INSTITUTO FEDERAL DE SANTA CATARINA

LUCAS DA SILVA

Controle e Localização de Robô Móvel via Aplicativo Android: Uso de OpenCV e KNN

São José - SC

fevereiro/2026

CONTROLE E LOCALIZAÇÃO DE ROBÔ MÓVEL VIA APLICATIVO ANDROID: USO DE OPENCV E KNN

Monografia apresentada ao Curso de Engenharia de Telecomunicações do campus São José do Instituto Federal de Santa Catarina para a obtenção do diploma de Engenheiro de Telecomunicações.

Orientador: Prof. Eraldo Silveira e Silva, Dr.

São José - SC

fevereiro/2026

Lucas da Silva

Controle e Localização de Robô Móvel via Aplicativo Android: Uso de OpenCV e KNN

Este trabalho foi julgado adequado para obtenção do título de Engenheiro de Telecomunicações, pelo Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina, e aprovado na sua forma final pela comissão avaliadora abaixo indicada.

São José - SC, 27 de fevereiro de 2026:

gov.br
Documento assinado digitalmente
ERALDO SILVEIRA E SILVA
Data: 06/03/2026 14:58:14-0300
Verifique em https://validar.iti.gov.br

Prof. Eraldo Silveira e Silva, Dr.
Orientador
Instituto Federal de Santa Catarina

gov.br
Documento assinado digitalmente
JORGE HENRIQUE BUSATTO CASAGRANDE
Data: 06/03/2026 17:45:35-0300
Verifique em https://validar.iti.gov.br

Professor Jorge Henrique Busatto Casagrande, Dr.
Instituto Federal de Santa Catarina

gov.br
Documento assinado digitalmente
SAUL SILVA CAETANO
Data: 07/03/2026 09:08:19-0300
Verifique em https://validar.iti.gov.br

Professor Saul Silva Caetano, Dr.
Instituto Federal de Santa Catarina

AGRADECIMENTOS

Sou grato aos meus pais, que desde cedo me ensinaram que, para vencer na vida, são necessários estudo e trabalho duro.

À minha esposa, pelo suporte nos dias bons e, principalmente, nos ruins, e pela compreensão por tudo o que abrimos mão para chegarmos até aqui.

Aos meus irmãos, sobrinhos e à minha cunhada, pelo apoio essencial ao longo de todos estes anos.

Ao meu orientador, professor Eraldo, que, além de orientar, me incentivou e me motivou a concluir este trabalho, mesmo quando eu pensava em desistir. Obrigado por sempre me tratar da maneira mais humanizada possível.

A todos os professores e demais servidores do IFSC São José, pelas orientações e pela contribuição ao longo da minha trajetória acadêmica.

Aos amigos, colegas de faculdade e de trabalho, que tornaram a jornada um pouco mais leve até aqui.

Não importa o quanto você bate, mas sim o quanto aguenta apanhar e continuar.
O quanto pode suportar e seguir em frente. É assim que se ganha.
Rocky Balboa

RESUMO

Este trabalho apresenta o desenvolvimento de um aplicativo móvel para a plataforma Android destinado ao controle manual de um robô e ao reconhecimento de ambientes internos por meio de técnicas de visão computacional. O aplicativo permite a comunicação com o robô por comandos de rede, o controle de suas funções principais via *socket* Transmission Control Protocol (TCP) e a exibição do *streaming* de vídeo no *smartphone* via requisição Hypertext Transfer Protocol (HTTP). Também foi implementado um subsistema de reconhecimento de ambientes utilizando a biblioteca OpenCV em conjunto com os algoritmos Local Binary Pattern (LBP) e K-Nearest Neighbors (KNN). Nesse subsistema, os ambientes são previamente mapeados a partir de quatro imagens correspondentes às direções norte, leste, sul e oeste, sendo posteriormente comparados com novas imagens para estimar o ambiente atual. Na versão validada neste trabalho, as imagens utilizadas no reconhecimento foram obtidas a partir da galeria do *smartphone*, embora a implementação tenha sido organizada para permitir, futuramente, a substituição dessa fonte por *frames* do vídeo do robô. Os resultados indicaram que a solução foi funcional no escopo avaliado, demonstrando potencial para aplicação de uma abordagem de baixo custo no reconhecimento de ambientes internos.

Palavras-chave: robótica móvel. visão computacional. OpenCV. LBP. KNN. Android.

ABSTRACT

This work presents the development of a mobile application for the Android platform designed for the manual control of a robot and the recognition of indoor environments using computer vision techniques. The application allows communication with the robot via network commands, control of its main functions via Transmission Control Protocol (TCP) socket, and display of streaming video on the smartphone via Hypertext Transfer Protocol (HTTP) request. An environment recognition subsystem was also implemented using the OpenCV library together with the Local Binary Pattern (LBP) and K-Nearest Neighbors (KNN) algorithms. In this subsystem, environments are previously mapped from four images corresponding to the north, east, south, and west directions, and subsequently compared with new images to estimate the current environment. In the version validated in this work, the images used for recognition were obtained from the smartphone's gallery, although the implementation was organized to allow, in the future, the replacement of this source with frames from the robot's video. The results indicated that the solution was functional within the evaluated scope, demonstrating potential for applying a low-cost approach to indoor environment recognition.

Keywords: mobile robotics. computer vision. OpenCV. LBP. KNN. Android.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AAR Android Archive.

AOT Ahead-Of-Time.

API Application Programming Interface.

ART Android Runtime.

BLE Bluetooth Low Energy.

FPS Frames Per Second.

GPS Global Positioning System.

GPU Graphics Processing Unit.

HAL Hardware Abstraction Layer.

HTTP Hypertext Transfer Protocol.

I/O Input/Output.

IA Inteligência Artificial.

IFC Industry Foundation Classes.

IFSC Instituto Federal de Santa Catarina.

IMU Inertial Measurement Unit.

IP Internet Protocol.

JPEG Joint Photographic Experts Group.

JSON JavaScript Object Notation.

KNN K-Nearest Neighbors.

LBP Local Binary Pattern.

LiDAR Light Detection and Ranging.

LQI Link Quality Indicator.

MJPEG Motion Joint Photographic Experts Group.

MVC Model-View-Controller.

MVI Model-View-Intent.

MVP Model-View-Presenter.

MVVM Model-View-ViewModel.

NDK Native Development Kit.

ORB Oriented FAST and Rotated BRIEF.

RGB Red, Green, Blue.

RGB-D Red, Green, Blue - Depth.

SDK Software Development Kit.

SIFT Scale-Invariant Feature Transform.

SLAM Simultaneous Localization and Mapping.

SSH Secure Shell.

SURF Speeded Up Robust Features.

TCC Trabalho de Conclusão de Curso.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

URL Uniform Resource Locator.

USB Universal Serial Bus.

USB OTG Universal Serial Bus On-The-Go.

XML Extensible Markup Language.

SUMÁRIO

1 INTRODUÇÃO

A robótica móvel tem se consolidado como uma das áreas mais dinâmicas e relevantes dentro da ciência da computação e da engenharia, com aplicações que vão desde a indústria até serviços de assistência e monitoramento. Um dos maiores desafios dessa área é a capacidade de perceber o ambiente e localizar-se adequadamente, etapa fundamental para qualquer forma de navegação autônoma. Nesse contexto, a visão computacional surge como um dos instrumentos mais promissores, pois permite extrair informações ricas a partir de câmeras, que são sensores de baixo custo e amplamente disponíveis. Em particular, o avanço de bibliotecas de código aberto, como a OpenCV, aliado a técnicas de aprendizado supervisionado, como o algoritmo K-Nearest Neighbors (KNN), ampliou as possibilidades de aplicação prática da visão computacional em sistemas embarcados.

No contexto da robótica móvel, diversos métodos têm sido estudados para resolver o problema da localização e navegação em ambientes internos. Embora existam soluções robustas baseadas em sensores dedicados, como Light Detection and Ranging (LiDAR) e sistemas de múltiplas câmeras, essas abordagens, muitas vezes, apresentam custo elevado e complexidade de integração. Nesse cenário, surge a oportunidade de explorar alternativas mais acessíveis, que utilizem câmeras comuns acopladas ao robô e realizem o processamento em dispositivos móveis. O *smartphone*, além de ser uma plataforma amplamente disponível, oferece poder computacional suficiente para executar algoritmos de visão computacional e aprendizado supervisionado, como OpenCV e KNN, tornando-se uma opção viável para pesquisas acadêmicas de baixo custo.

O problema de pesquisa que orienta este trabalho consiste em investigar a viabilidade e a eficácia do uso de técnicas de visão computacional e aprendizado supervisionado para localização de robôs móveis em ambientes internos, quando o processamento é realizado em um aplicativo Android. Mais especificamente, busca-se responder à seguinte questão: até que ponto o uso do KNN, aliado a descritores visuais extraídos via OpenCV, permite identificar corretamente a posição ou o ambiente em que o robô se encontra? Essa formulação não trata da possibilidade de implementação em si, mas da análise da precisão, robustez e limitações da abordagem adotada, o que contribui para avaliar seu potencial de uso em sistemas reais de navegação autônoma.

A relevância deste trabalho está tanto no campo acadêmico quanto no prático. Do ponto de vista acadêmico, o estudo se justifica por integrar conceitos de aprendizado supervisionado e visão computacional aplicados em dispositivos móveis, temas cada vez mais explorados na área de robótica móvel. Na prática, o projeto busca desenvolver uma solução acessível para auxiliar na localização de robôs em ambientes internos. Dessa

forma, o trabalho demonstra a importância de investigar alternativas simples e de baixo custo para problemas reais da robótica.

Assim, esta pesquisa procura unir conceitos de visão computacional e aprendizado de máquina à prática de desenvolvimento em Android, a fim de investigar soluções acessíveis e eficazes para a localização de robôs móveis em ambientes internos.

1.1 Objetivo geral

Desenvolver um aplicativo móvel para a plataforma Android, destinado ao controle manual de um robô e à obtenção de sua localização por meio de técnicas de processamento de imagens.

1.2 Objetivos específicos

- Desenvolver um aplicativo para dispositivos Android que possibilite o controle manual do robô, incluindo a transmissão e exibição da imagem da câmera em tempo real.
- Implementar um subsistema de localização baseado em processamento de imagens, utilizando a biblioteca OpenCV em conjunto com o algoritmo KNN, com o objetivo de identificar o ambiente no qual o robô está inserido.
- Realizar testes experimentais para avaliar a acurácia e a robustez do subsistema de localização proposto.
- Documentar o processo de desenvolvimento, os métodos empregados e as decisões de projeto, visando aprimoramentos e expansões futuras do sistema.

1.3 Organização do texto

O texto está organizado da seguinte forma: No Capítulo 2 é apresentada a fundamentação teórica, conhecimentos adquiridos ao longo do processo de pesquisa para a realização do trabalho. No Capítulo 3 são apresentadas as especificações e a implementação do aplicativo. O Capítulo 4 mostra quais validações foram realizadas para cada operação do aplicativo e os resultados obtidos. Por fim, o Capítulo 5 discute os resultados, os objetivos alcançados, as oportunidades de melhoria e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, será apresentada uma visão geral sobre os principais conceitos envolvidos na movimentação autônoma de robôs móveis, destacando os pontos relacionados à localização, que é um dos temas centrais deste trabalho. São abordados os fundamentos da robótica móvel, do algoritmo KNN, do uso da biblioteca OpenCV e da arquitetura de aplicações Android, que juntos formam a base teórica necessária para o desenvolvimento da proposta.

2.1 Movimentação, Navegação e Localização em Robôs Móveis

A robótica móvel é uma área da robótica que se concentra no desenvolvimento de sistemas capazes de se locomover de maneira autônoma em ambientes diversos, utilizando sensores, algoritmos de percepção, localização, mapeamento e planejamento para tomar decisões e executar tarefas. Esses sistemas não apenas se movem, mas também interagem com o ambiente, tomam decisões e adaptam seu comportamento a partir das informações percebidas.

De acordo com Niloy et al. (2021), "Robôs que podem se mover de forma autônoma e tomar decisões inteligentes, percebendo seus ambientes e objetos ao redor, são conhecidos como robôs móveis autônomos". Este tipo de robô tem aplicações crescentes em indústrias, residências, hospitais, ambientes comerciais e até em missões de exploração.

O desenvolvimento de robôs móveis autônomos envolve uma série de subsistemas que trabalham de forma integrada. Segundo Niloy et al. (2021), os componentes essenciais de um sistema de robótica móvel incluem: "locomoção, percepção, cognição, navegação, localização, mapeamento, planejamento de trajetória e desvio de obstáculos". Cada um desses elementos desempenha um papel fundamental no funcionamento do robô e na sua capacidade de se deslocar de maneira eficaz no ambiente.

2.1.1 Locomoção

A locomoção refere-se à capacidade física do robô de se mover no ambiente. Essa movimentação pode ocorrer de diferentes formas, dependendo do tipo de terreno, dos requisitos da aplicação e das restrições de projeto. De acordo com Niloy et al. (2021), "Mecanismos de locomoção são necessários em robôs móveis para que eles se movam pelos ambientes onde desempenham suas tarefas, o que pode ser alcançado por meio de uma compreensão adequada da cinemática, dinâmica e do mecanismo do sistema".

Os principais tipos de locomoção incluem:

- Locomoção com rodas: é o método mais comum devido à simplicidade, baixo custo, eficiência energética e facilidade de controle. Existem variações como rodas padrão, rodas *caster*, rodas omnidirecionais (como as suecas ou *mecanum*) e rodas esféricas (NILOY et al., 2021).
- Locomoção com pernas: inspirada na natureza, oferece maior mobilidade em terrenos irregulares, sendo utilizada em robôs bípedes, quadrúpedes, hexápodes, entre outros. Contudo, apresenta maior complexidade de controle e consumo energético (NILOY et al., 2021).
- Locomoção por esteiras: oferece estabilidade e capacidade de atravessar terrenos macios ou irregulares, sendo comuns em robôs de resgate e exploração (NILOY et al., 2021).
- Outros métodos não convencionais: incluem locomoção serpentina, combinando rodas e pernas (*walking wheels*), ou ainda sistemas inspirados no movimento de insetos e animais (NILOY et al., 2021).

A escolha do sistema de locomoção está diretamente relacionada à tarefa que o robô irá desempenhar e às características do ambiente.

2.1.2 Percepção

A percepção é responsável por permitir que o robô obtenha informações do ambiente e compreenda sua própria posição em relação a ele. De acordo com Niloy et al. (2021), "Percepção se refere à capacidade de ver, ouvir ou tornar-se consciente do ambiente ao redor". Esse processo envolve sensores, algoritmos de processamento de dados e modelos de representação do ambiente.

Os sensores podem ser classificados em:

- Proprioceptivos: medem o estado interno do robô, como velocidade, aceleração, inclinação e orientação.
- Exteroceptivos: capturam informações do ambiente externo, como distância de objetos, presença de obstáculos, textura de superfícies, entre outros.

Dentre os sensores mais comuns estão:

- Sensores ultrassônicos e infravermelhos: utilizados para detecção de obstáculos e medição de distância (ILIAS et al., 2016).

- Câmeras Red, Green, Blue (RGB) e Red, Green, Blue - Depth (RGB-D): permitem a obtenção de informações visuais e de profundidade, essenciais para reconhecimento de objetos e mapeamento tridimensional (LIU, 2022).
- Light Detection and Ranging (LiDAR): fornece medições precisas de distância em 2D ou 3D, extremamente útil para Simultaneous Localization and Mapping (SLAM) e navegação *indoor* (NILOY et al., 2021).
- Inertial Measurement Unit (IMU): composta por acelerômetros, giroscópios e magnetômetros, fornece dados sobre movimento e orientação (NILOY et al., 2021).

A percepção não ocorre de forma isolada. Segundo Niloy et al. (2021), "a integração de múltiplos sensores é quase sempre útil para fundir ou combinar diversas fontes de informações sensoriais em um único formato representacional". Essa fusão de dados permite aumentar a confiabilidade e a precisão das informações utilizadas pelos demais sistemas do robô.

2.1.3 Cognição

A cognição refere-se à capacidade do robô de interpretar os dados sensoriais, tomar decisões e gerar comandos de alto nível. De acordo com Niloy et al. (2021), "Após a percepção, a cognição desempenha um papel fundamental ao permitir que o robô processe informações, compreenda situações e planeje ações de forma adequada".

Esse processo inclui:

- Interpretação do ambiente: entender objetos, espaços, obstáculos e mudanças no cenário.
- Tomada de decisão: selecionar ações com base nos objetivos definidos, condições do ambiente e regras programadas ou aprendidas.
- Aprendizado: alguns sistemas incorporam aprendizado de máquina para melhorar seu desempenho ao longo do tempo.

A cognição é, portanto, a ponte entre percepção e navegação, permitindo que o robô não apenas reaja, mas também antecipe eventos e adapte seu comportamento.

2.1.4 Localização e Mapeamento

A localização e o mapeamento são processos fundamentais para que o robô entenda onde está e como é o ambiente ao seu redor. Segundo Niloy et al. (2021), "O fator-chave da navegação é a localização, que ajuda a otimizar o ponto de operação do robô dentro do ambiente".

- Mapeamento: consiste na criação de uma representação do ambiente, que pode ser métrica (mapas baseados em coordenadas), topológica (representações de conexões entre espaços) ou híbrida. De acordo com Li et al. (2019), existem diversos padrões de mapas *indoor*, como IndoorGML, CityGML, Industry Foundation Classes (IFC) e Indoor OpenStreetMap, cada um com características específicas.
- Localização: é a capacidade de determinar a posição do robô no mapa criado. Em ambientes *indoor*, onde não há sinal de Global Positioning System (GPS), são utilizadas técnicas como SLAM, algoritmos de correspondência de características visuais e métodos baseados em sensores ultrassônicos ou de distância.

Liu (2022) destaca que, para ambientes dinâmicos, é necessário utilizar algoritmos que consigam distinguir objetos estáticos e dinâmicos, evitando que elementos móveis prejudiquem a precisão do mapeamento e da localização.

Em termos práticos, a localização responde diretamente à pergunta "Onde estou?", que é a base sobre a qual todo o restante da navegação se apoia. Sem saber sua posição no ambiente, o robô não pode planejar caminhos, evitar obstáculos de maneira eficiente ou sequer garantir que está se movendo na direção correta.

Existem diversas abordagens para localização, dependendo do tipo de sensores e da representação do ambiente:

- Localização baseada em mapas e sensores: utiliza correspondência entre dados sensoriais (como LiDAR, câmeras ou ultrassom) e mapas previamente construídos.
- Simultaneous Localization and Mapping (SLAM): quando o robô não possui um mapa prévio, ele realiza simultaneamente o mapeamento do ambiente e a sua própria localização.
- Técnicas baseadas em visão computacional: utilizam imagens para extrair características do ambiente e realizar a correspondência com mapas ou bases de dados. É o caso do uso de *Optical Flow*, Oriented FAST and Rotated BRIEF (ORB)-SLAM ou métodos baseados em *deep learning* (LIU, 2022).
- Métodos de classificação, como KNN: são aplicáveis em problemas de localização baseada na identificação de padrões em dados sensoriais. Segundo Ilias et al. (2016), "O KNN foi selecionado como algoritmo de localização devido às suas vantagens e à sua capacidade de classificar grandes volumes de dados não estruturados".

2.1.5 Navegação, Planejamento de Caminho e Desvio de Obstáculos

A navegação é o processo que permite ao robô mover-se de um ponto a outro no ambiente, tomando decisões sobre qual caminho seguir e como evitar colisões. Segundo

Niloy et al. (2021), “A seleção da melhor rota para mover o robô móvel dentro do ambiente é feita com a ajuda de uma abordagem de navegação composta por algoritmos de planejamento de trajetória e desvio de obstáculos, que geram trajetórias em tempo real e evitam colisões”.

O planejamento de caminho é o processo que permite ao robô calcular uma rota adequada entre sua posição atual e um destino determinado, levando em conta o mapa e as condições do ambiente. Esse processo normalmente combina algoritmos de planejamento e técnicas de desvio de obstáculos para gerar trajetórias compatíveis com o cenário observado.

Este processo pode ser dividido em dois níveis principais:

- Planejamento global: utiliza o mapa completo do ambiente para gerar uma rota inicial, considerando apenas obstáculos fixos. Algoritmos como A* e Dijkstra são os mais tradicionais para esse fim.
- Planejamento local: adapta a rota em tempo real, lidando com obstáculos dinâmicos e mudanças não previstas no ambiente. Algoritmos como VFH+ *(Vector Field Histogram Plus*) são amplamente utilizados (LIU, 2022).

O desvio de obstáculos faz parte do planejamento local. Ele garante que o robô consiga reagir rapidamente a situações imprevistas, mantendo a segurança da operação e a continuidade da trajetória planejada.

2.1.6 Desafios e Tendências Futuras

A movimentação autônoma de robôs em ambientes internos apresenta diversos desafios, dentre eles:

- Ausência de GPS: exige o uso de sensores e algoritmos robustos de localização e mapeamento (NILOY et al., 2021).
- Fusão de dados sensoriais: lidar com informações provenientes de diferentes sensores e diferentes níveis de ruído (NILOY et al., 2021).
- Limitações computacionais: executar algoritmos complexos, como SLAM, visão computacional e aprendizado de máquina, em plataformas com *hardware* limitado.
- Ambientes dinâmicos: a presença de pessoas e objetos em movimento dificulta a manutenção de mapas precisos (LIU, 2022).
- Ambiguidade espacial: ambientes com características visuais muito semelhantes podem gerar erros de localização, especialmente em sistemas baseados em visão.

As tendências apontam para o uso crescente de *deep learning*, algoritmos mais robustos de SLAM, além da integração de inteligência artificial na cognição dos robôs, permitindo maior autonomia e adaptabilidade.

2.2 *K-Nearest Neighbors* (KNN)

2.2.1 Histórico e Conceito

O algoritmo KNN é um método de aprendizado supervisionado baseado na comparação entre amostras. Segundo Han e Kamber (2006) e Larose (2005), sua principal característica é classificar novas instâncias com base nas amostras mais próximas do conjunto de treinamento, considerando uma métrica de distância previamente definida. Essa abordagem parte do princípio de que elementos semelhantes tendem a pertencer à mesma classe.

As raízes do método remontam à década de 1960, quando seus fundamentos foram explorados no contexto de reconhecimento de padrões; consolidações posteriores o colocaram como um classificador multiclasse não paramétrico (SILVA, 2014). Em termos de base conceitual, Aha, Kibler e Albert (1991) enquadram o KNN dentro do guarda-chuva do *instance-based learning*, no qual a generalização ocorre "tardiamente" (*lazy learning*): não há uma fase de treinamento que produza um modelo explícito; o "modelo" é o próprio conjunto de exemplos.

Essa característica distingue o KNN de métodos paramétricos — como regressões lineares/logísticas ou redes neurais — nos quais se parte de um modelo a priori, isto é, uma forma funcional previamente definida que se acredita representar o comportamento do fenômeno observado. O processo de aprendizado, nesse caso, consiste em ajustar os parâmetros desse modelo teórico de modo a aproximá-lo da realidade empírica. No KNN, por outro lado, não há ajuste de parâmetros globais nem suposição de forma funcional: a decisão é local, baseada na vizinhança do ponto a ser classificado.

O KNN tem sido amplamente utilizado em diversas áreas da ciência da computação, como reconhecimento de padrões, classificação de imagens, detecção de anomalias e sistemas de recomendação. Em visão computacional, destaca-se pela capacidade de associar descritores visuais a categorias conhecidas, permitindo o reconhecimento de objetos, ambientes ou situações observadas por sensores visuais.

2.2.2 Funcionamento e Aspectos Práticos

O algoritmo KNN funciona com base na comparação de novos dados com exemplos já conhecidos. O processo pode ser descrito em etapas:

1. Recebe-se uma nova instância (um ponto cujas características são conhecidas, mas cuja classe ainda não foi determinada);
2. Calcula-se a distância entre essa instância e todas as amostras do conjunto de treinamento (por exemplo, distância euclidiana);
3. Selecionam-se os k exemplos mais próximos dessa nova instância;
4. Para classificação, observa-se qual classe é mais frequente entre esses k vizinhos e ela é atribuída ao novo ponto;
5. Para regressão, calcula-se a média (ou média ponderada pela distância) dos valores desses vizinhos para estimar o resultado.

Esse processo caracteriza o algoritmo como *lazy learner*, pois não há fase de treinamento explícito nem ajuste de parâmetros globais; o esforço computacional concentra-se na etapa de consulta (AHA; KIBLER; ALBERT, 1991).

Na prática, três componentes centrais influenciam diretamente o desempenho do KNN:

- **Valor de k**: Valores pequenos de k tornam o algoritmo altamente sensível a ruídos e outliers, enquanto valores grandes podem suavizar excessivamente as fronteiras entre classes, comprometendo a precisão. A escolha adequada é normalmente feita por validação cruzada (LAROSE, 2005).
- **Métrica de distância**: A seleção da métrica depende do tipo de dados. Para vetores contínuos normalizados, a distância Euclidiana é a mais utilizada. Em dados categóricos ou binários, distâncias como Hamming podem ser mais apropriadas (HAN; KAMBER, 2006).
- **Pré-processamento**: Normalização ou padronização dos atributos é essencial para garantir que cada característica contribua de maneira equilibrada na distância calculada. Além disso, técnicas de redução de dimensionalidade, como PCA, podem melhorar o desempenho ao reduzir redundâncias e ruídos (MIRANDA, 2011).

O **pipeline típico** em problemas de classificação com KNN pode ser estruturado da seguinte forma:

1. Coleta e preparação do conjunto de dados;
2. Extração de características relevantes (no caso de imagens, descritores como momentos de Hu, Local Binary Pattern (LBP) ou histogramas de cor);

3. Normalização ou padronização dos vetores de características;
4. Definição dos parâmetros (k, métrica de distância, regra de decisão);
5. Execução do algoritmo de classificação;
6. Avaliação do desempenho (métricas como acurácia, F_1-*score* e matriz de confusão).

Esse fluxo garante consistência na aplicação do KNN e favorece comparações mais robustas entre diferentes configurações.

2.2.3 Métricas de Distância e Normalização

Para medir a semelhança entre duas instâncias, o algoritmo KNN utiliza uma métrica de distância, que indica o quão próximos ou diferentes os pontos estão entre si. A escolha da métrica influencia diretamente o resultado da classificação ou regressão. Segundo Han e Kamber (2006), diferentes medidas podem ser aplicadas conforme o tipo de dado e o objetivo da análise. A seguir, são apresentadas as principais métricas utilizadas.

- **Distância Euclidiana:** é a métrica mais comum. Mede a distância em linha reta entre dois pontos em um espaço n-dimensional.

$$d_{\text{Euc}}(p, q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2} \tag{2.1}$$

  Onde:

  - p e q: vetores (ou instâncias) a serem comparados;
  - n: número de características (dimensões);
  - p_i e q_i: valor da i-ésima característica de cada ponto.

  A distância Euclidiana é simples e intuitiva, mas pode ser influenciada por diferenças de escala entre as variáveis.

- **Distância Manhattan:** também chamada de distância em blocos, soma as diferenças absolutas entre as coordenadas dos pontos.

$$d_{\text{Man}}(p, q) = \sum_{i=1}^{n} |p_i - q_i| \tag{2.2}$$

  Onde as variáveis têm o mesmo significado da métrica anterior. Essa métrica é menos afetada por grandes variações em um único atributo e tende a ser mais robusta a valores extremos (*outliers*).

- **Distância Minkowski:** é uma generalização das distâncias Euclidiana e Manhattan, controlada por um parâmetro r.

$$d_{\text{Min}}(p, q) = \left(\sum_{i=1}^{n} |p_i - q_i|^r \right)^{1/r} \tag{2.3}$$

  Onde:

  - $r = 1$ corresponde à distância Manhattan;
  - $r = 2$ corresponde à distância Euclidiana;
  - p_i, q_i e n mantêm o mesmo significado.

- **Distância do Cosseno:** mede o ângulo entre dois vetores, sendo útil quando o que importa é a orientação e não a magnitude dos vetores.

$$d_{\cos}(p, q) = 1 - \frac{p \cdot q}{\|p\|\|q\|} \tag{2.4}$$

  Onde o resultado varia entre -1 e 1, sendo 1 quando os vetores são idênticos e 0 quando são totalmente distintos. No KNN, geralmente utiliza-se $1 -$ similaridade para converter essa medida em distância.

- **Distância de Hamming:** aplicada a dados categóricos, conta o número de posições diferentes entre dois vetores.

$$d_{\text{Ham}}(p, q) = \frac{\text{número de posições diferentes}}{n} \tag{2.5}$$

  Onde n é o número total de atributos comparados. O valor varia de 0 (iguais) a 1 (totalmente diferentes).

Antes de aplicar qualquer métrica, é comum normalizar os dados para que todas as variáveis fiquem na mesma escala. Sem esse ajuste, atributos com valores maiores podem dominar o cálculo da distância. Segundo Miranda (2011), o pré-processamento é fundamental para uniformizar a contribuição das variáveis. A normalização mais usada é a *Min–Max*, que transforma os valores para o intervalo $[0, 1]$:

$$x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}} \tag{2.6}$$

Onde:

- x: valor original do atributo;
- $x_{\min}$ e $x_{\max}$: valores mínimo e máximo observados;
- x': valor normalizado.

2.2.4 Complexidade e Otimizações

O custo de predição do KNN "puro" é $O(Nd)$ por consulta (N amostras, d dimensões), pois requer comparar com todas as instâncias. Em bases grandes, isso é crítico em tempo real. Três frentes de otimização são comuns:

- **Indexação espacial**: estruturas como *k-d tree* e *ball tree* aceleram buscas quando d é moderado; para d alto (imagens), métodos aproximados (ANN) como FLANN (MUJA; LOWE, 2009) reduzem custo mantendo boa precisão.
- **Redução/seleção de dimensionalidade**: PCA, seleção por importância, descritores compactos (ex.: ORB) melhoram qualidade e tempo (LOWE, 2004; RUBLEE et al., 2011).
- **Aceleração em Graphics Processing Unit (GPU)**: paralelização de cálculos massivos (CORDEIRO; MEYER; ZOLA, 2023). Em dispositivos móveis, avalia-se o *trade-off* energia/latência; em servidor, GPU ou bibliotecas como FAISS (JOHNSON; DOUZE; JéGOU, 2019) são alternativas.

2.2.5 Aplicação no TCC

No presente Trabalho de Conclusão de Curso (TCC), o KNN é avaliado como técnica de classificação para responder à questão "Onde estou?". A ideia é que, a partir de imagens capturadas pela câmera do robô, sejam extraídos vetores de características por meio da biblioteca OpenCV, normalizados e comparados com uma base previamente mapeada de ambientes internos.

De forma simples, o processo pode ser entendido assim: cada imagem capturada gera um vetor de características, como $[0{,}12, 0{,}85, 0{,}44, 0{,}91]$. Esse vetor é então comparado com outros já armazenados, que representam diferentes ambientes, como "laboratório", "biblioteca" e "sala de aula". O KNN calcula a distância entre esses vetores e identifica o ambiente mais semelhante (por exemplo, o robô pode reconhecer que está no "laboratório").

Parâmetros como o valor de k serão testados de forma empírica, buscando um equilíbrio entre precisão e desempenho no dispositivo Android. Assim, não se define um conjunto fixo de parâmetros, mas sim uma abordagem exploratória para identificar a configuração mais adequada ao caso de uso.

Além disso, o sensor de bússola será utilizado para auxiliar na captura das imagens. Ele permitirá que o robô registre fotos em diferentes ângulos de rotação, garantindo uma base de dados mais completa e variada para o treinamento e a avaliação do algoritmo.

2.3 *Open Computer Vision Library* (OpenCV)

2.3.1 Histórico e Papel na Visão Computacional

A OpenCV surgiu em 2000 com a proposta de oferecer uma coleção abrangente de algoritmos de visão computacional e processamento digital de imagens de alto desempenho, de código aberto e multiplataforma (BRADSKI, 2000). Evoluiu para um ecossistema com centenas de funções otimizadas, suporte a GPU em alguns módulos, integrações com aprendizado de máquina clássico e *deep learning* (módulo dnn). Mantém Application Programming Interfaces (APIs) em C++, Python e Java, além de suporte para Android (TEAM, 2024).

A ampla adoção decorre de três fatores: desempenho prático, amplitude de funcionalidades e comunidade ativa, tornando-a natural para prototipagem e implantação de pipelines de visão no dispositivo.

2.3.2 Arquitetura e Módulos Relevantes

A biblioteca é organizada em módulos, cada um responsável por um conjunto específico de funcionalidades:

- `core`: fornece as estruturas de dados fundamentais, como `Mat`, e operações básicas sobre matrizes.
- `imgproc`: concentra funções de pré-processamento, incluindo filtros, limiarização, transformações geométricas e detecção de contornos.
- `features2d`: implementa algoritmos de detecção e descrição de pontos de interesse, como SIFT, SURF e ORB.
- `ml`: disponibiliza algoritmos de aprendizado de máquina, incluindo o KNN.
- `video`: oferece técnicas de análise temporal, como rastreamento de objetos e fluxo óptico.
- `dnn`: possibilita a inferência de redes neurais treinadas em *frameworks* externos.

Essa modularização favorece a construção de pipelines completos de visão, desde a captura da imagem até a classificação, adaptando-se a diferentes requisitos de desempenho e robustez.

2.3.3 Pré-processamento e Extração de Características

O pré-processamento é um passo essencial para lidar com as variações típicas de ambientes internos, como mudanças de iluminação ou presença de ruídos. Entre as técnicas mais utilizadas destacam-se: conversão para escala de cinza, equalização de histograma, filtragem Gaussiana e operações morfológicas (BARELLI, 2018; DOMíNGUEZ; PASCUAL, 2017). Esses procedimentos aumentam a consistência dos dados de entrada, reduzindo o impacto de fatores externos na etapa de classificação.

A extração de características pode seguir duas abordagens:

- **Descritores globais**: capturam informações de toda a imagem, como momentos de Hu (HU, 1962), histogramas normalizados e descritores de textura (LBP (OJALA; PIETIKäINEN; MäENPää, 2002), Haralick (HARALICK; SHANMUGAM; DINSTEIN, 1973)).
- **Descritores locais**: concentram-se em regiões específicas da imagem, sendo SIFT (LOWE, 2004), SURF (BAY; TUYTELAARS; GOOL, 2006) e ORB (RUBLEE et al., 2011) alguns dos mais populares. Esses descritores são particularmente úteis para tarefas de reconhecimento de objetos e localização robusta.

2.3.4 Integração com Android

A integração da OpenCV em aplicativos Android pode ser feita de duas formas: (i) via Software Development Kit (SDK) oficial (Android Archive (AAR)), utilizando APIs Java de alto nível, ou (ii) via Native Development Kit (NDK), permitindo acesso a implementações em C++ de maior desempenho (SIERRA, 2021). Em ambos os casos, é fundamental seguir boas práticas de desenvolvimento móvel:

- Captura eficiente de imagens com CameraX ou Camera2;
- Processamento em *threads* separadas para evitar bloqueio da interface;
- Reutilização de objetos `Mat` para reduzir overhead de memória;
- Redimensionamento controlado das imagens, equilibrando custo computacional e preservação de características relevantes.

Essas práticas garantem que os algoritmos de visão funcionem em tempo real, mesmo em dispositivos móveis com recursos limitados.

2.3.5 Aplicações em Robótica Indoor

Na robótica indoor, a OpenCV tem sido amplamente utilizada em tarefas como detecção de obstáculos, segmentação de ambientes, reconhecimento de lugares e rastreamento de objetos (DOMíNGUEZ; PASCUAL, 2017). Em cenários de navegação autônoma, destaca-se sua aplicação no *place recognition*, onde descritores globais ou locais permitem identificar se o robô já visitou determinado ambiente. Além disso, técnicas de pré-processamento auxiliam na robustez contra variações de iluminação ou presença de objetos móveis.

O uso da biblioteca em sistemas embarcados e dispositivos móveis se deve justamente à sua eficiência e à diversidade de algoritmos disponíveis, que vão desde operações básicas até redes neurais mais recentes.

2.3.6 Aplicação no TCC

Neste TCC, a biblioteca OpenCV será utilizada para:

1. Conversão de imagens do formato *Bitmap* para *Mat*;
2. Pré-processamento para redução de ruído e padronização;
3. Extração de descritores através de LBP;
4. Integração com o algoritmo KNN para classificação dos ambientes.

Espera-se avaliar diferentes combinações de descritores e estratégias de pré-processamento, buscando conciliar precisão na identificação dos ambientes e eficiência no tempo de resposta no dispositivo Android.

KNN e OpenCV formam um par complementar: o primeiro fornece uma regra de decisão simples e eficaz quando munido de descritores informativos; a segunda oferece ferramentas robustas para obter tais descritores de forma eficiente no Android. Com normalização, escolha criteriosa de k e distância, e descritores adequados (Hu/LBP/ORB), alcança-se equilíbrio entre acurácia e tempo de resposta, central para a localização *indoor* no robô móvel.

2.4 Arquitetura de Aplicações Android

Este tópico apresenta os principais conceitos sobre a arquitetura de aplicações Android, que serve como base para o sistema desenvolvido neste trabalho. Compreender como o Android organiza seus componentes e camadas de software é essencial para entender de que forma o aplicativo se comunica com o robô, processa dados e executa as funções

de controle e localização. Assim, este item traz uma visão geral da estrutura do Android e dos elementos que formam um aplicativo, servindo de apoio para o desenvolvimento da proposta apresentada no capítulo 3.

2.4.1 Camadas de Baixo Nível do Android

A plataforma Android é estruturada em uma arquitetura modular composta por várias camadas, sendo cada uma delas responsável por uma função específica dentro do sistema. Essa arquitetura visa fornecer uma base estável, segura e flexível para o desenvolvimento de aplicativos em dispositivos móveis. As principais camadas são: o *kernel* do Linux, a Android Runtime (ART), a Hardware Abstraction Layer (HAL), as bibliotecas nativas e o Framework Android.

2.4.1.1 Linux Kernel

O *kernel* do Linux é a camada fundamental do sistema Android, sendo responsável por gerenciar recursos essenciais como processos, memória, arquivos, *drivers* de *hardware* e comunicação entre componentes. O Android faz uso de uma versão personalizada do *kernel*, incluindo subsistemas específicos como:

- Binder IPC: mecanismo de comunicação entre processos usado amplamente no Android para conectar componentes do sistema e aplicativos de forma segura e eficiente.
- *Power Management*: controle detalhado de uso de energia por meio de *wakelocks* e outras políticas que otimizam o consumo da bateria.
- *Drivers* específicos: para Wi-Fi, câmera, sensores, *touchscreen*, entre outros, que garantem o funcionamento adequado do *hardware* nos dispositivos Android.

Essas modificações tornam o *kernel* adaptado às demandas energéticas, de segurança e de responsividade características de sistemas embarcados móveis (DEVELOPERS, 2024a).

2.4.1.2 Android Runtime (ART)

A ART substituiu a antiga Dalvik VM a partir da versão 5.0 do sistema. Sua principal característica é o uso da compilação Ahead-Of-Time (AOT), que converte o *bytecode* dos aplicativos em código nativo no momento da instalação, resultando em ganhos significativos de desempenho.

Além da AOT, o ART também oferece:

- Coleta de lixo otimizada, que melhora a alocação de memória e reduz pausas indesejadas.
- Perfis de uso: o sistema analisa como o aplicativo é utilizado e recompila apenas os trechos mais acessados, otimizando ainda mais o desempenho e reduzindo o tempo de instalação e atualização (DEVELOPERS, 2024b).
- Compatibilidade com linguagens modernas como Kotlin, que passou a ser a linguagem recomendada pelo Google a partir de 2019.

2.4.1.3 Hardware Abstraction Layer (HAL)

A HAL atua como ponte entre o *framework* do Android e os *drivers* de *hardware* fornecidos por fabricantes. Cada tipo de dispositivo (câmera, sensor, GPS, etc.) possui uma interface HAL específica que define os métodos necessários para que o sistema possa utilizá-lo de forma padronizada.

A grande vantagem da HAL é permitir que fabricantes implementem suas próprias versões de hardware sem impactar diretamente o sistema ou os aplicativos desenvolvidos sobre ele. Com isso, o Android alcança alta portabilidade e compatibilidade entre dispositivos (DEVELOPERS, 2024d).

2.4.2 Padrões Arquiteturais em Aplicações Android

O desenvolvimento de aplicações Android modernas exige a adoção de arquiteturas bem definidas, com foco em escalabilidade, testabilidade e separação de responsabilidades. Dentre os padrões mais utilizados estão:

O Google formalizou uma proposta chamada Arquitetura Android Moderna (Modern Android Architecture), que combina os padrões MVVM ou MVI com um conjunto de bibliotecas oficiais do Jetpack, compondo um modelo robusto, reativo e desacoplado. Os principais componentes dessa arquitetura incluem:

- ViewModel: armazena e gerencia dados da interface mesmo após mudanças de configuração;
- LiveData / StateFlow: possibilitam reatividade entre a UI e a lógica de apresentação;
- Repository: atua como camada intermediária entre a lógica da aplicação e as fontes de dados (Room, Retrofit, etc.);
- Room: fornece acesso seguro e eficiente ao banco de dados SQLite;

Quadro 1 – Comparação entre padrões arquiteturais em Android

Padrão	**Estrutura**	**Principais Características**
Model-View-Controller (MVC)	(i) Model: Dados e lógica de negócio. (ii) View: Interface com o usuário. (iii) Controller: Coordena ações entre Model e View.	Padrão clássico. Pouco utilizado no Android moderno por dificultar testabilidade e apresentar acoplamento elevado.
Model-View-Presenter (MVP)	(i) Model: Dados. (ii) View: Interface. (iii) Presenter: Lógica da View desacoplada.	Mais testável que o MVC. Muito utilizado antes da popularização do MVVM e da arquitetura oficial do Google.
Model-View-ViewModel (MVVM)	(i) Model: Dados. (ii) View: Interface. (iii) ViewModel: Lógica da UI e estado da tela.	Recomendado pelo Google. Ótimo para integração com LiveData, StateFlow e Data Binding. Ideal para aplicativos modernos.
Model-View-Intent (MVI)	(i) Model: Estado imutável. (ii) View: Exibe estado atual. (iii) Intent: Representa ações do usuário.	Fluxo unidirecional. Bastante usado com Jetpack Compose ou *frameworks* reativos como RxJava.

Fonte: Elaborada pelo autor.

- WorkManager / CoroutineWorker: executam tarefas assíncronas ou programadas, mesmo com o aplicativo fechado;
- Jetpack Compose (opcional): *framework* declarativo para construção de interfaces modernas.

Segundo a documentação oficial, essa arquitetura promove aplicações mais resilientes, testáveis e alinhadas às boas práticas de engenharia de *software* móvel (DEVELOPERS, 2024c).

2.4.3 Integração entre Aplicativos Android e Hardware Externo

Aplicações Android que interagem com hardware externo — como robôs, sensores, câmeras ou dispositivos customizados — podem ser construídas utilizando diferentes estratégias de integração. Entre as abordagens mais comuns, destacam-se:

2.4.3.1 SDKs e APIs

Fabricantes de hardware frequentemente disponibilizam Software Development Kits (SDKs) próprios para facilitar a integração com seus dispositivos. Esses SDKs podem incluir bibliotecas Java/Kotlin, documentação técnica e exemplos de uso, permitindo que os desenvolvedores utilizem os recursos do *hardware* sem precisar lidar com protocolos de comunicação de baixo nível.

Exemplo: câmeras Internet Protocol (IP), impressoras térmicas, dispositivos biométricos e até robôs educacionais (como o LEGO Mindstorms) costumam vir com SDKs próprios para Android.

2.4.3.2 Protocolos de Comunicação Direta

Quando não há SDKs disponíveis, os aplicativos Android podem se comunicar diretamente com o *hardware* por meio da troca de dados em formato binário ou textual. As formas mais comuns incluem:

- User Datagram Protocol (UDP) ou Transmission Control Protocol (TCP)/IP: quando o hardware está conectado via rede Wi-Fi ou Ethernet. O aplicativo pode enviar comandos diretamente para o IP do dispositivo, como ocorre em sistemas de controle remoto.
- Bluetooth/Bluetooth Low Energy (BLE): protocolos amplamente usados para integração com sensores e atuadores de curto alcance.
- Universal Serial Bus On-The-Go (USB OTG): para comunicação com dispositivos que suportam conexão física direta, como Arduino, Raspberry Pi, etc.
- Serial via Universal Serial Bus (USB): uso de bibliotecas como usb-serial-for-android permite comunicação serial entre o Android e microcontroladores.

A arquitetura do sistema Android e os padrões empregados no desenvolvimento de aplicativos são elementos fundamentais para garantir qualidade, desempenho e escalabilidade. Para aplicações que vão além da simples interação com o usuário — como aquelas que controlam robôs móveis, capturam vídeo em tempo real ou se comunicam via rede — é imprescindível um projeto arquitetônico sólido. A compreensão das camadas do sistema, das bibliotecas de suporte e das estratégias de integração com hardware externo oferece a base técnica necessária para a construção de soluções robustas e inovadoras no ecossistema Android.

2.5 Robô Móvel

O robô utilizado neste trabalho teve sua origem em um projeto anterior do Instituto Federal de Santa Catarina (IFSC) câmpus São José, apresentado no trabalho *Sistema Automatizado de Inspeção de Dutos de Sistemas de Condicionamento de Ar*, por Pinho et al. (2015). Naquela proposta, o robô foi desenvolvido para auxiliar na inspeção visual interna de dutos de sistemas de condicionamento de ar, permitindo a movimentação do robô em locais de difícil acesso e a transmissão de imagens para auxiliar na análise do interior dos dutos.

Atualmente, esse robô é utilizado em outro contexto, ele está em desenvolvimento no projeto de extensão PJ482-2024 (*Divulgação da área de telecomunicações: experimento demonstrativo de comunicação via link simulado com satélite*) do IFSC campus São José. Seu propósito foi redirecionado para atividades de demonstração e divulgação da área de telecomunicações, servindo como plataforma móvel experimental para testes e apresentações. É nesse cenário que se insere o aplicativo Android desenvolvido neste trabalho.

O robô utiliza como unidade principal uma BeagleBone Black executando Linux Debian que integra controle de motores, comunicação sem fio e captura de dados em tempo real. O chassi possui tração do tipo lagarta, com dois motores DC de 12 V controlados por um *driver* TB6612FNG, garantindo tração robusta. A placa PWM PCA9685, acessada via barramento I²C, aciona servomotores que movimentam uma antena direcional, capaz de varrer suavemente o espaço horizontal e vertical.

No subsistema de comunicação, o robô combina múltiplas tecnologias sem fio. A interface NRF24L01+ via SPI1 possibilita experimentos de rádio ponto a ponto, enquanto módulos Zigbee (XBee S2C e Xiao ESP32-C6) permitem testes de redes em malha e telemetria. Ele também pode operar como *access point* Wi-Fi, permitindo a conexão direta de um *smartphone* controlador, que recebe vídeo em tempo real e envia comandos de movimento e ajuste de antena. Essa arquitetura híbrida foi projetada para o estudo de protocolos de rede e comportamento dinâmico de enlace, incluindo latência, perdas e *throughput* sob condições controladas de interferência e variação de potência.

O sistema sensorial integra uma câmera digital USB voltada para aplicações de visão computacional embarcada, possibilitando experimentos de reconhecimento de imagem e rastreamento de alvos. A orientação espacial é obtida pelo módulo MPU-9250, que combina acelerômetro, giroscópio e magnetômetro em nove eixos, fornecendo dados para estimar o rumo (bússola) e a inclinação da antena. Esses sensores permitem avaliar o impacto da vibração, do movimento e do ambiente eletromagnético na estabilidade da comunicação e na precisão do apontamento. A coleta e fusão dos dados é feita em Python, com integração via I²C e suporte a bibliotecas de processamento numérico.

A inteligência do sistema é distribuída: a BeagleBone executa o controle motor e a aquisição dos sensores, enquanto o processamento de visão e decisão ocorre preferencialmente no dispositivo remoto (celular ou notebook). Essa separação possibilita medir o impacto da transmissão de dados multimídia e de algoritmos de Inteligência Artificial (IA) sobre o desempenho e o consumo energético do robô.

O projeto, conhecido como “Satélite Simulado” ou “Landell Rover”, tem caráter pedagógico e demonstrativo, e inclui um módulo simulador de satélite que fica suspenso, permitindo a comunicação robô com satélite. Também pode funcionar como laboratório para alunos explorarem integração entre sensores, atuadores e redes, aplicando na prática princípios de engenharia de computação, eletrônica e telecomunicações.

3 ESPECIFICAÇÕES E IMPLEMENTAÇÃO DO SISTEMA

Neste capítulo são apresentadas as especificações e a implementação do aplicativo desenvolvido. Primeiro é feita uma visão geral do funcionamento do robô em conjunto com o aplicativo Android, destacando os fluxos principais de comandos e de vídeo. Em seguida, é descrita a estrutura do aplicativo, incluindo a tela principal, a organização por abas e as funcionalidades de cada uma delas. Por fim, são discutidas algumas decisões e restrições técnicas consideradas durante a implementação, como compatibilidade com a versão do Android, organização do código e cuidados para manter o aplicativo responsivo durante comunicação de rede, *streaming* e processamento de imagens.

3.1 Visão Geral do Sistema

O sistema é composto por dois elementos principais: um robô móvel e um aplicativo Android responsável por enviar comandos ao robô e receber o vídeo da câmera. O objetivo do aplicativo é permitir o controle remoto do robô (movimentação, câmera, antena e funções do sistema) e também oferecer uma funcionalidade de reconhecimento de ambientes internos, que responde a pergunta "onde estou?"a partir da análise de imagens.

O robô está sendo desenvolvido no contexto de um projeto de extensão (PJ482-2024) do IFSC campus São José e tem como objetivo promover a área de telecomunicações para a comunidade, em especial para potenciais candidatos aos cursos ofertados pelo campus. A figura 1 mostra como está o robô atualmente. Dentro desse projeto, este trabalho contribui com o desenvolvimento de um aplicativo Android que amplia as formas de interação com o robô, oferecendo uma interface de controle remoto e apoio a testes e demonstrações.

Para viabilizar essa integração, o robô disponibiliza um protocolo de comunicação baseado em mensagens JSON minificadas. Os comandos são recebidos via rede, interpretados e respondidos conforme a documentação. Este recurso permite ampla compatibilidade, inclusive com aplicativos Android, pois não depende de uma plataforma específica.

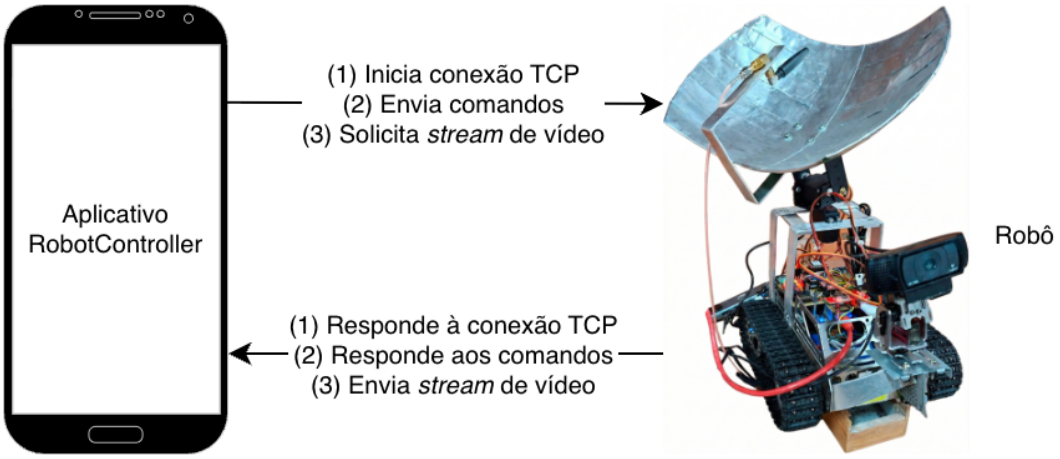
Conforme mostra a figura 1, o aplicativo trabalha com dois fluxos principais:

- Fluxo de comandos: o usuário interage com os controles na interface, o aplicativo monta uma mensagem de comando e envia ao robô via conexão Transmission Control Protocol (TCP). O protocolo de aplicação utilizado organiza os comandos em um

formato de mensagem JSON minificado com módulo e ação, e o robô retorna uma resposta indicando sucesso ou erro da operação.

- Fluxo de vídeo: o robô fornece um *streaming* Motion Joint Photographic Experts Group (MJPEG) em tempo real e o aplicativo exibe os *frames* na tela principal, permitindo que o usuário acompanhe o que o robô "está vendo" durante os movimentos. No protocolo, existe inclusive um módulo específico para iniciar e parar o *streaming* de vídeo.

Figura 1 – Fluxos entre aplicativo e robô



Fonte: Elaborada pelo autor.

Além disso, existe um módulo de autonomia no aplicativo com duas ações: *Mapear ambiente* e *Onde estou?*. No mapeamento, o usuário define um nome para o ambiente e fornece quatro imagens (Norte, Leste, Sul e Oeste). No "onde estou?", o usuário fornece novamente quatro imagens e o sistema compara com os ambientes já salvos para estimar o ambiente mais provável. Para facilitar testes, essa etapa foi feita de forma que a fonte das imagens possa ser trocada: hoje as imagens vêm da galeria do *smartphone*, mas a intenção é que depois elas possam vir diretamente dos *frames* do vídeo do robô, sem precisar alterar a lógica principal do reconhecimento.

Do ponto de vista do usuário, os principais casos de uso do aplicativo são:

- Configurar os parâmetros de rede do robô (IP/domínio e portas) e iniciar a conexão.
- Controlar a movimentação do robô (frente, ré, esquerda, direita e parada), com ajuste de velocidade.
- Controlar a câmera do robô, tanto por movimentação contínua quanto por posicionamento absoluto (*pan/tilt*), incluindo a opção de centralizar ($pan = 90°$ e $tilt = 90°$).

- Controlar a antena do robô por posicionamento (*pan/tilt*), iniciar varredura completa e obter informações de posição e Link Quality Indicator (LQI).
- Executar comandos gerais do sistema (por exemplo, parada de emergência e leitura de bússola).
- Iniciar/parar o serviço de *streaming* e visualizar o vídeo do robô na tela principal.
- Mapear um ambiente interno a partir de quatro imagens, salvando os descritores localmente.
- Estimar o ambiente atual comparando novas imagens com a base já mapeada.

Os detalhes de interface e funcionamento de cada funcionalidade são apresentados nas seções a seguir.

3.2 Especificações Funcionais do Aplicativo

O aplicativo foi desenvolvido em *Kotlin*, que é linguagem de programação nativa para Android. A interface gráfica foi feita com Extensible Markup Language (XML). A tela principal foi organizada para concentrar o vídeo e, ao mesmo tempo, separar os controles em abas. Cada aba lida com um contexto de controles diferentes.

3.2.1 Tela Principal e Navegação

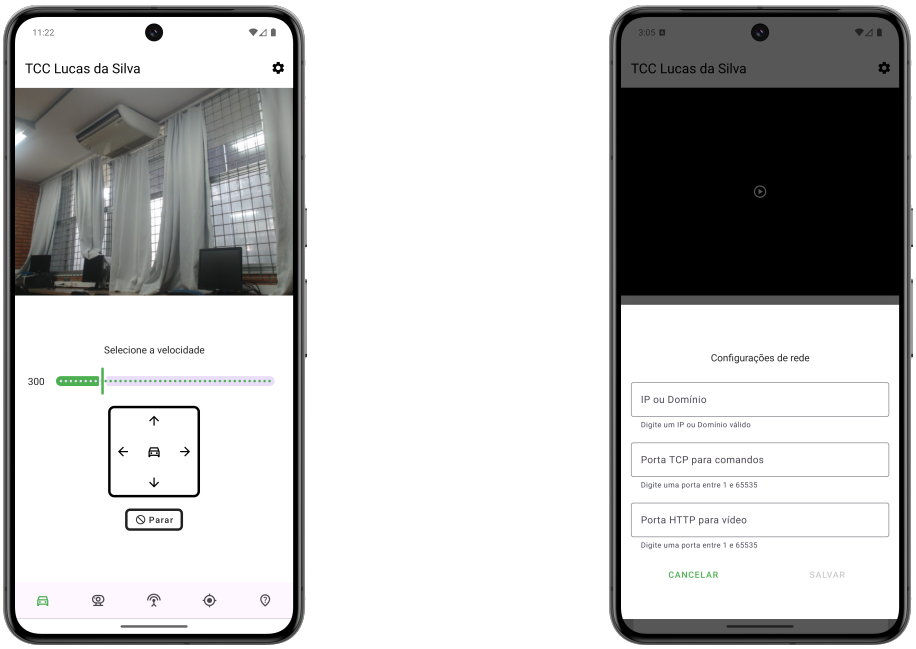
A *MainActivity* é a base do aplicativo. Ela possui uma MaterialToolbar no topo, uma ImageView dedicada para exibição do vídeo MJPEG e um conjunto ViewPager2 + TabLayout para navegação entre as funcionalidades. Com isso, o usuário consegue ver o vídeo o tempo todo e alternar entre os controles de tração, câmera, antena, sistema e autonomia apenas trocando de aba, como mostra a figura 2a.

Na própria Toolbar existe um ícone de configurações que abre uma *bottom sheet*. Nela, o usuário informa os parâmetros de rede do robô (Internet Protocol (IP) ou domínio, porta TCP de comandos e porta Hypertext Transfer Protocol (HTTP) de vídeo), conforme figura 2b. Ao salvar, essas informações são armazenadas no SharedPreferences do Android para facilitar o uso nas próximas execuções. A partir dessas configurações, o aplicativo consegue estabelecer a comunicação de comandos e iniciar o fluxo de vídeo.

3.2.2 *Fragments* de Controle

Os controles do robô foram separados em quatro abas (*fragments*) principais, cada uma focada em um contexto específico:

Figura 2 – Tela principal e configurações de rede



(a) Tela principal do aplicativo com vídeo (b) *Bottom sheet* de configurações de rede

Fonte: Elaborada pelo autor.

1. TractionControllerFragment: responsável pelos comandos de movimentação do robô (tração), como deslocamento e giros. Também possui a configuração de velocidade de tração do robô. Esse conjunto de comandos aparece no protocolo como um módulo específico de tração. A interface pode ser vista na figura 2a.

2. CameraControllerFragment: responsável pelo controle de *pan/tilt* da câmera. O protocolo também prevê comandos tanto de movimento contínuo (através das setas) quanto de ajuste absoluto de ângulo, o que facilita implementar diferentes modos de controle na interface. A tela pode ser vista na figura 3a.

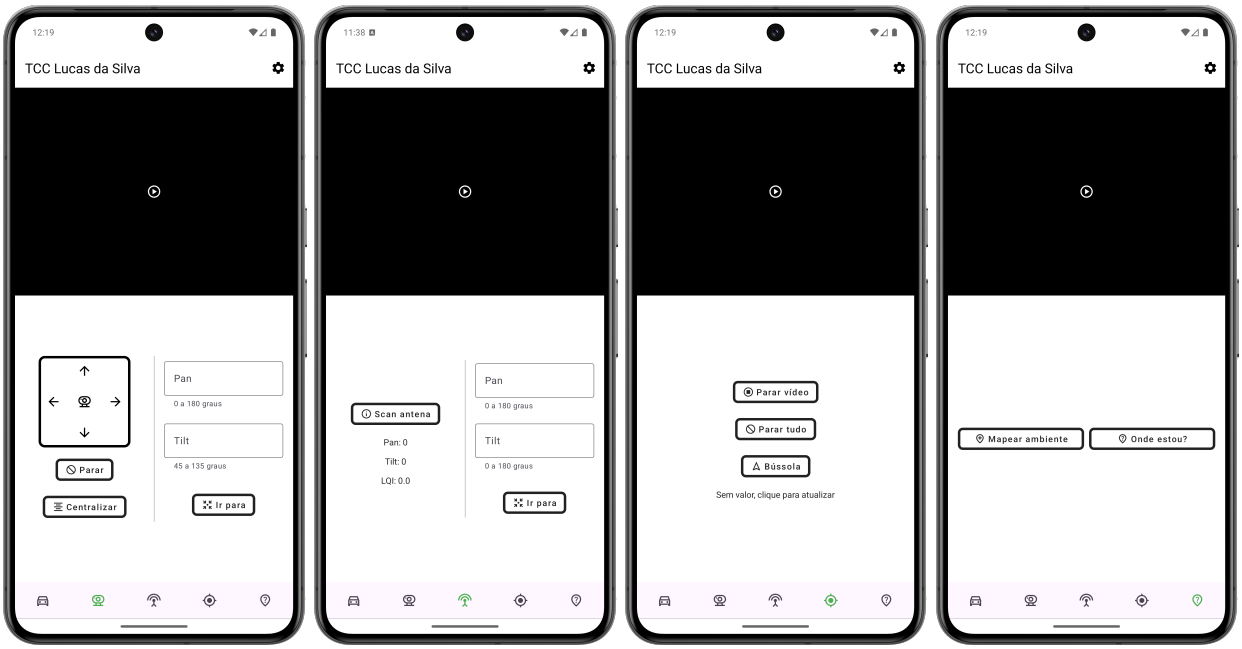
3. AntennaControllerFragment: responsável pelo controle de *pan/tilt* da antena através de comandos de movimento absoluto de ângulo, início de varredura completa e obtenção de informações relacionadas, como o LQI. A interface pode ser vista na figura 3b.

4. SystemControllerFragment: concentra comandos mais gerais do sistema, como parada geral (parada de emergência), parada de vídeo e leitura da bússola. A interface pode ser vista na figura 3c.

3.2.3 *Fragment* de Autonomia

O AutonomyControllerFragment é o *fragment* responsável pelo reconhecimento de ambientes. A interface pode ser vista na figura 3d. Ele possui dois botões:

Figura 3 – *Fragments* do aplicativo



(a) CameraControllerFragment (b) AntennaControllerFragment (c) SystemControllerFragment (d) AutonomyControllerFragment

Fonte: Elaborada pelo autor.

- Mapear ambiente: abre um fluxo onde o usuário informa o nome do ambiente e seleciona quatro imagens da galeria do *smartphone*. Elas são interpretadas na ordem Norte -> Leste -> Sul -> Oeste. Depois disso, o aplicativo processa essas imagens com OpenCV para extrair descritores e salva os dados localmente associados ao nome do ambiente. No fim do processo, as imagens são descartadas para otimizar consumo de memória do aplicativo.
- Onde estou?: abre um fluxo onde o usuário seleciona quatro imagens atuais, o aplicativo extrai descritores e realiza a classificação comparando com os ambientes já mapeados, retornando o ambiente mais provável. Nesse fluxo, os descritores das imagens selecionadas não são salvos, eles são usados apenas para a classificação.

Essa parte foi organizada de tal maneira que, futuramente, permita a substituição da seleção de imagens na galeria pelas imagens vindas do *streaming* de vídeo do robô.

3.3 Especificações Não Funcionais e Restrições

Além das funções do aplicativo, algumas decisões foram tomadas por questões práticas de implementação, testes e desempenho.

Em relação à compatibilidade, o aplicativo foi desenvolvido para ser compatível com Android 13 (Application Programming Interface (API) 33) ou superior. Essa escolha

foi feita para simplificar integrações utilizadas no projeto, como a seleção de imagens e a parte de permissões, além de evitar a necessidade de lidar com muitas variações de comportamento presentes em versões mais antigas do Android.

Uma das decisões principais foi deixar o módulo de reconhecimento de ambientes independente da fonte das imagens. Isso ajuda a testar mais rápido (por exemplo, repetindo o mesmo conjunto de imagens) e também facilita a evolução do projeto, já que a mesma lógica pode receber imagens de outra origem depois, como *frames* do *streaming*.

Outra preocupação foi manter o aplicativo responsivo. O sistema envolve comunicação de rede, exibição de vídeo e processamento de imagens com OpenCV. Por isso, as etapas mais pesadas (como extração de descritores, leitura/gravação de dados e classificação) precisam ser executadas fora da *thread* principal, evitando travamentos na interface.

Também foi considerada a organização do código para facilitar manutenção e testes. Como existem várias funcionalidades (tração, câmera, antena, sistema e autonomia), a separação por *fragments* ajuda a manter responsabilidades claras. Do lado do reconhecimento, a ideia é permitir mudanças rápidas, como troca de extrator de descritores ou alteração do parâmetro k do K-Nearest Neighbors (KNN), sem precisar reescrever o fluxo inteiro.

3.4 Arquitetura e Organização do Código do Aplicativo

Para organizar o código e evitar que regras de negócio ficassem misturadas com a interface, o aplicativo foi estruturado seguindo o padrão Model-View-ViewModel (MVVM). A ideia principal desse padrão é deixar a interface (*Activity/Fragments*) mais simples, enquanto o *ViewModel* concentra as decisões de fluxo, chamadas para repositórios e atualização do estado exibido na tela.

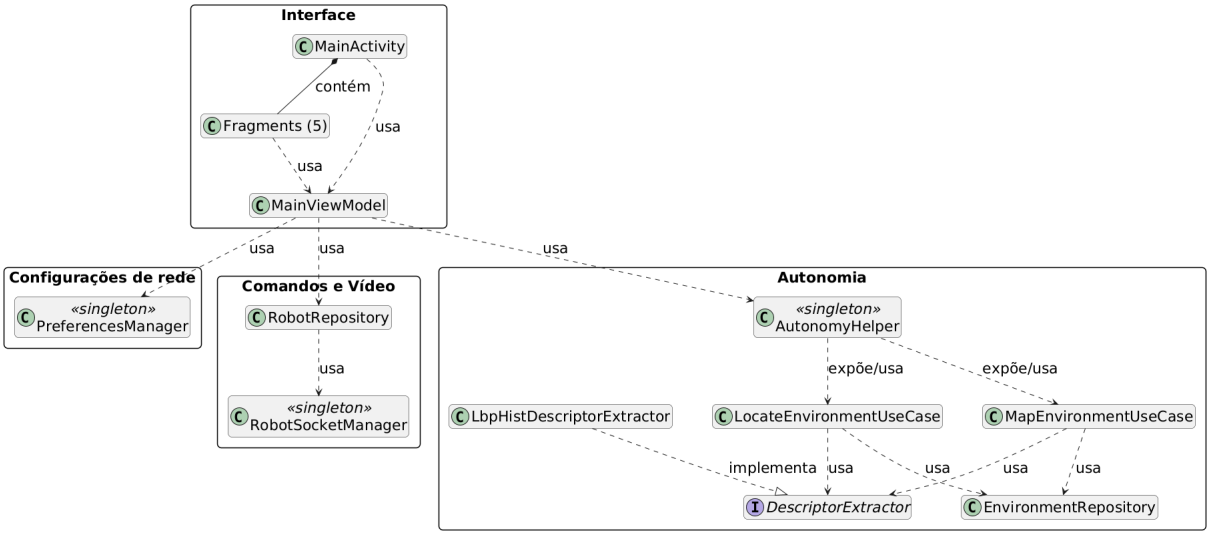
No aplicativo, a tela principal (*MainActivity*) fica responsável por configurar os componentes visuais (Toolbar, ImageView do vídeo e navegação por abas). Cada aba é um *Fragment* com uma responsabilidade específica (tração, câmera, antena, sistema e autonomia). Esses *fragments* disparam ações no *ViewModel* e observam mudanças de estado através de *LiveData*.

O *ViewModel* foi escolhido porque ele mantém dados de interface e lógica de orquestração de forma desacoplada da *Activity/Fragment*, e também porque ele é mantido enquanto o "dono" do ciclo de vida existir, ajudando a evitar perda de estado em mudanças de configuração (DEVELOPERS, 2025c).

3.4.1 Padrão MVVM e Responsabilidades

As responsabilidades ficam separadas assim:

Figura 4 – Diagrama de classes simplificado do aplicativo



Fonte: Elaborada pelo autor.

- *View (MainActivity e Fragments)*: mostra os componentes na tela, recebe cliques/entradas do usuário e chama métodos do *ViewModel*. Também observa *LiveData* para atualizar a interface (por exemplo, atualizar o *frame* do vídeo na *ImageView*, mostrar mensagens de *feedback*, habilitar/desabilitar botões, etc.).

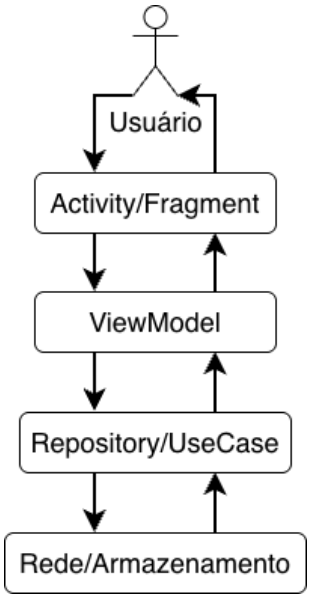
- *ViewModel (MainViewModel)*: centraliza o controle de fluxo da tela. Ele expõe estados por *LiveData* e contém métodos que iniciam operações como: conectar, iniciar vídeo, parar vídeo, enviar comandos e executar operações de autonomia (mapear ambiente e localizar ambiente). O *ViewModel* também é responsável por transformar resultados (sucesso/erro) em algo que a interface consiga mostrar.

- *Model*/Camada de dados (*Repositories* e *Managers*): a camada de dados concentra o acesso às partes "externas" do aplicativo, como rede (*socket* de comandos, *stream* de vídeo) e persistência (preferências e dados de ambientes). No projeto, por exemplo, existe um manager para preferências (*SharedPreferences*) e repositórios para comunicação/persistência de dados.

- Camada de domínio (*UseCases*/operações): para a parte de autonomia, a regra de negócio (extrair descritor, salvar, carregar e classificar) fica em classes próprias (*UseCases*), que recebem repositórios e extratores por composição. Isso ajuda a manter o código mais separável e facilita testar/trocar partes sem mexer no restante.

A comunicação entre *ViewModel* e interface é feita usando *LiveData*, que é um tipo de dado observável e "ciente" do ciclo de vida da *Activity/Fragment*. Isso reduz problemas de atualizar a tela quando ela não está em um estado ativo (DEVELOPERS, 2025a).

Figura 5 – Arquitetura MVVM



Fonte: Elaborada pelo autor.

3.4.2 Programação Assíncrona

Como o aplicativo realiza comunicação de rede, recebimento de *stream* de vídeo e também processamento de imagens (OpenCV), várias operações podem ser demoradas. Por isso, o projeto utiliza *RxJava* para lidar com tarefas assíncronas, combinando chamadas e controlando em qual *thread* elas executam.

Na prática, a separação costuma seguir esta lógica:

- *Thread* de Input/Output (I/O): operações de rede e disco. Por exemplo, envio de comandos e recebimento de respostas, escrita e leitura de dados em armazenamento não volátil.
- *Thread* de *Computation*: operações pesadas de processamento. Por exemplo, extração de descritores das imagens com o OpenCV.
- *Thread* principal: operações de atualização de interface. Por exemplo, cliques de botões e *feedbacks* de sucesso ou falha de operações.

Além disso, para evitar vazamento de memória e manter o controle do ciclo de vida das assinaturas, o *ViewModel* utiliza um *CompositeDisposable*. No *RxJava*, quando uma operação é assinada (*subscribe*), ela retorna um *Disposable*, que é basicamente uma referência usada para "cancelar" aquela assinatura quando ela não for mais necessária. O *CompositeDisposable* funciona como um contêiner desses *Disposables*, permitindo descartar todos de uma vez (por exemplo, quando o *ViewModel* é destruído), evitando que

operações continuem rodando e tentando atualizar a interface quando a tela deixa de estar ativa (REACTIVEX, 2025).

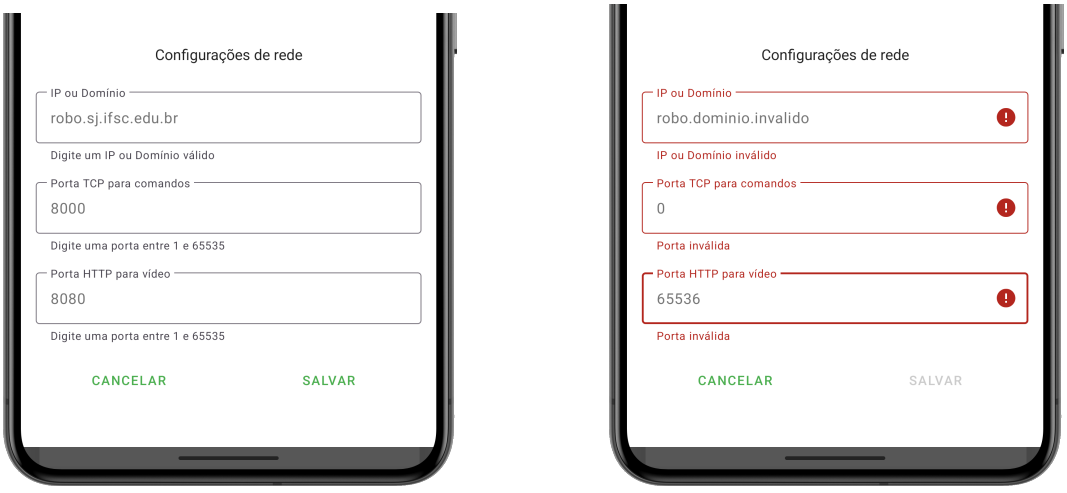
3.5 Configurações de Rede e Conexão com o Robô

Para que o aplicativo consiga se conectar ao robô em diferentes redes e situações de teste, foi implementado um fluxo de configuração acessível pela tela principal. Na *MainActivity*, existe um ícone de configurações na *MaterialToolbar*. Ao tocar nesse ícone, o aplicativo abre uma *bottom sheet* onde o usuário informa os parâmetros de rede necessários para a comunicação.

Os parâmetros configurados são:

- IP ou domínio do robô: identifica o dispositivo na rede.
- Porta TCP de comandos: utilizada para envio de comandos e recebimento de respostas.
- Porta HTTP de vídeo: utilizada para acessar o *streaming* de vídeo do robô.

Figura 6 – *Bottom sheet* com dados preenchidos



(a) Dados válidos (b) Dados inválidos

Fonte: Elaborada pelo autor.

Para reduzir erros durante a configuração, os campos da *bottom sheet* possuem validação em tempo real usando expressões regulares (*Regex*). Assim, enquanto o usuário digita, o aplicativo verifica se o IP/domínio e as portas estão em um formato válido, como na figura 6a. Se algum campo estiver inválido, o botão Salvar permanece desabilitado, como na figura 6b. O botão só é habilitado quando todos os campos estão válidos, evitando que o aplicativo tente se conectar com parâmetros incorretos.

Esses valores são persistidos no *SharedPreferences* para não exigir que o usuário digite novamente a cada execução do aplicativo. Ele é indicado para armazenar pares simples de chave-valor e costuma ser usado para configurações do aplicativo (DEVELOPERS, 2025b).

Para evitar espalhar o acesso ao *SharedPreferences* em vários pontos do código, foi utilizada uma classe *PreferencesManager*, que centraliza as operações de leitura e gravação dessas configurações. Dessa forma, a *Activity/ViewModel* não precisa lidar diretamente com as chaves e com a forma de armazenamento.

Além de salvar os dados, ao clicar sobre o botão Salvar, o aplicativo também tenta iniciar a conexão do canal de comandos. Isso foi feito para facilitar o uso, pois o usuário não precisa executar uma etapa extra de "conectar" depois de configurar. Assim, ao final do processo de configuração, o aplicativo já está preparado para enviar comandos pelas abas de controle.

A conexão de comandos é realizada através do *RobotRepository*, que utiliza internamente um *RobotSocketManager* para gerenciar o *socket* TCP. Caso o robô esteja *offline*, o IP/domínio esteja inválido ou a porta esteja incorreta, a conexão falhará. Nesses casos, o aplicativo mantém o estado de "não conectado" e exibe um *feedback* ao usuário, evitando que comandos sejam enviados sem uma conexão válida.

3.6 Comunicação por Comandos

A comunicação de comandos entre o aplicativo e o robô é feita por meio de um protocolo baseado em TCP e mensagens em JavaScript Object Notation (JSON) minificado. Cada comando enviado pelo aplicativo é uma linha de JSON finalizada por `\n`, e o robô retorna uma resposta também em JSON, indicando sucesso ou erro.

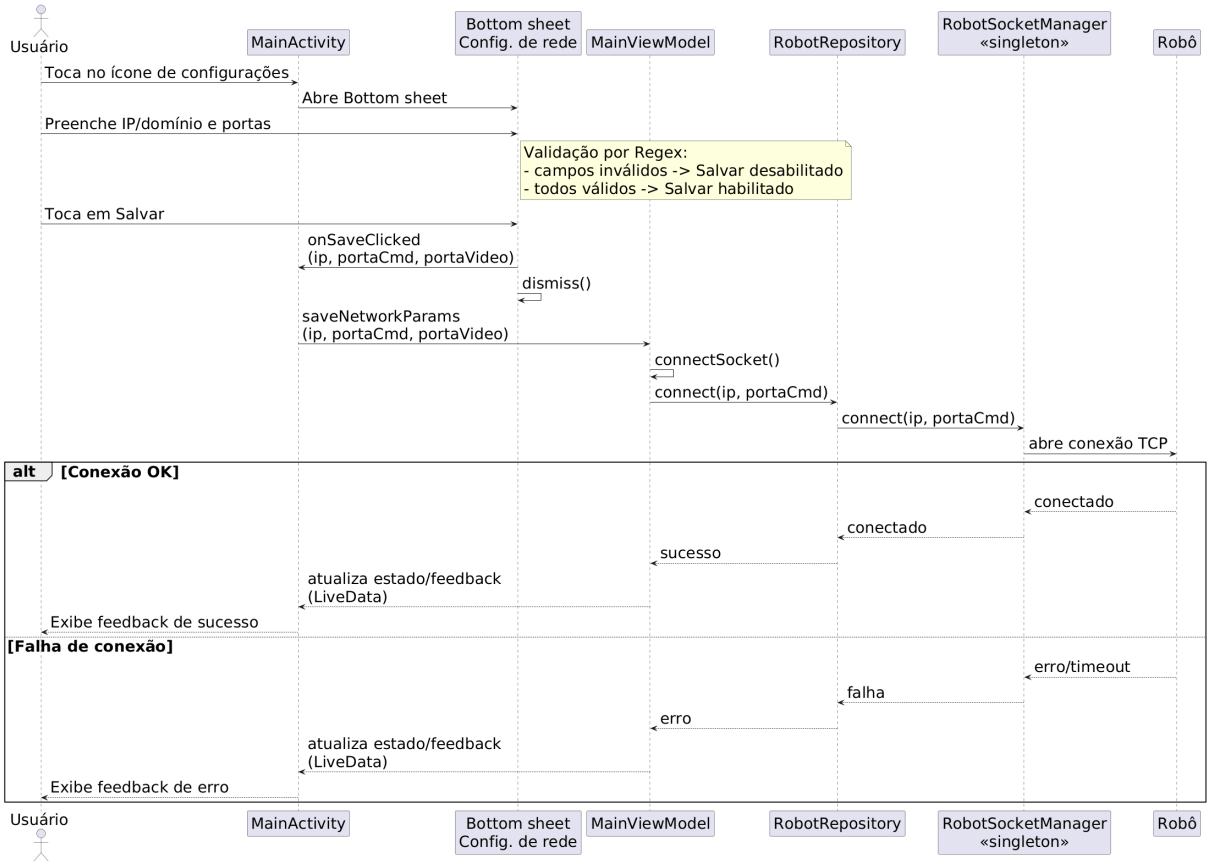
De forma geral, os comandos seguem um padrão com dois campos principais: m (módulo) e a (ação). Dependendo do módulo, outros campos podem ser incluídos. O robô responde com:

- `{"ok":1}` em caso de sucesso simples.
- `{"ok":1, ...}` quando há dados adicionais.
- `{"ok":0,"err":"...", ...}` em caso de erro.

Os códigos de erros padronizados são: `invalid_module`, `invalid_action`, `invalid_param` e `not_supported`.

O Quadro 2 apresenta exemplos representativos de comandos e respostas do protocolo JSON minificado, com base na especificação adotada no projeto. Esses exemplos

Figura 7 – Fluxo de configuração e conexão com o robô



Fonte: Elaborada pelo autor.

foram úteis para guiar a implementação e validar se as mensagens enviadas pelo aplicativo estavam no formato esperado.

Um comando que merece destaque é o de início do *streaming*: `{"m":"stream","a":"start"}`. Esse comando não transmite o vídeo pela conexão TCP. Ele solicita ao robô que inicie o serviço responsável por disponibilizar o *streaming*. A partir desse momento, o aplicativo pode consumir o vídeo por HTTP usando a porta configurada para o vídeo. Os detalhes do consumo do *stream* são apresentados na seção 3.7

No aplicativo, a implementação dos comandos ficou centralizada no *RobotRepository*, que utiliza o *RobotSocketManager* para gerenciar a conexão TCP. A resposta do robô é lida na mesma conexão e interpretada pelo aplicativo. Para facilitar a construção e o *parse* das mensagens, foi utilizada serialização/desserialização em JSON com a biblioteca Gson. No uso do aplicativo, quem dispara essas ações é o *MainViewModel*, enquanto os *Fragments* ficam responsáveis por capturar a ação do usuário e exibir o retorno na tela.

A Figura 8 apresenta o diagrama de sequência do fluxo de envio e resposta de um comando. O exemplo mostrado representa um comando simples (`stop_all`), mas o comportamento geral é o mesmo para os demais comandos: o usuário aciona um botão, o *fragment* repassa para o *MainViewModel*, o *RobotRepository* envia o comando ao *Ro-*

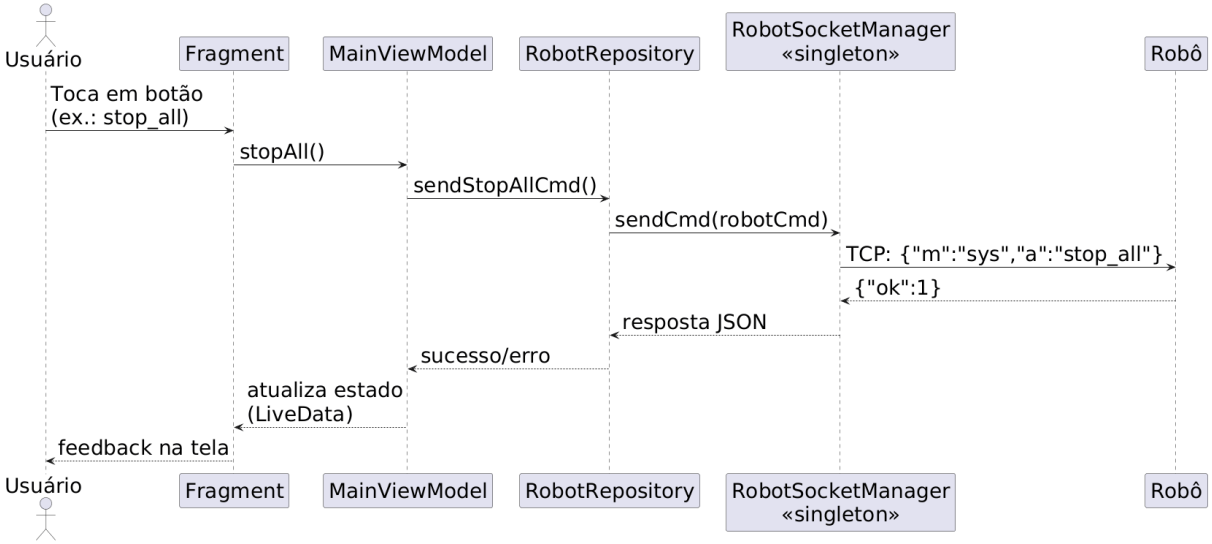
Quadro 2 – Exemplos de comandos e respostas do protocolo JSON minificado

Módulo	**Ação**	**Exemplo de envio (JSON)**	**Exemplo de resposta (JSON)**
`tra`	`go`	`{"m":"tra","a":"go", "d":"w"}`	`{"ok":1}`
`tra`	`go` (com velocidade)	`{"m":"tra","a":"go", "d":"w","v":800}`	`{"ok":1}`
`tra`	`stop`	`{"m":"tra","a":"stop"}`	`{"ok":1}`
`tra`	`status`	`{"m":"tra","a":"status"}`	`{"ok":1,"dir":"w", "v":700}`
`cam`	`set` (pan/tilt)	`{"m":"cam","a":"set", "pan":120,"tilt":90}`	`{"ok":1}`
`sys`	`stop_all`	`{"m":"sys", "a":"stop_all"}`	`{"ok":1}`
`compass`	`read`	`{"m":"compass", "a":"read"}`	`{"ok":1,"deg":274.3}`
`stream`	`start`	`{"m":"stream", "a":"start"}`	`{"ok":1}`
`tra`	erro (velocidade inválida)	`{"m":"tra","a":"go", "d":"w","v":99999}`	`{"ok":0, "err":"invalid_speed", "min":0,"max":1500}`

Fonte: Elaborado pelo autor com base na especificação do protocolo.

botSocketManager, que envia o JSON pela conexão TCP e aguarda a resposta do robô indicando sucesso ou erro.

Figura 8 – Fluxo de envio de comandos e recebimento de respostas do robô



Fonte: Elaborada pelo autor.

3.7 *Streaming* de Vídeo

Além do canal de comandos descrito na seção 3.6, o aplicativo também recebe o vídeo do robô para exibição na tela principal. Esse vídeo é consumido como um *stream* MJPEG, ou seja, uma sequência contínua de imagens Joint Photographic Experts Group (JPEG) enviada pelo robô, que o aplicativo vai recebendo e exibindo *frame* a *frame*.

O *streaming* não é iniciado automaticamente. Antes de requisitar o vídeo, o aplicativo envia um comando ao robô para iniciar o serviço responsável por disponibilizar o *stream*. Esse controle é feito pelo módulo *stream* do protocolo:

- Iniciar o serviço: `{"m":"stream","a":"start"}`
- Parar o serviço: `{"m":"stream","a":"stop"}`

É importante destacar que esses comandos não carregam o vídeo pela conexão TCP. Eles apenas controlam o serviço no robô (servidor MJPEG). Depois que o serviço está ativo, o vídeo passa a ficar disponível para ser consumido via HTTP, usando a porta de vídeo configurada no aplicativo.

Além do *start/stop*, o protocolo também prevê comandos para configuração e consulta de status do *streaming* (como resolução, Frames Per Second (FPS) e porta). No projeto, a ideia é utilizar esses comandos principalmente como apoio de desenvolvimento, para testes e *debug*, eles não aparecem diretamente na interface para o usuário final. Os retornos desses comandos podem ser acompanhados pelo desenvolvedor via *logs* (no Logcat do Android Studio), auxiliando a validar se o robô está com a configuração esperada e se o serviço está rodando corretamente.

Com o serviço iniciado, o aplicativo consome o fluxo de vídeo por uma Uniform Resource Locator (URL) HTTP no formato:

```
http://<ip_ou_dominio>:<porta_http_video>/?action=stream
```

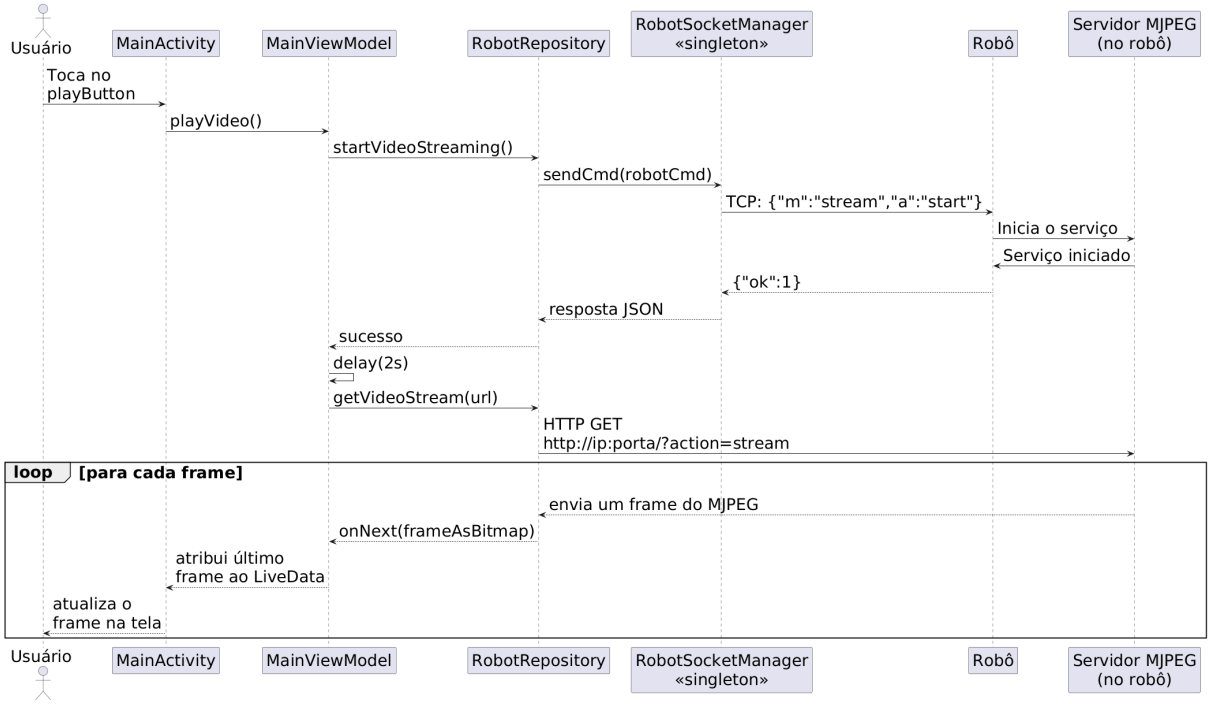
No aplicativo, o vídeo é iniciado pelo método *playVideo()* no *MainViewModel*. De maneira resumida, esse fluxo acontece assim:

1. O *ViewModel* verifica se o robô está conectado.
2. Envia o comando para iniciar o serviço de *streaming* no robô.
3. Monta a URL com o endereço e a porta do vídeo.
4. Realiza a requisição HTTP para a URL do *stream* e passa a receber os *frames*.
5. A cada *frame* recebido, o *ViewModel* atualiza o *LiveData* utilizado pela interface.

Na prática, o *RobotRepository* fornece um fluxo contínuo de *frames*. O *MainViewModel* observa esse fluxo e, para cada novo *frame*, publica o *bitmap* mais recente em um *LiveData*. A *MainActivity* observa esse *LiveData* e atualiza a *ImageView* exibindo o *frame* atual. Essa abordagem permite exibir o vídeo em tempo real sem bloquear a interface, já que o recebimento e a decodificação dos *frames* ocorrem fora da *thread* principal.

A Figura 9 apresenta o fluxo completo do *streaming* de vídeo.

Figura 9 – Fluxo do *streaming* de vídeo



Fonte: Elaborada pelo autor.

3.8 Reconhecimento de Ambientes

Este módulo foi organizado em duas funções principais, disponíveis na aba de autonomia: "Mapear ambiente", que salva uma base local de descritores, e "Onde estou?", que consulta essa base e indica o ambiente mais provável.

Em ambos os casos, a entrada é um conjunto de quatro imagens, representando ângulos diferentes do mesmo ambiente (norte, leste, sul e oeste). No estado atual do projeto, essas imagens são selecionadas a partir da galeria do *smartphone*, mas o módulo foi desenvolvido para receber imagens em formato *Bitmap*, o que facilita trocar a origem no futuro. Por exemplo, em vez de converter imagens da galeria, pode-se utilizar diretamente *frames* do *streaming* de vídeo do robô, sem precisar reescrever a lógica de extração, persistência ou classificação.

Como o processamento envolve OpenCV e operações relativamente pesadas (conversão de imagem, extração de descritores, leitura/gravação em arquivo e classificação), essas etapas são executadas fora da *thread* principal, utilizando a *thread* de *computation* do Android, evitando travamentos e mantendo a interface responsiva.

A inicialização da biblioteca OpenCV ocorre na classe *RobotController*, que também inicializa os serviços usados pelo módulo de autonomia. Como essa classe está definida em *android:name* no *Manifest*, ela é instanciada pelo Android na inicialização do aplicativo, antes até mesmo da *MainActivity*.

3.8.1 Funcionalidade “Mapear Ambiente”

A função “Mapear ambiente” tem como objetivo criar a base local de reconhecimento. O usuário informa um nome para o ambiente e seleciona quatro imagens correspondentes aos ângulos norte, leste, sul e oeste (nessa exata ordem). Em seguida, o aplicativo extrai descritores de cada imagem e salva esses dados no armazenamento interno do *smartphone*, permitindo que o ambiente seja reconhecido futuramente.

O fluxo do aplicativo para o mapeamento foi implementado da seguinte forma:

1. O usuário toca no botão "Mapear ambiente"no *AutonomyControllerFragment*.
2. O aplicativo exibe uma *bottom sheet* solicitando o nome do ambiente. O nome não pode ser vazio e nem ser composto apenas por espaços em branco.
3. Ao confirmar, o aplicativo abre o seletor de imagens e o usuário escolhe 4 fotos.
4. O *MainViewModel* inicia o processamento das imagens usando o *MapEnvironmentUseCase* e, ao final, exibe um *feedback* informando que o ambiente foi mapeado.

Figura 10 – *Bottom sheet* para nome do ambiente

Fonte: Elaborada pelo autor.

Para a etapa de extração de descritores das imagens foi utilizado o descritor Local Binary Pattern (LBP) com histograma. A principal motivação é que o LBP gera um vetor

de tamanho fixo, o que facilita o uso de classificadores como o KNN. Além disso, é um descritor relativamente simples e rápido, o que ajuda na execução em *smartphones*.

Em comparação, extratores como Oriented FAST and Rotated BRIEF (ORB) e Scale-Invariant Feature Transform (SIFT) normalmente produzem um conjunto de descritores baseado em pontos de interesse, e a quantidade de pontos pode variar bastante de uma imagem para outra. Isso dificulta aplicar o KNN diretamente do jeito tradicional (matriz de treino com vetores de mesmo tamanho). Para usar ORB ou SIFT com um classificador como KNN, seria necessário transformar esse conjunto variável de descritores em uma representação fixa ou adotar outra estratégia de comparação. Por esse motivo, o LBP foi uma escolha mais direta para a primeira versão do reconhecimento de ambientes.

Nesta etapa, para cada imagem (N, L, S, O) é calculado um descritor do tipo LBP-histograma, seguindo a ideia apresentada por Ojala, Pietikäinen e Mäenpää (2002), em que a vizinhança de cada pixel é comparada com o valor do pixel central, gerando um código binário que depois é contabilizado em um histograma. A sequência abaixo mostra o processo completo para uma imagem. Ao final, o mesmo procedimento se repete para as quatro direções.

1. A imagem chega como *Bitmap* e é convertida para *Mat* para permitir o processamento com OpenCV.

2. A imagem é convertida para tons de cinza. Assim, cada pixel passa a ter um único valor de intensidade (0 a 255), que é o valor usado nas comparações do LBP.

3. A imagem é redimensionada para um tamanho fixo. Isso ajuda a manter consistência entre capturas e deixa o custo de processamento mais previsível.

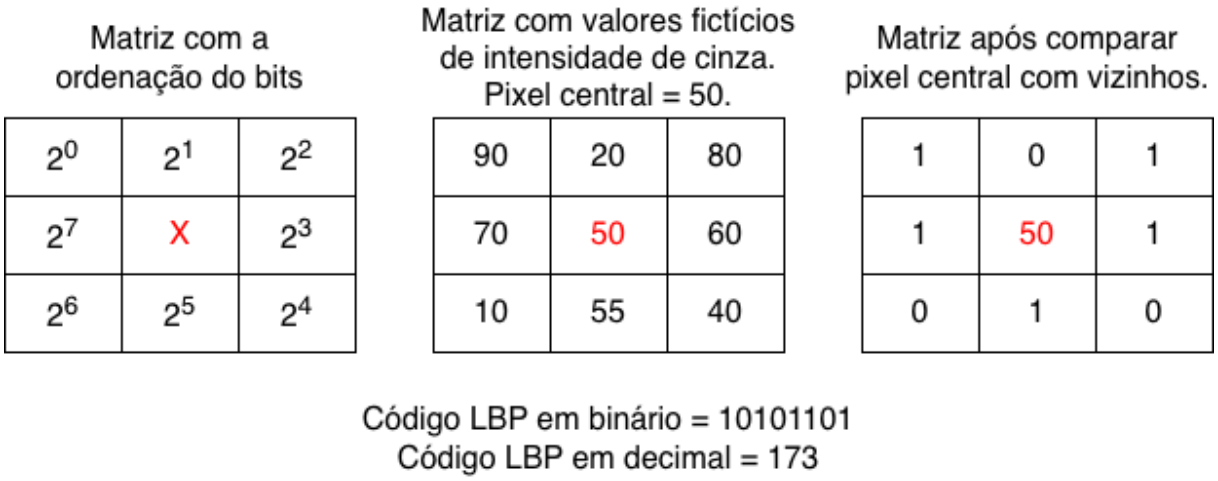
4. O LBP utilizado no projeto considera uma janela 3x3 (pixel central + 8 vizinhos). Por isso, a varredura percorre a imagem ignorando a primeira e a última linha/coluna, já que nelas não existem vizinhos suficientes para o cálculo.

5. Para cada pixel central, são comparados os 8 vizinhos com o valor do centro. Se o vizinho for maior ou igual ao centro, atribui-se 1. Caso contrário, 0. Assim, obtém-se 8 bits que, ao serem convertidos para a base decimal, nos darão o código LBP do pixel central. A figura 11 mostra um exemplo de cálculo de código LBP.

6. Para cada código calculado (0 a 255), o algoritmo incrementa a posição correspondente em um histograma. Ao final da varredura, o histograma representa a frequência de ocorrência de cada padrão local na imagem.

7. Para tornar o descritor comparável entre imagens, o histograma é normalizado dividindo cada posição pelo total de pixels processados. O resultado final vira um vetor de 256 valores com frequências entre 0 a 1, que é o descritor da imagem.

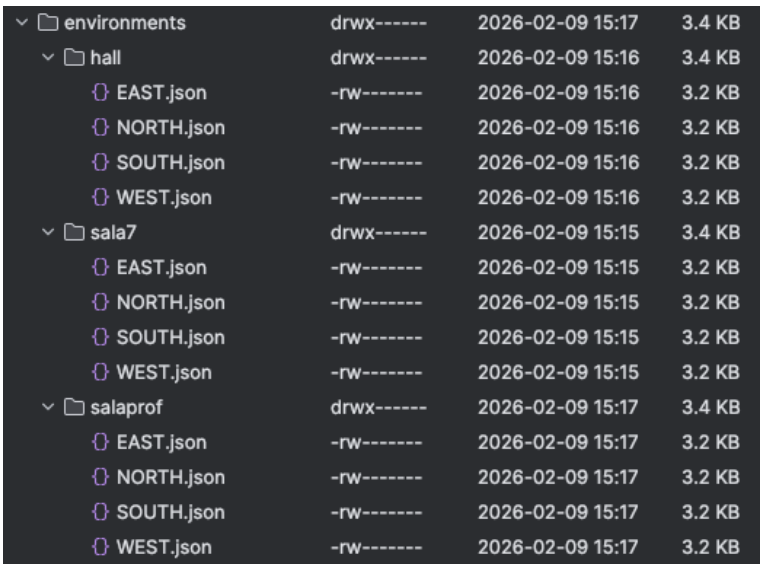
Figura 11 – Exemplo de cálculo do código LBP



Fonte: Elaborada pelo autor.

8. Cada vetor é salvo em um arquivo JSON separado (por exemplo, *north.json*, *east.json*, *south.json* e *west.json*) dentro da pasta do ambiente. Essa separação facilita a organização e mantém a comparação por ângulo no reconhecimento. A figura 12 mostra como fica a árvore de pastas após mapear alguns ambientes.

Figura 12 – Arquivos JSON com descritores



Fonte: Elaborada pelo autor.

Para facilitar possíveis testes em trabalhos futuros com diferentes algoritmos de descritores, o projeto utiliza a interface *DescriptorExtractor*. A lógica principal de Mapear ambiente não depende diretamente do LBP, mas sim dessa interface. Dessa forma, caso seja necessário testar outro extrator no futuro (desde que também gere um vetor fixo), a mudança pode ser feita concentrando a alteração na implementação do extrator e no

ponto de inicialização onde ele é escolhido (*AutonomyHelper*), sem precisar reescrever o fluxo do mapeamento.

3.8.2 Funcionalidade “Onde estou?”

A função “Onde estou?” utiliza a base local criada no mapeamento para indicar qual ambiente é mais provável para o conjunto atual de imagens. Diferente do "Mapear ambiente", aqui o objetivo não é salvar novos dados, mas sim comparar o ambiente atual com os ambientes previamente cadastrados e retornar apenas o melhor resultado.

O fluxo no aplicativo segue uma lógica parecida com a do mapeamento:

1. O usuário toca no botão "Onde estou?"no *AutonomyControllerFragment*.
2. O aplicativo abre o seletor de imagens e o usuário escolhe 4 fotos correspondentes aos ângulos norte, leste, sul e oeste (nessa exata ordem).
3. O *MainViewModel* inicia o processo de reconhecimento usando o *LocateEnvironmentUseCase*.
4. Ao final, o aplicativo exibe o nome do ambiente mais provável como *feedback* ao usuário.

O reconhecimento é realizado com o classificador KNN, utilizando a implementação disponível no módulo de *machine learning* do OpenCV. A ideia geral do KNN é: dado um exemplo novo (vetor de consulta), ele é comparado com exemplos já conhecidos (vetores de treino), e o resultado é decidido pelos k exemplos mais próximos.

No projeto, cada ambiente é representado por um vetor numérico fixo. Esse vetor é obtido a partir dos descritores LBP-histograma extraídos das quatro direções. Assim, o KNN consegue trabalhar comparando vetores do mesmo tamanho. A sequência abaixo mostra o processo executado.

1. O aplicativo consulta o *EnvironmentRepository* para obter a lista de ambientes mapeados (nomes das pastas existentes).
2. Para cada ambiente encontrado, o aplicativo carrega os quatro arquivos JSON salvos. Cada arquivo contém um vetor LBP normalizado de tamanho 256 (histograma).
3. O aplicativo concatena os vetores em sequência (norte, leste, sul e oeste), formando um único vetor por ambiente. Esse novo vetor de 1024 posições passa a ser a amostra do ambiente no conjunto de treino.

4. O KNN precisa que cada vetor de treino tenha um rótulo numérico. Então, para cada ambiente, o aplicativo associa um identificador (um número inteiro) e mantém um mapeamento entre esse identificador e o nome real do ambiente. Assim, o algoritmo trabalha com números, mas no final o aplicativo consegue converter de volta para o nome do ambiente.

5. Para usar o KNN do OpenCV, os vetores de cada ambiente precisam ser organizados em matrizes *Mat*. Assim, o aplicativo monta uma matriz de treino onde cada linha é um ambiente e as colunas são os valores do vetor (por exemplo, 1024 valores). Além disso, monta uma segunda matriz apenas com os rótulos, com um rótulos por linha, na mesma ordem das linhas do treino.

6. Para o ambiente atual, o aplicativo extrai os descritores das quatro imagens selecionadas usando o mesmo *DescriptorExtractor* e também concatena os quatro vetores, formando um único vetor de consulta com o mesmo tamanho do vetor de treino.

7. O valor de k é configurável no aplicativo para facilitar testes. Ainda assim, existe um limite prático em que k não pode ser maior do que a quantidade de ambientes cadastrados (quantidade de linhas no treino). Por isso, o aplicativo garante que k sempre será no mínimo 1 e no máximo o número de ambientes existentes.

8. No KNN, o termo "treinar" diz respeito a armazenar a base (vetores de treino e seus rótulos). Depois disso, o KNN compara o vetor de consulta com todos os vetores de treino, calculando uma distância entre eles. Em seguida, seleciona os k ambientes mais próximos. O resultado final é o rótulo que aparece com mais frequência entre esses k vizinhos. Em caso de empate, o algoritmo tende a privilegiar o vizinho mais próximo. No caso de $k = 1$, a decisão é simplesmente o rótulo do ambiente mais próximo. Essa classificação é realizada pela função *findNearest* da classe *KNearest* do OpenCV.

9. Ao final, o OpenCV retorna um rótulo inteiro. O aplicativo usa o mapeamento criado no passo 4 para converter esse valor de volta no nome real do ambiente e exibe apenas o melhor ambiente para o usuário.

Como o reconhecimento depende de imagens, fatores como iluminação, objetos temporários e mudanças no enquadramento podem influenciar o resultado. Além disso, no projeto foi adotada a convenção de trabalhar com quatro direções (norte, leste, sul e oeste). Por isso, é importante que as imagens usadas no "Onde estou?" sigam a mesma ordem usada no mapeamento. Caso as direções sejam trocadas (por exemplo, usar uma imagem "norte" no lugar de "leste"), a comparação tende a perder consistência e o reconhecimento pode piorar.

4 VALIDAÇÕES E RESULTADOS

Este capítulo apresenta os testes realizados no aplicativo e os resultados obtidos em cada parte do sistema. O objetivo foi validar se a comunicação com o robô está funcionando como esperado e se as funcionalidades principais do aplicativo respondem corretamente aos comandos enviados.

Os testes foram separados por funcionalidade: conexão com o robô, movimentos do robô, movimentos da câmera, movimentos e informações da antena, comandos de sistema e bússola, *streaming* e informações de vídeo e reconhecimento de ambientes. Essa organização garante que tudo o que foi implementado seja validado de maneira independente e facilita a identificação de possíveis problemas.

Para comprovar os testes, foram usados *logs* dos dois lados. No aplicativo, os *logs* foram coletados pelo *Logcat* do Android Studio. No robô, os *logs* foram acompanhados através de uma conexão Secure Shell (SSH) que, enquanto executa o código do robô, imprime os comandos recebidos e respostas enviadas no terminal. Assim, em cada teste é possível observar o comando saindo do aplicativo e chegando no robô, depois a resposta do robô para o aplicativo.

No caso do reconhecimento de ambientes, os resultados são apresentados com base no ambiente esperado e no ambiente retornado pelo método (Local Binary Pattern (LBP) + K-Nearest Neighbors (KNN)), destacando os acertos e os pontos em que o reconhecimento falhou ou ficou sensível a mudanças no ambiente.

Todos os testes foram realizados no mesmo ambiente, cujas características são:

- Modelo do *smartphone*: Samsung Galaxy A56.
- Versão de Android do *smartphone*: 16 (API 36).
- Conexão de rede do *smartphone*: 5G da operadora de internet Tim.
- Conexão de rede do robô: cabeada através da estrutura do Instituto Federal de Santa Catarina (IFSC) campus São José.

Não há configurações externas ao aplicativo e ao robô necessárias para realização dos testes.

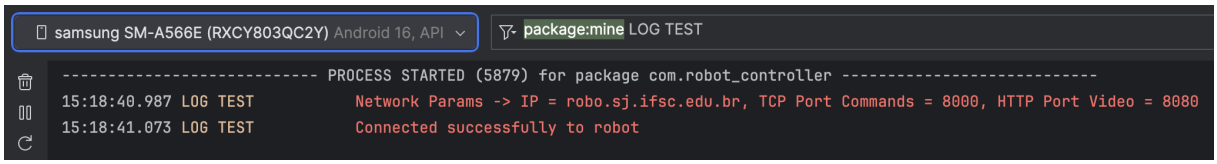
4.1 Conexão com o Robô

Este teste teve como objetivo verificar se o aplicativo consegue estabelecer conexão Transmission Control Protocol (TCP) com o robô. Para isso, o robô permaneceu ligado e com o servidor TCP JavaScript Object Notation (JSON) ativo na porta de comandos 8000. O procedimento realizado foi:

1. Abrir o aplicativo no *smartphone*.
2. Acessar as configurações de rede pelo ícone na *MaterialToolbar*.
3. Informar o domínio do robô e as portas TCP para comandos e Hypertext Transfer Protocol (HTTP) para vídeo.
4. Clicar no botão *Salvar* da *bottom sheet*.
5. Acompanhar o *Logcat* no Android Studio e, em paralelo, o terminal via SSH no robô.

Como evidência do teste, a figura 13 apresenta o *Logcat* com a impressão dos parâmetros de rede configurados e a mensagem indicando conexão bem sucedida. Já a figura 14 mostra o terminal conectado ao robô registrando o evento de "Cliente conectado", confirmando que o servidor TCP aceitou a conexão. Com isso, o resultado observado foi que o aplicativo conseguiu estabelecer a conexão TCP com o robô.

Figura 13 – *Logs* de conexão pelo lado do aplicativo



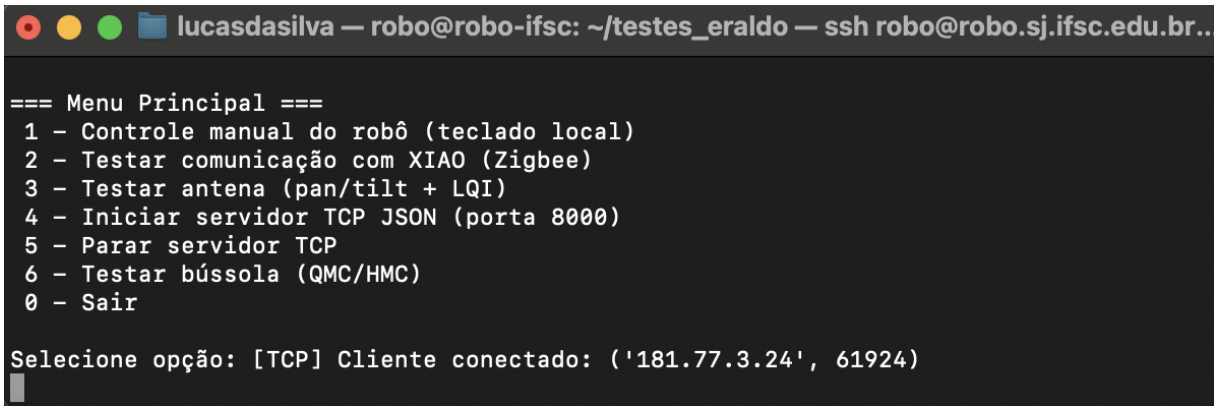
Fonte: Elaborada pelo autor.

Por fim, também é possível observar que o estabelecimento da conexão levou cerca de 80 milissegundos. Ao longo dos demais testes, a conexão mostrou-se estável.

4.2 Movimentos do Robô

Este teste teve como objetivo validar o controle de movimentação do robô pelo aplicativo, verificando se os comandos de tração enviados via TCP são recebidos corretamente no robô e se a resposta de sucesso é retornada ao aplicativo. Paralelamente, também foi testada a variação de velocidade da movimentação. Para isso, o robô permaneceu ligado, com o servidor TCP JSON ativo na porta de comandos 8000 e com a conexão TCP já estabelecida. O procedimento realizado foi:

Figura 14 – *Logs* de conexão pelo lado do robô



Fonte: Elaborada pelo autor.

1. No aplicativo, acessar a aba de controle de tração *TractionControllerFragment*.
2. Enviar o comando de movimento para frente com velocidade igual a 300.
3. Enviar o comando de movimento para direita com velocidade igual a 300.
4. Enviar o comando de movimento para trás (ré) com velocidade igual a 510.
5. Enviar o comando de movimento para esquerda com velocidade igual a 752.
6. Enviar o comando de parada de movimento.
7. Durante cada comando, acompanhar o *Logcat* no Android Studio e, em paralelo, o terminal via SSH no robô.

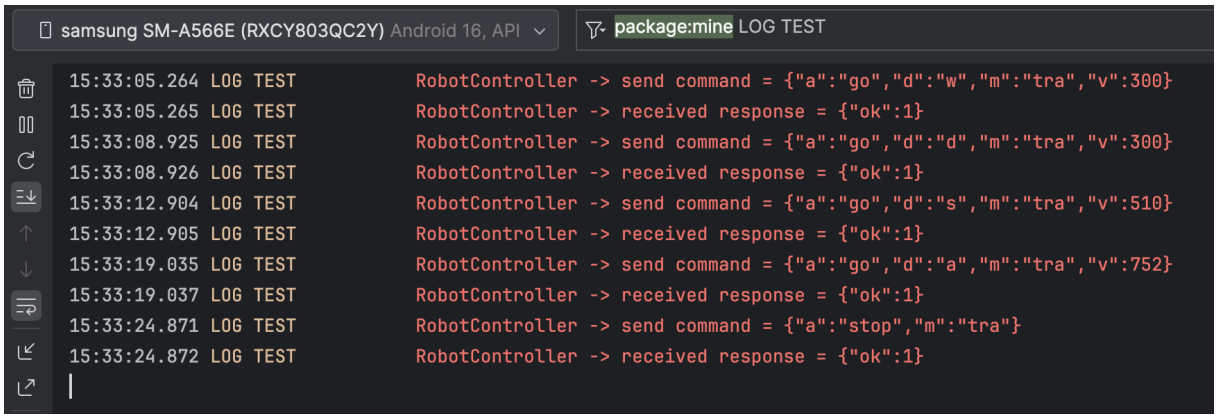
Como evidência do teste, a figura 15 apresenta o *Logcat* do aplicativo registrando o envio dos comandos de tração e o recebimento das respostas de sucesso para cada comando. Já a figura 16 mostra o terminal conectado ao robô confirmando o recebimento (linhas RX) dos mesmos comandos na ordem executada, inclusive o de parada de movimento. Também é possível observar que o robô imprime a velocidade de movimento recebida no último comando, então envia a confirmação (linhas TX) do comando.

Com isso, o resultado observado foi que o aplicativo conseguiu controlar a movimentação do robô conforme esperado, incluindo variação de velocidade e comando de parada.

4.3 Movimentos da Câmera

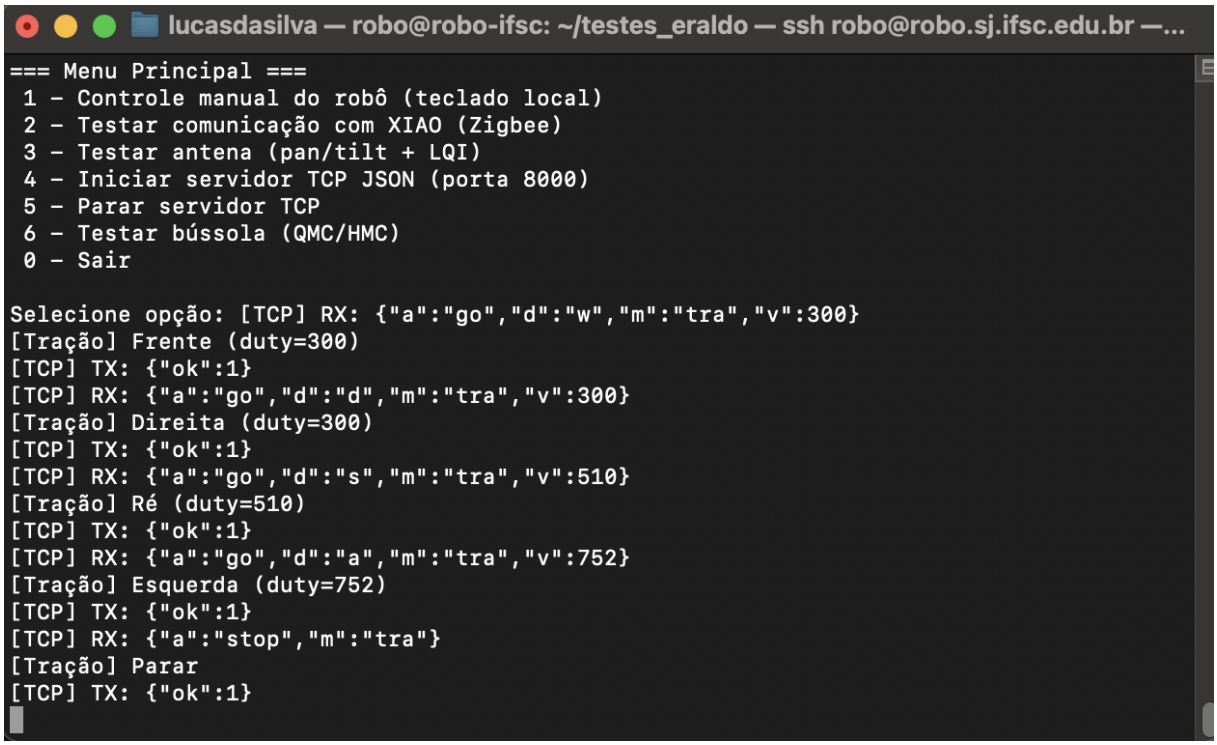
Estes testes tiveram como objetivo validar o controle da câmera do robô pelo aplicativo, verificando se os comandos de movimento da câmera enviados via TCP são recebidos corretamente no robô e se a resposta de sucesso é retornada ao aplicativo. Como

Figura 15 – *Logs* de movimento do robô pelo lado do aplicativo



Fonte: Elaborada pelo autor.

Figura 16 – *Logs* de movimento do robô pelo lado do robô



Fonte: Elaborada pelo autor.

a câmera possui dois modos de controle no protocolo (contínuo e absoluto), os testes foram divididos nas duas partes apresentadas a seguir.

Para realizar estes testes, o robô permaneceu ligado, com o servidor TCP JSON ativo na porta de comandos 8000 e com a conexão TCP já estabelecida.

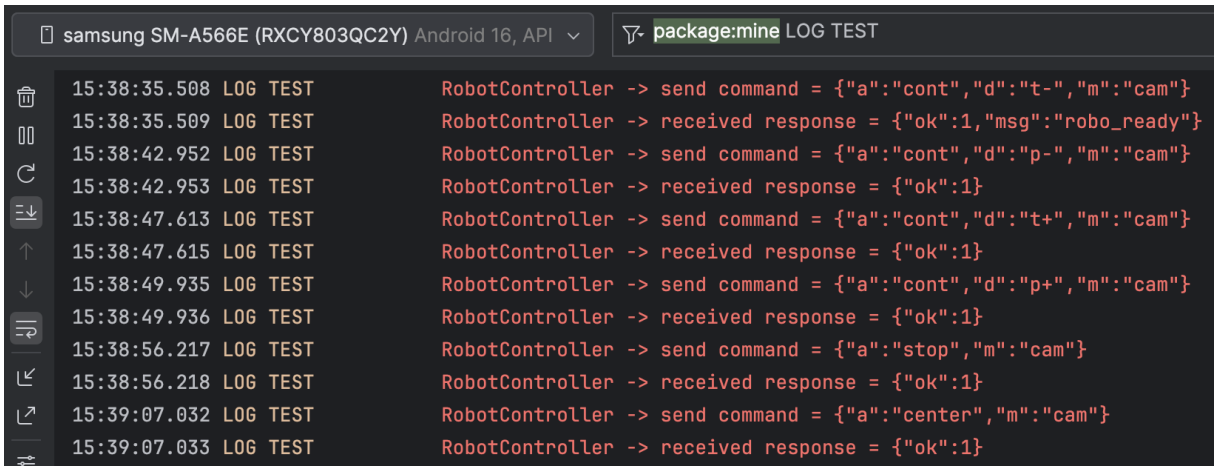
4.3.1 Movimentação Contínua

O procedimento realizado foi:

1. No aplicativo, acessar a aba de controle da câmera *CameraControllerFragment*.
2. Enviar o comando de movimentação contínua para cima.
3. Enviar o comando de movimentação contínua para direita.
4. Enviar o comando de movimentação contínua para baixo.
5. Enviar o comando de movimentação contínua para esquerda.
6. Enviar o comando de parada de movimento.
7. Durante cada comando, acompanhar o *Logcat* no Android Studio e, em paralelo, o terminal via SSH no robô.

Como evidência do teste, a figura 17 apresenta o *Logcat* do aplicativo registrando o envio dos comandos de movimentação e parada da câmera e o recebimento das respostas de sucesso para cada comando. Já a figura 18 mostra o terminal conectado ao robô confirmando o recebimento (linhas RX) desses comandos e a execução do movimento contínuo em cada direção, além da parada de movimento. Em todos os casos, o robô responde aos comandos com sucesso (linhas TX).

Figura 17 – *Logs* de movimento contínuo da câmera pelo lado do aplicativo



Fonte: Elaborada pelo autor.

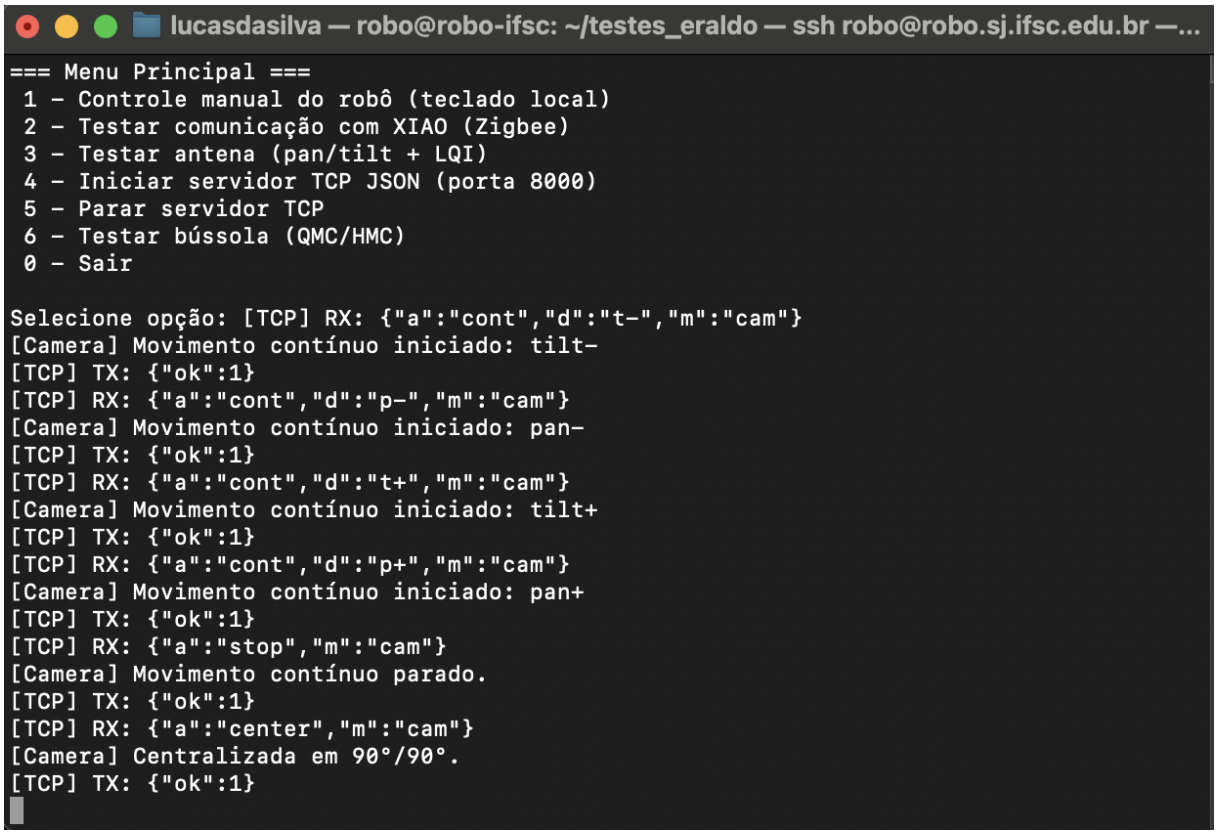
Com isso, o resultado observado foi que o aplicativo conseguiu controlar a movimentação contínua da câmera do robô conforme esperado, incluindo o comando de parada.

4.3.2 Movimentação Absoluta

O procedimento realizado foi:

1. No aplicativo, acessar a aba de controle da câmera *CameraControllerFragment*.

Figura 18 – *Logs* de movimento contínuo da câmera pelo lado do robô



Fonte: Elaborada pelo autor.

2. Digitar o valor para *pan* respeitando os limites indicados no aplicativo (0 a 180 graus). O valor usado no teste foi 30 graus.

3. Digitar o valor para *tilt* respeitando os limites indicados no aplicativo (45 a 135 graus). O valor usado no teste foi 60 graus.

4. Clicar no botão *Ir para*.

5. Repetir o teste digitando novos valores para *pan* e *tilt*. Os valores usados no novo teste foram: *pan* igual a 180 graus e *tilt* igual a 90 graus.

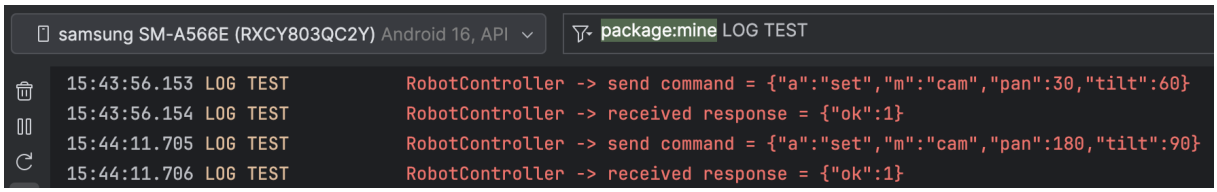
6. Clicar no botão *Centralizar*.

7. Durante cada comando, acompanhar o *Logcat* no Android Studio e, em paralelo, o terminal via SSH no robô.

Como evidência do teste, a figura 19 apresenta o *Logcat* do aplicativo registrando os comandos contendo os valores exatos de *pan* e *tilt*, além do recebimento da resposta de cada comando. Já a figura 20 mostra o terminal conectado ao robô confirmando o recebimento (linhas RX) dos mesmos comandos e respondendo com sucesso (linhas TX).

No caso da validação do botão *Centralizar*, a figura 17 apresenta o *Logcat* do aplicativo registrando o comando para centralizar a câmera (90 graus para *pan* e *tilt*), além do recebimento da resposta ao comando. Já a figura 18 mostra o terminal conectado ao robô confirmando o recebimento (linha RX) do mesmo comando e respondendo com sucesso (linha TX).

Figura 19 – *Logs* de movimento absoluto da câmera pelo lado do aplicativo



Fonte: Elaborada pelo autor.

Figura 20 – *Logs* de movimento absoluto da câmera pelo lado do robô

Fonte: Elaborada pelo autor.

Com isso, o resultado observado foi que o aplicativo conseguiu controlar a movimentação absoluta da câmera do robô conforme esperado, incluindo o comando de centralizar.

4.4 Movimentos, Varredura e Informações da Antena

Estes testes tiveram como objetivo validar o controle manual da posição da antena, a varredura completa para identificar o melhor posicionamento e a leitura de informações da antena do robô pelo aplicativo, verificando se os comandos TCP são recebidos corretamente no robô e se a resposta de sucesso é retornada ao aplicativo. A validação foi dividida em duas partes: movimentação e obtenção de informações.

Para realizar estes testes, o robô permaneceu ligado, com o servidor TCP JSON ativo na porta de comandos 8000 e com a conexão TCP já estabelecida.

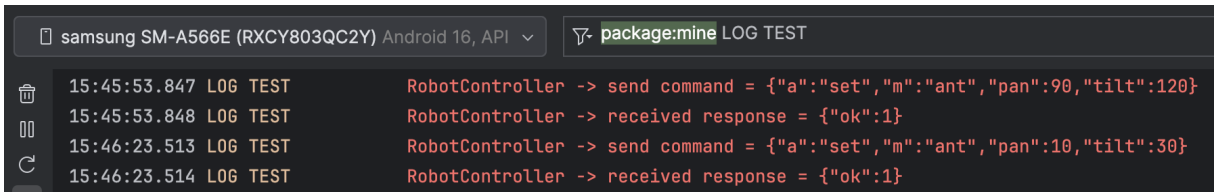
4.4.1 Movimentação da Antena

O procedimento realizado foi:

1. No aplicativo, acessar a aba de controle da antena *AntennaControllerFragment*.
2. Digitar o valor para *pan* respeitando os limites indicados no aplicativo (0 a 180 graus). O valor usado no teste foi 90 graus.
3. Digitar o valor para *tilt* respeitando os limites indicados no aplicativo (0 a 180 graus). O valor usado no teste foi 120 graus.
4. Clicar no botão *Ir para*.
5. Repetir o teste digitando novos valores para *pan* e *tilt*. Os valores usados no novo teste foram: *pan* igual a 10 graus e *tilt* igual a 30 graus.
6. Durante cada comando, acompanhar o *Logcat* no Android Studio e, em paralelo, o terminal via SSH no robô.

Como evidência do teste, a figura 21 apresenta o *Logcat* do aplicativo registrando os comandos contendo os valores exatos de *pan* e *tilt*, além do recebimento da resposta de cada comando. Já a figura 22 mostra o terminal conectado ao robô confirmando o recebimento (linhas RX) dos mesmos comandos e respondendo com sucesso (linhas TX).

Figura 21 – *Logs* de movimento absoluto da antena pelo lado do aplicativo



Fonte: Elaborada pelo autor.

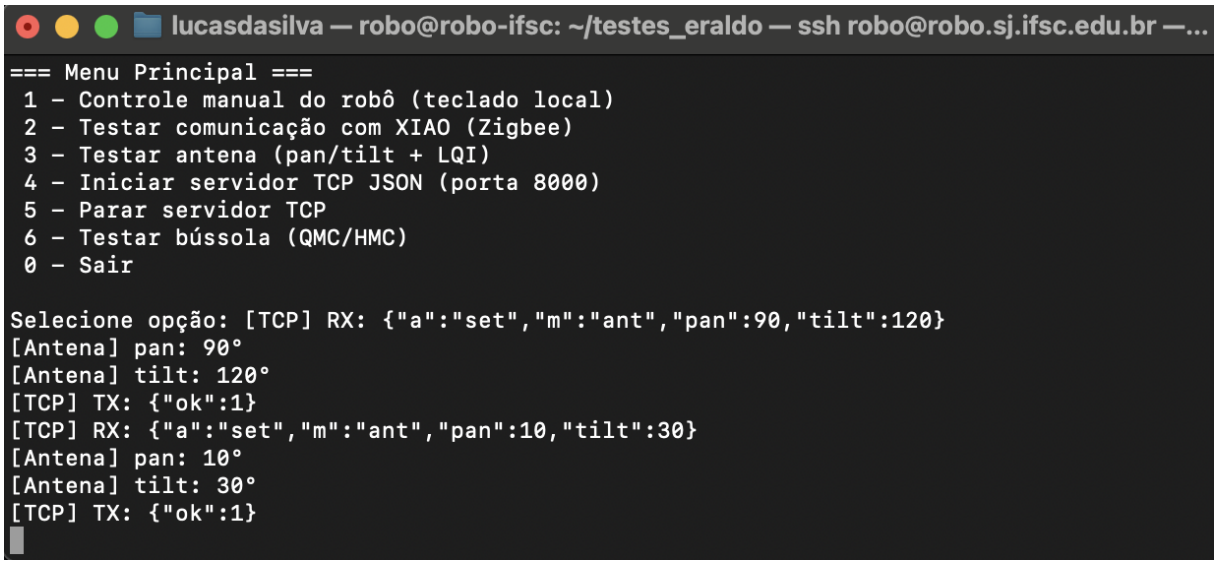
Com isso, o resultado observado foi que o aplicativo conseguiu controlar a antena por posicionamento absoluto conforme esperado.

4.4.2 Varredura Completa e Obtenção de Informações da Antena

Neste teste, espera-se validar o comando de varredura completa da antena. Ao receber o comando, o robô deve movimentar a antena procurando o melhor posicionamento para comunicação. Então, ao encontrar esta posição, deve retornar ao aplicativo os valores de *pan*, *tilt* e LQI da antena.

O procedimento realizado foi:

Figura 22 – *Logs* de movimento absoluto da antena pelo lado do robô

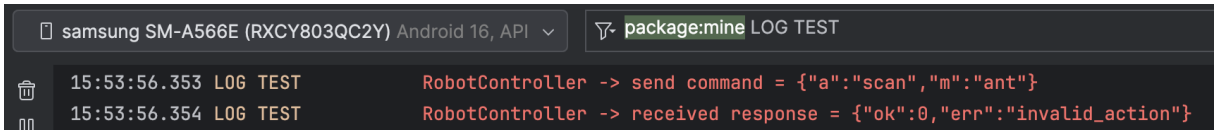


Fonte: Elaborada pelo autor.

1. No aplicativo, acessar a aba de controle da antena *AntennaControllerFragment*.
2. Clicar no botão *Scan antena*.
3. Durante o comando, acompanhar o *Logcat* no Android Studio e, em paralelo, o terminal via SSH no robô.

Como evidência do teste, a figura 23 apresenta o *Logcat* do aplicativo registrando o comando enviado, além do recebimento da resposta de erro do robô. Já a figura 24 mostra o terminal conectado ao robô confirmando o recebimento (linha RX) do mesmo comando e respondendo com erro (linha TX).

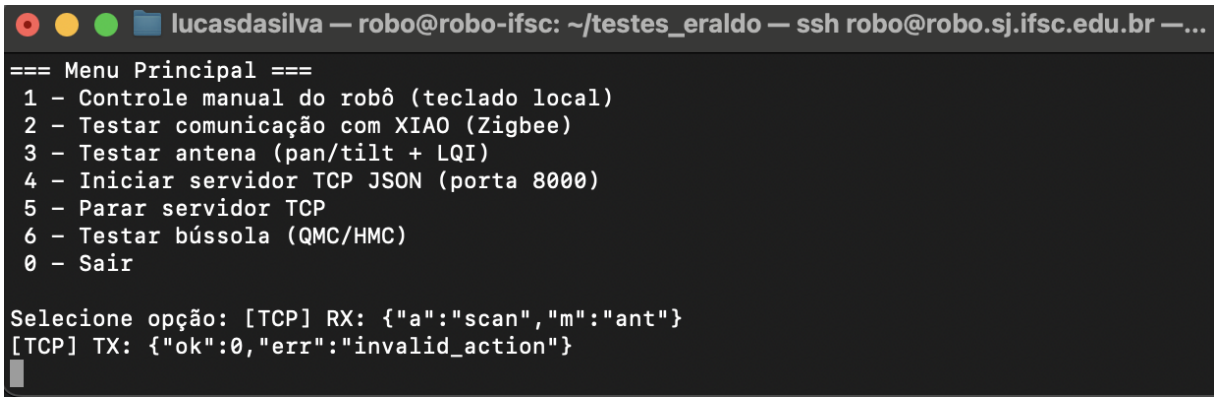
Figura 23 – *Logs* de comando *scan* da antena pelo lado do aplicativo



Fonte: Elaborada pelo autor.

Com isso, o resultado observado foi que o aplicativo não conseguiu controlar a varredura da antena do robô. O erro retornado pelo robô ao aplicativo é "*invalid action*". Embora conste o suporte a este comando na documentação do protocolo JSON, o robô pode não tê-lo implementado ou então possui falha na implementação deste comando.

Figura 24 – *Logs* de comando *scan* da antena pelo lado do robô



Fonte: Elaborada pelo autor.

4.5 Comandos de Sistema e Bússola

Estes testes tiveram como objetivo validar comandos dos módulos de sistema e bússola do robô através do aplicativo, verificando se os comandos TCP são recebidos corretamente no robô e se a resposta de sucesso é retornada ao aplicativo. A validação foi dividida em três partes: parada de emergência, telemetria do sistema e leitura de bússola.

Para realizar estes testes, o robô permaneceu ligado, com o servidor TCP JSON ativo na porta de comandos 8000 e com a conexão TCP já estabelecida.

4.5.1 Parada de Emergência

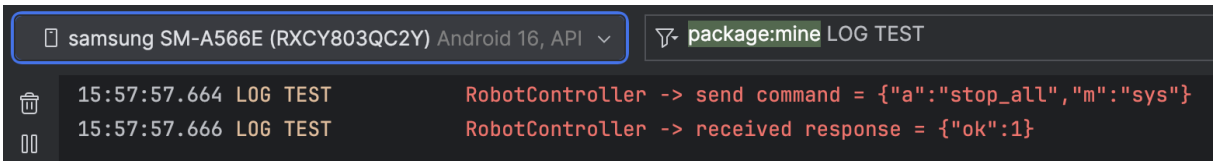
Este teste teve como objetivo verificar se o comando de parada de emergência consegue interromper as ações em execução no robô. O procedimento realizado foi:

1. No aplicativo, acessar a aba de controle de sistema e bússola *SystemControllerFragment*.
2. Clicar no botão *Parar tudo*.
3. Acompanhar o *Logcat* no Android Studio e, em paralelo, o terminal via SSH no robô.

Como evidência do teste, a figura 25 apresenta o *Logcat* do aplicativo registrando o envio do comando de parada e o recebimento da resposta de sucesso. Já a figura 26 mostra o terminal conectado ao robô confirmando o recebimento (linha RX) do comando e respondendo com sucesso (linha TX). Além disso, o terminal mostra a interrupção de funções como tração, câmera e *streaming* de vídeo.

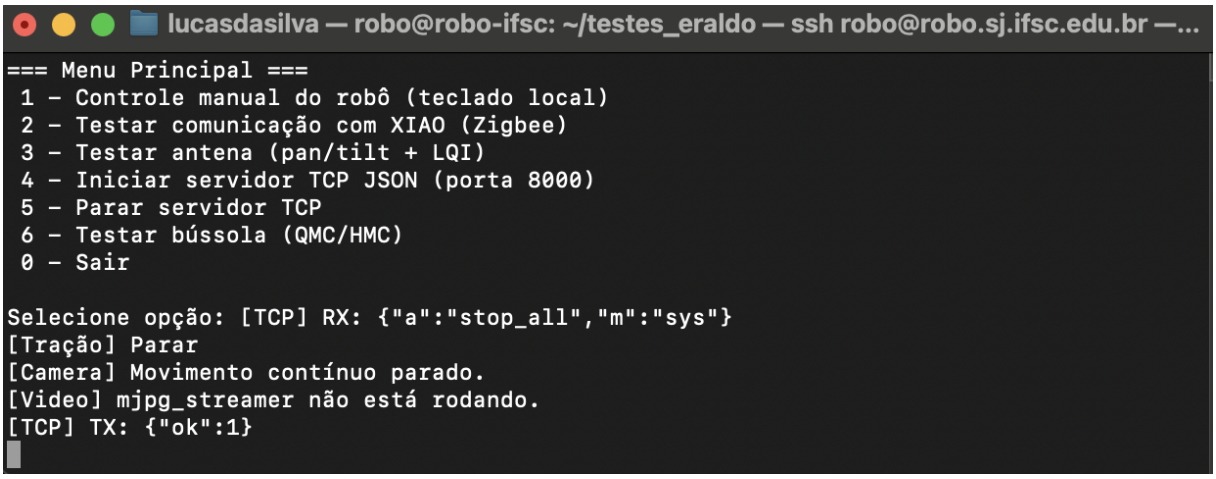
Com isso, o resultado observado foi que o aplicativo conseguiu controlar o comando de parada de emergência do robô conforme esperado.

Figura 25 – *Logs* de comando *stop all* do sistema pelo lado do aplicativo



Fonte: Elaborada pelo autor.

Figura 26 – *Logs* de comando *stop all* do sistema pelo lado do robô



Fonte: Elaborada pelo autor.

4.5.2 Telemetria do Sistema

Este teste teve como objetivo validar a solicitação de telemetria do robô, verificando se o aplicativo recebe uma resposta com dados do robô e se essa resposta é consistente com o que prevê o protocolo. O procedimento realizado foi:

1. No aplicativo, acessar a aba de controle de sistema e bússola *SystemControllerFragment*.
2. Clicar no botão *Bússola*. Não há botão exclusivo para telemetria na interface, portanto foi utilizado o botão *Bússola* temporariamente alterado para obter as informações de telemetria.
3. Acompanhar o *Logcat* no Android Studio e, em paralelo, o terminal via SSH no robô.

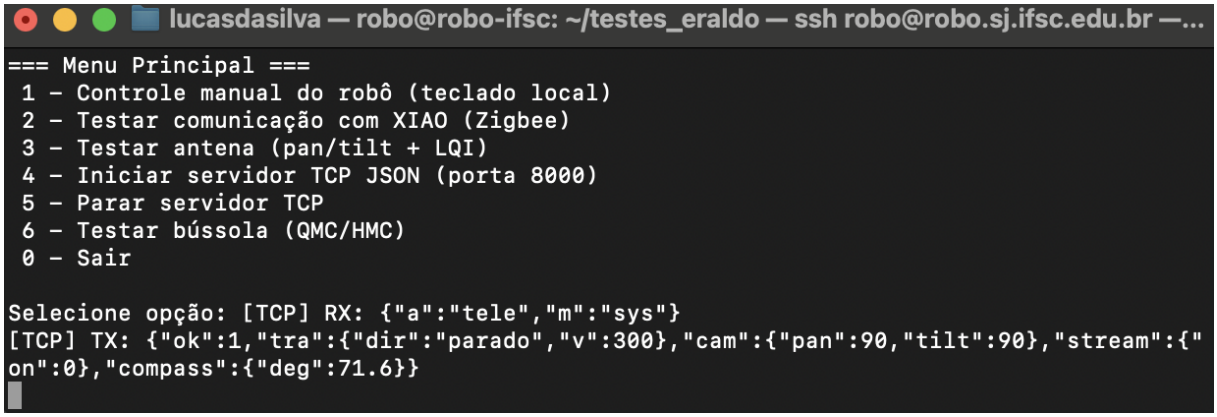
Como evidência do teste, a figura 27 mostra o *Logcat* do aplicativo registrando o envio do comando de telemetria e o recebimento da resposta de sucesso. Já a figura 28 mostra o terminal conectado ao robô confirmando o recebimento (linha RX) do comando e respondendo com sucesso (linha TX).

Com isso, o resultado observado foi que o aplicativo conseguiu controlar o comando de telemetria do robô conforme esperado. Entretanto, durante o teste, nem todas as

Figura 27 – *Logs* de comando de telemetria do sistema pelo lado do aplicativo

Fonte: Elaborada pelo autor.

Figura 28 – *Logs* de comando de telemetria do sistema pelo lado do robô



Fonte: Elaborada pelo autor.

informações previstas na especificação do protocolo apareceram na resposta. O protocolo prevê o envio de dados como *pan/tilt* da antena, Link Quality Indicator (LQI) da antena e nível da bateria do robô. Porém, esses campos não foram retornados pelo robô. Por outro lado, outros dados foram enviados e registrados, o que confirma o funcionamento básico do comando, mas indica que a estrutura completa de telemetria ainda precisa ser ajustada no lado do robô.

4.5.3 Leitura da Bússola

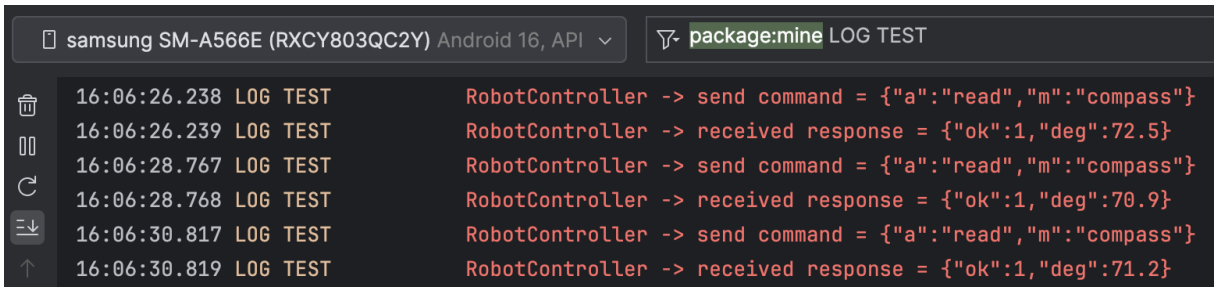
Este teste teve como objetivo validar o comando de leitura da bússola, verificando se o robô retorna o ângulo em graus e se o aplicativo consegue receber e registrar esse valor. O procedimento realizado foi:

1. No aplicativo, acessar a aba de controle de sistema e bússola *SystemControllerFragment*.
2. Clicar no botão *Bússola*.
3. Acompanhar o *Logcat* no Android Studio e, em paralelo, o terminal via SSH no robô.

Como evidência do teste, a figura 29 mostra o *Logcat* do aplicativo registrando o envio do comando de leitura da bússola e o recebimento da resposta de sucesso contendo o valor em graus. Já a figura 30 mostra o terminal conectado ao robô confirmando o

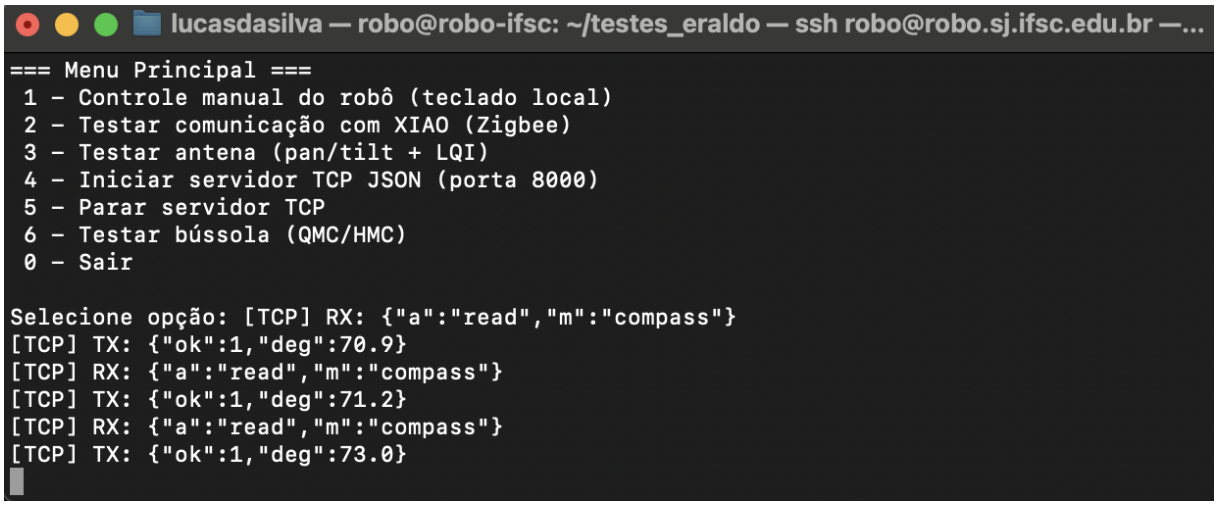
recebimento (linha RX) do comando e respondendo com sucesso (linha TX) o ângulo lido na bússola.

Figura 29 – *Logs* de comando de bússola pelo lado do aplicativo



Fonte: Elaborada pelo autor.

Figura 30 – *Logs* de comando de bússola pelo lado do robô



Fonte: Elaborada pelo autor.

Com isso, o resultado observado foi que o aplicativo conseguiu obter a leitura da bússola do robô conforme esperado. Também nota-se que realizando leituras em sequência é possível perceber pequenas variações entre um valor e outro, o que é esperado por se tratar de uma medida de sensor, sujeita a ruído e pequenas oscilações.

4.6 *Streaming* e Informações de Vídeo

Estes testes tiveram como objetivo validar as funcionalidades relacionadas ao vídeo do robô, incluindo iniciar/parar o serviço de *streaming* e consultar o status do *streaming*, verificando se os comandos TCP são recebidos corretamente no robô e se a resposta de sucesso é retornada ao aplicativo. Como são comandos com comportamentos diferentes (no primeiro caso existe também uma requisição HTTP para consumir o Motion Joint Photographic Experts Group (MJPEG)), o teste foi dividido em duas partes.

Para realizar estes testes, o robô permaneceu ligado, com o servidor TCP JSON ativo na porta de comandos 8000 e com a conexão TCP já estabelecida.

4.6.1 Iniciar e Parar o *Streaming* de Vídeo

Este teste teve como objetivo verificar se o aplicativo consegue iniciar o serviço de *streaming* MJPEG no robô e, em seguida, consumir o fluxo de vídeo via HTTP. Também foi validado o comando de parada do serviço. O procedimento realizado foi:

1. Clicar no botão de *Play* do vídeo na tela principal.
2. Aguardar a inicialização do serviço no robô e observar (no *Logcat*) a abertura da conexão HTTP para a Uniform Resource Locator (URL) de *streaming*.
3. Assistir ao vídeo por alguns minutos.
4. Acessar a aba de controle de sistema e bússola *SystemControllerFragment*.
5. Clicar no botão *Parar vídeo*.
6. Durante cada comando, acompanhar o *Logcat* no Android Studio e, em paralelo, o terminal via SSH no robô.

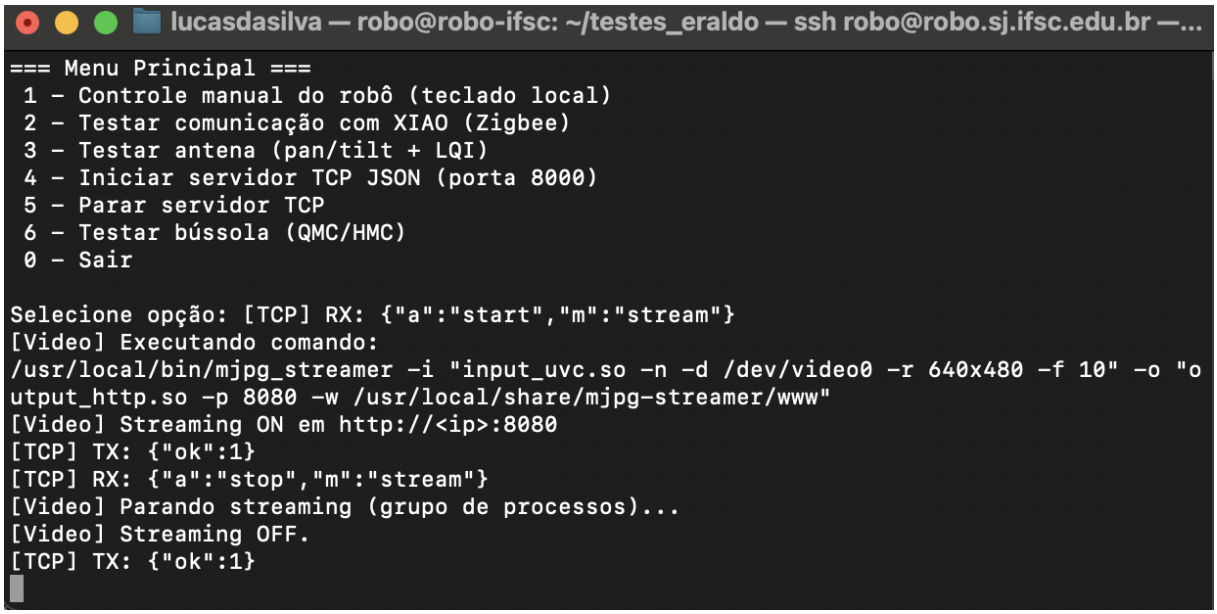
Como evidência do teste, a figura 31 apresenta o *Logcat* do aplicativo registrando o envio do comando de inicialização do serviço de *streaming* MJPEG e o recebimento da resposta de sucesso. Na sequência, o *Logcat* também registra a conexão HTTP com resposta *200 OK* para a URL, indicando que o serviço MJPEG estava disponível e aceitou a requisição. Já a figura 32 mostra o terminal conectado ao robô confirmando o recebimento (linhas RX) dos comandos, a inicialização do serviço de *streaming* e a indicação de que ele está *online*. Por fim, também é possível observar o recebimento do comando de parada de vídeo e a finalização do serviço.

Figura 31 – *Logs* de comando de *streaming* de vídeo pelo lado do aplicativo

Fonte: Elaborada pelo autor.

Com isso, o resultado observado foi que o aplicativo conseguiu iniciar e parar o serviço de vídeo corretamente, e que a requisição HTTP para o *streaming* MJPEG foi atendida com sucesso.

Figura 32 – *Logs* de comando de *streaming* de vídeo pelo lado do robô



Fonte: Elaborada pelo autor.

4.6.2 *Status* do *Streaming* de Vídeo

Este teste teve como objetivo validar se o aplicativo consegue consultar o *status* do *streaming*, verificando se o robô retorna informações de configuração do serviço de vídeo quando solicitado.

Durante a execução deste teste, o aplicativo estava exibindo o *streaming* de vídeo. Porém, isso não é um pré-requisito para o comando e apenas influencia no parâmetro *"on"* retornado pelo robô. O procedimento realizado foi:

1. No aplicativo, acessar a aba de controle de sistema e bússola *SystemControllerFragment*.
2. Clicar no botão *Bússola*. Não há botão exclusivo para *status* do *streaming* de vídeo na interface, portanto foi utilizado o botão *Bússola* temporariamente alterado para obter as informações de *status* do *streaming*.
3. Acompanhar o *Logcat* no Android Studio e, em paralelo, o terminal via SSH no robô.

Como evidência do teste, a figura 33 apresenta o *Logcat* do aplicativo registrando o envio do comando de *status* do *streaming* e o recebimento da resposta de sucesso. Já a figura 34 mostra o terminal conectado ao robô confirmando o recebimento (linha RX) do comando e respondendo com sucesso (linha TX) o *status* atual.

Com isso, o resultado observado foi que o aplicativo conseguiu obter o *status* do *streaming* de vídeo conforme esperado. Também nota-se que o robô retorna uma infor-

Figura 33 – *Logs* de comando de *status* de vídeo pelo lado do aplicativo

Fonte: Elaborada pelo autor.

Figura 34 – *Logs* de comando de *status* de vídeo pelo lado do robô

Fonte: Elaborada pelo autor.

mação a mais do que está previsto no protocolo. O campo *"www"* não está documentado no protocolo.

4.7 Reconhecimento de Ambientes

Este conjunto de testes teve como objetivo validar o reconhecimento de ambientes implementado no aplicativo, utilizando LBP para extração de descritores e KNN para classificação. Para os testes, foram mapeados três ambientes: dois quartos e uma sala de televisão. O mapeamento de cada ambiente foi feito com quatro fotos, uma para cada direção (norte, leste, sul e oeste). No momento do *"Onde estou?"*, foram tiradas novas fotos, também nas quatro direções, para simular uma consulta real.

Um ponto importante neste teste é o valor de k utilizado no KNN. Como neste trabalho foi utilizado apenas um conjunto de amostras por ambiente (ou seja, um vetor de treino por ambiente), o valor padrão adotado foi $k = 1$. Nesse caso, a decisão do KNN fica equivalente a escolher o ambiente cujo vetor de treino está mais próximo do vetor de consulta. Já o valor $k = 3$ foi utilizado em alguns casos apenas por didática e comparação, mas, como existe apenas um vetor por classe, não se espera um ganho de assertividade. Por isso, $k = 1$ foi mantido como referência principal nos resultados, e $k = 3$ aparece apenas como apoio quando necessário.

Os testes foram organizados em três partes: funcionamento no caso ideal, sensibilidade a mudanças no ambiente e importância da ordem das direções (N/L/S/O).

Para realizar estes testes, os três ambientes foram previamente mapeados através da função *"Mapear ambiente"*. Nesta etapa de mapeamento, as fotos foram tiradas com o ambiente iluminado apenas por luz artificial branca, mantendo essa condição igual para todos os ambientes.

4.7.1 Caso Ideal

Este teste teve como objetivo validar se o reconhecimento funciona quando o ambiente está nas condições mais próximas do mapeamento. Ou seja, o ambiente é o mesmo, sem alterações de iluminação, objetos ou ângulo da captura das imagens. Ainda, as novas fotos foram inseridas no teste na mesma ordem de direção (N/L/S/O) do mapeamento. O procedimento realizado foi:

1. Executar o "Onde estou?"e submeter novas fotos (N/L/S/O) do mesmo ambiente.
2. Anotar o resultado apresentado pelo aplicativo.
3. Repetir o processo três vezes para os três ambientes.

Como evidência do teste, a tabela 1 resume os resultados obtidos para os três ambientes, utilizando $k = 1$. Além disso, a figura 35 apresenta o *Logcat* do aplicativo registrando o reconhecimento por meio da mensagem "Provável ambiente: [nome-ambiente]". Na figura, é possível observar que, ao executar o *"Onde estou?"* com novas fotos do mesmo ambiente, o aplicativo retornou os nomes correspondente aos ambientes mapeados, confirmando o funcionamento do fluxo de reconhecimento.

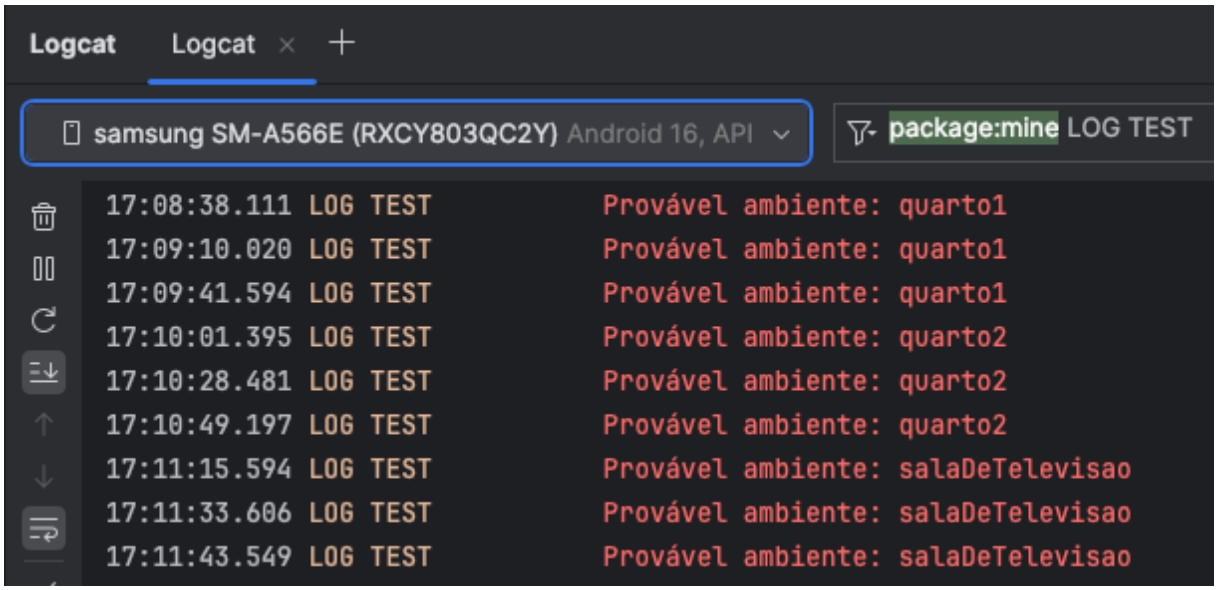
Tabela 1 – Resultados do reconhecimento de ambientes no caso ideal

Teste	***k***	**Ambiente esperado**	**Ambiente estimado**	**Acertou?**
1	1	Quarto 1	Quarto 1	Sim
2	1	Quarto 2	Quarto 2	Sim
3	1	Sala de televisão	Sala de televisão	Sim

Fonte: Elaborada pelo autor.

Com isso, o resultado observado foi que o reconhecimento de ambientes funcionou corretamente no cenário ideal. Conforme a tabela 1, os três ambientes foram estimados de forma correta, resultando em 3 acertos em 3 testes. Além disso, os logs na figura 35 mostram que o resultado se manteve consistente em todas as execuções. Sendo assim, o teste confirma que, quando o ambiente está nas condições mais próximas do mapeamento e as imagens seguem a mesma direção (N/L/S/O), o método consegue identificar o ambiente corretamente.

Figura 35 – *Logs* com resultados do teste de caso ideal



Fonte: Elaborada pelo autor.

4.7.2 Sensibilidade a Mudanças

Este teste teve como objetivo validar o quanto o reconhecimento é sensível a mudanças comuns no ambiente. Como o vetor final depende do padrão visual das imagens (textura e distribuição de intensidades), alterações como iluminação e presença de objetos podem afetar o histograma LBP e mudar as distâncias calculadas pelo KNN.

Nesta etapa, foi aplicada uma mudança por vez antes de tirar as fotos do *"Onde estou?"*. As mudanças foram:

- Variação de iluminação: a iluminação original é artificial branca. Aqui, foram testadas fotos dos mesmos ambientes iluminados por luz artificial amarela e também sem iluminação.
- Inclusão/remoção de objetos: em uma das direções, a foto capturada possuía objeto de tamanho relevante adicionado/removido da cena.
- Variação do ângulo da imagem: em uma das direções, a foto foi capturada fora do alinhamento original (por exemplo, com o celular inclinado 45 graus).

O procedimento realizado foi:

1. Escolher um ambiente já mapeado.
2. Alterar uma condição (por exemplo, iluminação) e executar o *"Onde estou?"* com novas fotos (N/L/S/O).

3. Anotar o resultado apresentado pelo aplicativo.

4. Repetir para as outras condições selecionadas.

Como evidência do teste, a tabela 2 apresenta os resultados do reconhecimento após alterações controladas na cena, mantendo $k = 1$. As variações aplicadas foram mudanças de iluminação (luz amarela e ambiente sem iluminação), inclusão de um objeto novo em uma das direções e variação do ângulo de captura em aproximadamente 45 graus. Além disso, a figura 36 apresenta o *Logcat* do aplicativo registrando o ambiente estimado durante as execuções dos testes por meio da mensagem *"Provável ambiente: [nome-ambiente]"*. Assim, foi possível conferir se o resultado retornado pelo aplicativo fazia sentido em cada situação testada.

Tabela 2 – Resultados do reconhecimento de ambientes com alterações

Teste	**Variação**	***k***	**Ambiente esperado**	**Ambiente estimado**	**Acertou?**
1	Luz amarela	1	Quarto 1	Quarto 1	Sim
2	Luz apagada	1	Quarto 1	Quarto 1	Sim
3	Objeto novo	1	Quarto 1	Quarto 1	Sim
4	Luz amarela	1	Quarto 2	Quarto 2	Sim
5	Luz apagada	1	Quarto 2	Quarto 2	Sim
6	Objeto novo	1	Quarto 2	Quarto 2	Sim
7	Luz amarela	1	Sala de televisão	Sala de televisão	Sim
8	Luz apagada	1	Sala de televisão	Sala de televisão	Sim
9	Objeto novo	1	Sala de televisão	Sala de televisão	Sim
10	Ângulo em 45°	1	Quarto 2	Quarto 2	Sim

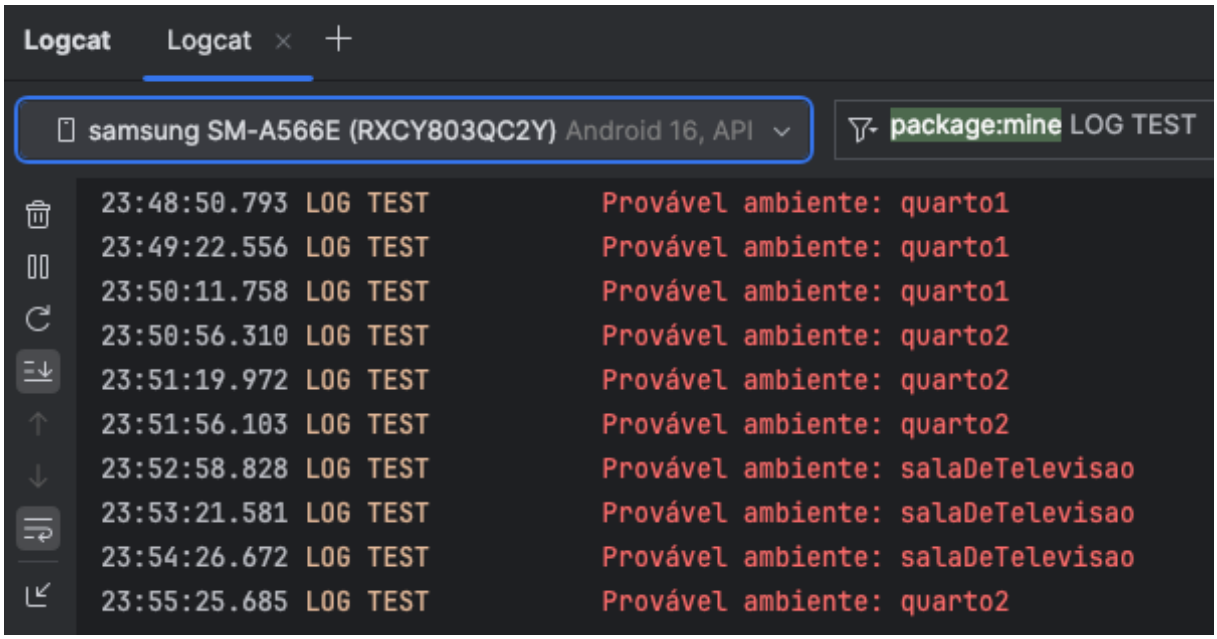
Fonte: Elaborada pelo autor.

Com isso, o resultado observado foi que o reconhecimento de ambientes permaneceu correto em todas as variações testadas, totalizando 10 acerto em 10 tentativas. No caso da iluminação, a variação de luz pode alterar a intensidade e o contraste da imagem (por exemplo, deixando partes mais claras ou mais escuras), o que poderia influenciar o descritor. Mesmo assim, nos testes realizados, o aplicativo manteve o ambiente estimado corretamente.

Na alteração por objeto e por variação de ângulo, a mudança foi aplicada apenas em uma das direções, enquanto as demais imagens permaneceram semelhantes ao mapeamento. Como o reconhecimento utiliza quatro imagens (N/L/S/O) para formar o vetor final, isso acaba ajudando quando uma direção fica diferente, pois as outras ainda contribuem para aproximar a consulta do ambiente correto.

Por fim, vale destacar que variações mais extremas, como escuro total, imagens com estouro de brilho ou muitas mudanças de objetos nas quatro direções, podem aumentar a chance de erro do reconhecimento e não foram o foco destes testes.

Figura 36 – *Logs* com resultados do teste de ambientes com alterações



Fonte: Elaborada pelo autor.

4.7.3 Importância da Ordem das Direções

Este teste teve como objetivo validar a importância de manter a mesma ordem das direções no *"Mapear ambiente"* e no *"Onde estou?"*. No aplicativo, o vetor final é formado pela concatenação dos descritores na ordem norte-leste-sul-oeste. Por isso, quando o *"Onde estou?"* segue a mesma ordem do *"Mapear ambiente"*, a comparação é feita norte com norte, leste com leste, etc. O procedimento realizado foi:

1. Escolher um ambiente já mapeado. Utilizar as mesmas fotos do teste "Caso Ideal".
2. Executar o *"Onde estou?"* com a ordem alterada (L/S/O/N) e anotar o resultado.
3. Executar o *"Onde estou?"* com a ordem alterada (S/O/N/L) e anotar o resultado.
4. Comparar os resultados obtidos com a ordem correta (no teste Caso Ideal) e com a ordem alterada.
5. Em caso de falha, repetir o processo com $k = 3$.

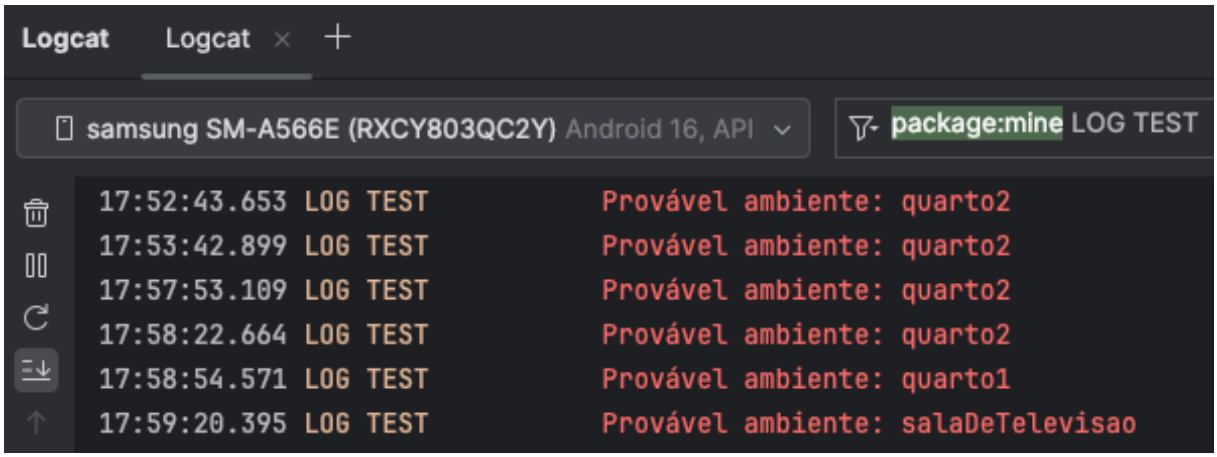
Como evidência do teste, a tabela 3 apresenta os resultados obtidos ao executar o *"Onde estou?"* com a ordem das imagens alterada (L/S/O/N e S/O/N/L), registrando o ambiente estimado e se houve acerto ou erro em cada execução. Além disso, as figuras 37 e 38 mostram os *logs* do *Logcat* durante os testes, onde o aplicativo imprime a mensagem *"Provável ambiente: [nome-ambiente]"* com o resultado estimado. Ainda, nota-se que, em alguns casos, o reconhecimento com $k = 1$ retornou o ambiente incorreto. Nesses casos, o mesmo procedimento foi repetido com $k = 3$ para verificar se o resultado mudava.

Tabela 3 – Resultados do reconhecimento de ambientes com ordem alterada

Teste	Ordem utilizada	k	Ambiente esperado	Ambiente estimado	Acertou?
1	L/S/O/N	1	Quarto 1	Quarto 2	Não
2	L/S/O/N	3	Quarto 1	Quarto 1	Sim
3	S/O/N/L	1	Quarto 1	Quarto 2	Não
4	S/O/N/L	3	Quarto 1	Quarto 1	Sim
5	L/S/O/N	1	Quarto 2	Quarto 2	Sim
6	S/O/N/L	1	Quarto 2	Quarto 2	Sim
7	L/S/O/N	1	Sala de televisão	Quarto 1	Não
8	L/S/O/N	3	Sala de televisão	Quarto 1	Não
9	S/O/N/L	1	Sala de televisão	Sala de televisão	Sim

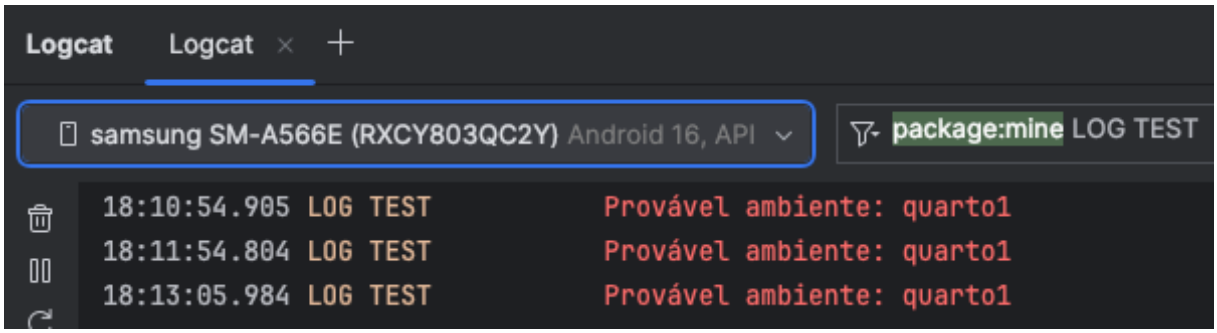
Fonte: Elaborada pelo autor.

Figura 37 – *Logs* com resultados do teste com direções alteradas e $k = 1$



Fonte: Elaborada pelo autor.

Figura 38 – *Logs* com resultados do teste com direções alteradas e $k = 3$



Fonte: Elaborada pelo autor.

Com isso, o resultado observado foi que alterar a ordem das direções pode impactar o reconhecimento, e o efeito não foi igual para todos os ambientes. Para o *Quarto 1*, os testes com $k = 1$ falharam nas duas ordens avaliadas (testes 1 e 3), estimando o ambiente como *Quarto 2*. Ao repetir com $k = 3$, o resultado passou a acertar o ambiente (testes 2 e 4). Já para o *Quarto 2*, o reconhecimento permaneceu correto mesmo com a ordem alterada (testes 5 e 6). Para a *Sala de televisão*, houve comportamentos diferentes: na

ordem L/S/O/N o resultado foi incorreto tanto para $k = 1$ quanto para $k = 3$ (testes 7 e 8), enquanto na ordem S/O/N/L o ambiente foi reconhecido corretamente com $k = 1$ (teste 9).

O teste indica que, na implementação atual, a ordem das direções influencia o vetor final usado pelo KNN e pode levar a erros quando não é mantida a mesma convenção do mapeamento. Apesar disso, os resultados também mostram que a sensibilidade à ordem pode variar conforme o ambiente e o valor de k, já que em alguns casos o reconhecimento continuou correto mesmo com a ordem alterada.

5 CONCLUSÕES

Este trabalho teve como objetivo principal desenvolver um aplicativo Android para controle manual de um robô móvel, com comunicação por comandos via rede e suporte ao *streaming* de vídeo. Também foi implementado um módulo de reconhecimento de ambientes utilizando visão computacional, com extração de características com Local Binary Pattern (LBP) e classificação com K-Nearest Neighbors (KNN) através da biblioteca OpenCV, buscando investigar a viabilidade dessa abordagem para localizar o robô em ambientes internos.

As funcionalidades de conexão, controle manual e streaming de vídeo foram validadas por meio de testes organizados por operação, registrando evidências tanto no Logcat do aplicativo quanto no terminal do robô, o que ajudou a confirmar o funcionamento ponta a ponta do sistema.

Na parte de reconhecimento de ambientes, foi possível mapear ambientes e realizar a consulta pelo *"Onde estou?"*, obtendo bons resultados dentro do escopo testado. Atualmente, a etapa de obtenção de imagens para o reconhecimento é feita a partir da galeria de fotos do *smartphone*, o que facilitou repetir testes e comparar resultados. No entanto, o módulo foi implementado de forma a permitir a troca da origem das imagens (por exemplo, para *frames* do vídeo do robô) sem a necessidade de alterar o seu código. Por esse motivo, o objetivo de operar diretamente sobre o vídeo pode ser considerada parcialmente atendida, pois o reconhecimento foi implementado e validado, mas a entrada ainda não está integrada ao *streaming* de vídeo do robô.

A forma de desenvolvimento e testes adotada foi suficiente para chegar a uma versão funcional do sistema e permitir validações por etapas. Algumas limitações naturais de escopo e de tempo ficaram evidentes, principalmente na parte de reconhecimento de ambientes. Houve uma base pequena de ambientes e poucas amostras por ambiente, o que reduz a variedade de cenários avaliados e abre espaço para evoluções. Ainda assim, os resultados obtidos indicam que a abordagem é viável como solução de baixo custo para cenários indoor considerados neste trabalho.

5.1 Trabalhos Futuros

Como continuação deste trabalho, novas implementações e testes podem ser realizados a fim de melhorar o suporte do aplicativo às funcionalidades do robô e o reconhecimento de ambientes. Entre as indicações, destacam-se:

- Integrar o reconhecimento de ambientes com *frames* do *streaming* de vídeo do robô, substituindo a etapa atual de seleção de imagens pela galeria do smartphone.
- Implementar alternativas para extração de características e comparar os resultados com a solução feita utilizando LBP.
- Expandir a base de treinamento para permitir mais de um conjunto de imagens por ambiente (mais vetores de treino por ambiente) e, com isso, testar valores maiores de k no KNN.
- Implementar alternativas para classificação dos descritores e comparar com a solução feita utilizando KNN.
- Implementar módulos que complementem uma solução visando a movimentação autônoma do robô.

REFERÊNCIAS

AHA, D. W.; KIBLER, D.; ALBERT, M. K. Instance-based learning algorithms. *Machine Learning*, v. 6, n. 1, p. 33–66, 1991. 24, 25

BARELLI, F. *Introdução à visão computacional: uma abordagem prática com Python e OpenCV*. São Paulo: Casa do Código, 2018. 30

BAY, H.; TUYTELAARS, T.; GOOL, L. V. Surf: Speeded up robust features. In: *European Conference on Computer Vision (ECCV)*. [S.l.: s.n.], 2006. p. 404–417. 30

BRADSKI, G. The opencv library. *Dr. Dobb's Journal of Software Tools*, 2000. 29

CORDEIRO, M.; MEYER, B.; ZOLA. Knn exato em gpu. In: SBC. *Anais da XXIII Escola Regional de Alto Desempenho da Região Sul*. Porto Alegre/RS, Brasil, 2023. p. 17–20. 28

DEVELOPERS, G. *Android Architecture*. 2024. Disponível em: https://source.android.com/docs/core/architecture. 32

DEVELOPERS, G. *ART and Dalvik*. 2024. Disponível em: https://source.android.com/docs/core/runtime. 33

DEVELOPERS, G. *Guide to App Architecture*. 2024. Disponível em: https://developer.android.com/topic/architecture. 34

DEVELOPERS, G. *Hardware Abstraction Layer (HAL)*. 2024. Disponível em: https://source.android.com/docs/core/architecture/hal. 33

DEVELOPERS, G. *LiveData overview*. 2025. Disponível em: https://developer.android.com/topic/libraries/architecture/livedata. 43

DEVELOPERS, G. *Save simple data with SharedPreferences*. 2025. Disponível em: https://developer.android.com/training/data-storage/shared-preferences. 46

DEVELOPERS, G. *ViewModel overview*. 2025. Disponível em: https://developer.android.com/topic/libraries/architecture/viewmodel. 42

DOMíNGUEZ, J.; PASCUAL. Ij-opencv: Combining imagej and opencv for processing images in biomedicine. *Computers in Biology and Medicine*, v. 84, p. 189–194, 2017. 30, 31

HAN, J.; KAMBER, M. *Data Mining: Concepts and Techniques*. 2nd. ed. San Francisco: Morgan Kaufmann, 2006. 24, 25, 26

HARALICK, R. M.; SHANMUGAM, K.; DINSTEIN, I. Textural features for image classification. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-3, n. 6, p. 610–621, 1973. 30

HU, M.-K. Visual pattern recognition by moment invariants. *IRE Transactions on Information Theory*, v. 8, n. 2, p. 179–187, 1962. 30

ILIAS, B. et al. Indoor mobile robot localization using knn. In: *2016 6th IEEE International Conference on Control System, Computing and Engineering (ICCSCE)*. [S.l.: s.n.], 2016. p. 211–216. 20, 22

JOHNSON, J.; DOUZE, M.; JéGOU, H. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, v. 7, n. 3, p. 535–547, 2019. 28

LAROSE, D. T. *Discovering Knowledge in Data: An Introduction to Data Mining*. Hoboken, NJ: John Wiley & Sons, 2005. 24, 25

LI, K.-J. et al. Survey on indoor map standards and formats. In: *2019 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*. [S.l.: s.n.], 2019. p. 1–8. 22

LIU, Y. *Localization and Navigation System for Indoor Mobile Robot*. 2022. Disponível em: https://arxiv.org/abs/2212.06391. 21, 22, 23

LOWE, D. G. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, v. 60, n. 2, p. 91–110, 2004. 28, 30

MIRANDA, B. S. *Algoritmos clássicos de aprendizado de máquina aplicados ao problema do reconhecimento de imagens*. Alegrete: [s.n.], 2011. Acesso em: 9 ago. 2025. Disponível em: http://dspace.unipampa.edu.br/bitstream/riu/1547/1/Algoritmos%20classicos%20de%20aprendizado%20de%20maquina%20aplicados%20ao%20problema%20do%20reconhecimento%20de%20imagens.pdf. 25, 27

MUJA, M.; LOWE, D. G. Fast approximate nearest neighbors with automatic algorithm configuration. In: *International Conference on Computer Vision Theory and Applications (VISSAPP)*. [S.l.: s.n.], 2009. p. 331–340. 28

NILOY, M. A. et al. Critical design and control issues of indoor autonomous mobile robots: A review. *IEEE Access*, Institute of Electrical and Electronics Engineers Inc., v. 9, p. 35338–35370, 2021. ISSN 21693536. 19, 20, 21, 23

OJALA, T.; PIETIKäINEN, M.; MäENPää, T. Multiresolution gray-scale and rotation invariant texture classification with local binary patterns. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 24, n. 7, p. 971–987, 2002. 30, 52

PINHO, B. A. d. et al. Sistema automatizado de inspeção de dutos de sistemas de condicionamento de ar. In: INSTITUTO FEDERAL DE SANTA CATARINA (IFSC). *Seminário de Ensino, Pesquisa, Extensão e Inovação do IFSC (SEPEI 2016)*. Santa Catarina, Brasil, 2015. Resumo expandido. 35

REACTIVEX. *CompositeDisposable (RxJava 3 Javadoc)*. 2025. Disponível em: https://reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/disposables/CompositeDisposable.html. 45

RUBLEE, E. et al. Orb: An efficient alternative to sift or surf. In: *IEEE International Conference on Computer Vision (ICCV)*. [S.l.: s.n.], 2011. p. 2564–2571. 28, 30

SIERRA, V. A. C. *Desarrollo de una aplicación móvil con Android Studio y OpenCV para la clasificación de tuercas, tornillos y arandelas*. 2021. https://repository.upb.edu.co/handle/20.500.11912/12544. 30

SILVA, R. D. C. D. *Um estudo sobre a extração de características e a classificação de imagens in-variantes à rotação extraídas de um sensor industrial 3D*. Dissertação (Mestrado) — Universidade Federal do Ceará, Fortaleza, 2014. Acesso em: 9 ago. 2025. Disponível em: http://www.repositorio.ufc.br/bitstream/riufc/10852/1/2014_dis_rdcsilva.pdf. 24

TEAM, O. *OpenCV: Open Source Computer Vision Library*. 2024. Versão 4.10.0, Acesso em: 23 ago. 2025. Disponível em: https://docs.opencv.org/4.x/. 29

Apêndices

APÊNDICE A – PROTOCOLO JSON MINIFICADO

PROTOCOLO JSON MINIFICADO – v1.0 (ESPECIFICAÇÃO UNIFICADA)

Projeto Robô de Telecomunicações – IFSC

(Landell Rover / Shannoninho)

A.1 INTRODUÇÃO

Este documento define o protocolo oficial de comunicação entre clientes remotos e o robô, utilizando JSON minificado transmitido via TCP.

Objetivos do protocolo:

- compacto;
- legível;
- extensível;
- independente de interface (menu local é apenas UI);
- adequado a controle remoto, automação e aplicações móveis.

Cada comando é enviado como UMA LINHA contendo JSON minificado, terminada por `\n`.

O robô sempre responde em JSON minificado.

A.2 FORMATO GERAL DAS MENSAGENS

Formato básico de comando:

```
{"m":"<modulo>","a":"<acao>", ... }
```

Campos comuns:

- m : módulo
- a : ação

- demais campos dependem do módulo

Formato geral de resposta:

- Sucesso simples: `{"ok":1}`
- Sucesso com dados: `{"ok":1, ... }`
- Erro: `{"ok":0,"err":"codigo_erro", ... }`

A.3 MÓDULO "tra"– TRAÇÃO (TB6612)

Controla os motores de tração do robô.

A.3.1 Movimento contínuo

`{"m":"tra","a":"go","d":"w"}` → frente

`{"m":"tra","a":"go","d":"s"}` → ré

`{"m":"tra","a":"go","d":"a"}` → girar esquerda

`{"m":"tra","a":"go","d":"d"}` → girar direita

Resposta: `{"ok":1}`

A.3.2 Movimento com velocidade

Campo adicional:

- v : velocidade (inteiro, escala abstrata)

`{"m":"tra","a":"go","d":"w","v":800}`

`{"m":"tra","a":"go","d":"a","v":600}`

Resposta: `{"ok":1}`

A velocidade passa a valer para movimentos subsequentes.

A.3.3 Ajustar velocidade (sem mover)

`{"m":"tra","a":"set","v":700}`

Resposta: `{"ok":1}`

A.3.4 Parar tração

`{"m":"tra","a":"stop"}`

Resposta: `{"ok":1}`

Observação: a velocidade configurada é mantida

A.3.5 Status da tração

`{"m":"tra","a":"status"}`

Resposta típica: `{"ok":1,"dir":"w","v":700}`

A.3.6 Escala de velocidade (referência)

0–1500 (*clamp* interno)

300–500 : manobra lenta

600–800 : deslocamento normal

900–1200 : rápido

1500 : máximo seguro

A.3.7 Erros de tração

`{"ok":0,"err":"invalid_speed","min":0,"max":1500}`

`{"ok":0,"err":"invalid_direction"}`

A.4 MÓDULO "cam"– CÂMERA PAN/TILT

Controla os servos da câmera.

Limites físicos:

- *pan*: 0–180°
- *tilt*: 45–135° (±45°)

A.4.1 Movimento contínuo

`{"m":"cam","a":"cont","d":"p+"}`

`{"m":"cam","a":"cont","d":"p-"}`

`{"m":"cam","a":"cont","d":"t+"}`

```
{"m":"cam","a":"cont","d":"t-"}
```

```
{"m":"cam","a":"cont","d":"stop"}
```

Resposta: `{"ok":1}`

A.4.2 Movimento absoluto

```
{"m":"cam","a":"set","pan":120}
```

```
{"m":"cam","a":"set","tilt":60}
```

```
{"m":"cam","a":"set","pan":120,"tilt":90}
```

Resposta: `{"ok":1}`

A.4.3 Centralizar câmera

```
{"m":"cam","a":"center"}
```

Resposta: `{"ok":1}`

A.4.4 Oscilação automática

```
{"m":"cam","a":"osc","e":"pan"}
```

```
{"m":"cam","a":"osc","e":"tilt"}
```

Resposta ao final: `{"ok":1}`

A.4.5 Parar câmera

```
{"m":"cam","a":"stop"}
```

Resposta: `{"ok":1}`

A.4.6 Erros de câmera

```
{"ok":0,"err":"invalid_angle","eixo":"tilt","min":45,"max":135}
```

A.5 MÓDULO "stream"– TRANSMISSÃO DE VÍDEO

Controla o *streaming* da câmera (*backend* típico: `mjpg_streamer`).

A.5.1 Iniciar *streaming*

```
{"m":"stream","a":"start"}
```

Resposta: `{"ok":1}`

Erro: `{"ok":0,"err":"stream_running"}`

A.5.2 Parar streaming

`{"m":"stream","a":"stop"}`

Resposta: `{"ok":1}`

A.5.3 Configurar vídeo

`{"m":"stream","a":"cfg","res":"640x480","fps":10,"port":8080,`

`"dev":"/dev/video0"}`

Campos:

- res: "WxH"
- fps: inteiro
- port: inteiro
- dev: string

Resposta: `{"ok":1}`

Erro: `{"ok":0,"err":"stream_running"}`

A.5.4 Status do vídeo

`{"m":"stream","a":"status"}`

Resposta:

```
{
 "ok":1,
 "on":1,
 "cfg":{
   "res":"640x480",
   "fps":10,
   "port":8080,
   "dev":"/dev/video0"
 }
}
```

A.5.5 Erros de vídeo

`{"ok":0,"err":"invalid_resolution"}`

`{"ok":0,"err":"invalid_fps"}`

`{"ok":0,"err":"invalid_port"}`

`{"ok":0,"err":"device_busy"}`

`{"ok":0,"err":"backend_error"}`

A.6 MÓDULO "ant"– ANTENA PAN/TILT (ZIGBEE)

A.6.1 Movimento absoluto

`{"m":"ant","a":"set","pan":150}`

`{"m":"ant","a":"set","tilt":30}`

Resposta: `{"ok":1}`

A.6.2 Leitura de LQI

`{"m":"ant","a":"lqi"}`

Resposta: `{"ok":1,"lqi":-61.2}`

A.6.3 Varredura completa

`{"m":"ant","a":"scan"}`

Resposta ao final: `{"ok":1,"best":{"pan":150,"tilt":60,"lqi":-58.3}}`

A.7 MÓDULO "msg"– MENSAGENS ZIGBEE

`{"m":"msg","a":"send","txt":"ola satelite"}`

Resposta: `{"ok":1}`

A.8 MÓDULO "compass"– BÚSSOLA / IMU

`{"m":"compass","a":"read"}`

Resposta: `{"ok":1,"deg":274.3}`

`{"m":"compass","a":"calib"}` (reservado)

A.9 MÓDULO "sys"– SISTEMA E TELEMETRIA

A.9.1 Telemetria

`{"m":"sys","a":"tele"}`

Resposta típica:

```
{
 "ok":1,
 "bat":12.1,
 "lqi":-60,
 "tra":{"dir":"w","v":700},
 "cam":{"pan":120,"tilt":90},
 "ant":{"pan":150,"tilt":70},
 "imu":{"compass":274.3},
 "stream":{"on":1}
}
```

A.9.2 Parada de emergência

`{"m":"sys","a":"stop_all"}`

Resposta: `{"ok":1}`

A.10 ERROS PADRONIZADOS (GERAIS)

Formato: `{"ok":0,"err":"codigo", ... }`

Códigos comuns:

- `invalid_module`
- `invalid_action`
- `invalid_param`
- `not_supported`

A.11 OBSERVAÇÕES FINAIS

- Este JSON é a API oficial do robô;
- Menus locais são apenas interface humana;

- Protocolo extensível (GPS, bateria, sensores, etc.);
- Compatível com: clientes Python, apps Android, aplicações Web, scripts de automação.