usando as médias das leituras do laser) e escolhe o lado com mais espaço para iniciar o desvio.
    2. O robô avança com uma velocidade linear de 0.25 m/s e uma velocidade angular de 0.5 rad/s (ou -0.5) na direção escolhida.
    3. Uma lógica de segurança foi adicionada: se o robô ficar muito próximo de um obstáculo (menos de 0.2m) ou se passar muito tempo tentando desviar (mais de 50 ciclos), ele para, recua por aproximadamente 1 metro, e reavalia qual é o melhor lado para o desvio, reiniciando o processo.
  - **Transição de volta:** Quando o `vision_node` para de detectar o obstáculo azul, o robô retorna ao seu estado anterior (`BUSCANDO_ENCOMENDA` ou `RETORNANDO_PARA_BASE`).

- **ENTREGANDO_ENCOMENDA:** O estado final da missão.
  - **Ação:** O robô para completamente e o timer que executa a máquina de estados é cancelado, encerrando o programa.
  - **Transição:** Nenhuma. É um estado terminal.

# 5 Testes e Resultados

Para validar a robustez e eficiência do sistema, foram conduzidos testes automatizados em três cenários distintos, simulando diferentes condições de operação no ambiente do armazém. Cada cenário foi executado 20 vezes para obter uma média de desempenho consistente.

## 5.1 Cenários de Teste

1. **Sem Obstáculos, Encomenda na Frente:** O cenário mais simples, onde a encomenda é posicionada diretamente à frente do robô, sem nenhum obstáculo no caminho.

2. **Sem Obstáculos, Encomenda Atrás:** Um teste para a capacidade de busca do robô, onde a encomenda é posicionada atrás de seu ponto de partida, forçando-o a realizar uma rotação completa para encontrá-la.

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

## 5.2 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.