

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE
SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE METAL-MECÂNICA
CURSO DE GRADUAÇÃO EM ENGENHARIA MECATRÔNICA**

Pedro E. Saraiva

**Aplicação de um Veículo Guiado Automatizado (AGV) no Contexto de IoT
Aplicada à Inteligência Perimetral**

FLORIANÓPOLIS, Dezembro de 2023.

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE
SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE METAL-MECÂNICA
CURSO DE GRADUAÇÃO EM ENGENHARIA MECATRÔNICA**

Pedro E. Saraiva

**Aplicação de um Veículo Guiado Automatizado (AGV) no Contexto de IoT
Aplicada à Inteligência Perimetral**

Trabalho de Conclusão de Curso submetido
ao Instituto Federal de Educação, Ciência
e Tecnologia de Santa Catarina como parte
dos requisitos para obtenção do título de
Engenheiro em Engenharia Mecatrônica.

Orientador:
Prof. Gregory Chagas da Costa Gomes,
Mestre

FLORIANÓPOLIS, Dezembro de 2023.

Ficha de identificação da obra elaborada pelo autor.

Saraiva, Pedro
**Aplicação de um Veículo Guiado Automatizado (AGV)
no Contexto de IoT Aplicada à Inteligência Perimetral / Pedro
Saraiva; orientação de Gregory Chagas da Costa
Gomes. - Florianópolis, SC, 2024.**
72 p.

**Trabalho de Conclusão de Curso (TCC) - Instituto Federal
de Santa Catarina, Câmpus Florianópolis. Bacharelado
em Engenharia Mecatrônica. Departamento
Acadêmico de Metal Mecânica.
Inclui Referências.**

**1. Integração de sistemas. 2. Desenvolvimento de
Software. 3. CFTV IP. 4. Robótica móvel. 5. Segurança
eletrolétrica. I. Chagas da Costa Gomes, Gregory . II. Instituto
Federal de Santa Catarina. III. Aplicação de
um Veículo Guiado Automatizado (AGV) no Contexto de IoT
Aplicada à Inteligência Perimetral.**

**APLICAÇÃO DE UM VEÍCULO GUIADO AUTOMATIZADO (AGV) NO CONTEXTO
DE IOT APPLICADA À INTELIGÊNCIA PERIMETRAL**

PEDRO E. SARAIVA

Este trabalho foi julgado adequado para obtenção do título de Engenheiro em Engenharia Mecatrônica e aprovado na sua forma final pela banca examinadora do Curso de Graduação em Engenharia Mecatrônica do Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina.

Florianópolis, 16 de fevereiro, 2024.

Banca Examinadora:

Gregory Chagas da Costa Gomes, Me.

Maurício Edgar Stivanello, Dr.

IFSC – Instituto Federal
de Santa Catarina

Roberto Alexandre Dias, Dr.

IFSC – Instituto Federal
de Santa Catarina

RESUMO

Este projeto tem como objetivo principal a implementação de um Veículo Autônomo Guiado (AGV), especificamente o Robotino, para aprimorar a inteligência perimetral em um sistema integrado de vigilância, Defense IA, utilizando um Circuito Fechado de Televisão (CFTV). Com objetivos de automatizar o controle do Robotino em resposta a eventos externos, direcioná-lo autonomamente a pontos de interesse, gerar notificações digitais (eventos) por meio do Defense IA aplicado ao CFTV e integrar todos os sistemas via rede, a integração será realizada por meio de um middleware, com capacidades de gerenciamento e processamento de dados, permitindo a troca eficiente de informações entre todos os dispositivos envolvidos no sistema. Utilizando protocolos de comunicação MQTT e HTTP (comandos API), em conjunto com dispositivos ESP32 posicionados estrategicamente como âncoras ao redor do cenário do CFTV, o resultado almejado é um protótipo de robô autônomo capaz de deslocar-se em direção a pontos de interesse em resposta a eventos de inteligência perimetral, contribuindo significativamente para a automação e eficiência na validação de segurança.

Palavras-chave: Integração de sistemas. Desenvolvimento de Software. CFTV IP. Robótica móvel. Segurança eletrônica.

ABSTRACT

This project aims to implement an Autonomous Guided Vehicle (AGV), specifically the Robotino, to enhance perimeter intelligence in an integrated surveillance system, Defense IA, using a Closed-Circuit Television (CCTV). The objectives include automating the control of Robotino in response to external events, autonomously directing it to points of interest, generating digital notifications (events) through Defense IA applied to CCTV, and integrating all systems via a middleware with data management and processing capabilities, allowing efficient exchange of information between all devices involved in the system. Using MQTT and HTTP communication protocols (API commands), along with strategically positioned ESP32 devices as anchors around the CCTV scenario, the desired outcome is a prototype of an autonomous robot capable of moving towards points of interest in response to perimeter intelligence events, significantly contributing to automation and efficiency in security validation.

Keywords: System integration. Software development. IP CCTV. Mobile robotics. Electronic security.

LISTA DE FIGURAS

Figura 1 – Cenário exemplar	14
Figura 2 – Defense IA V3	16
Figura 3 – Arquitetura Cliente x Servidor	17
Figura 4 – Visualização de vídeo ao vivo - Reconhecimento facial	18
Figura 5 – Histórico de eventos	19
Figura 6 – Robotino 4	20
Figura 7 – Módulo motor	21
Figura 8 – Sensor IR GP2D12	22
Figura 9 – PC/104	22
Figura 10 – Condições de operação recomendadas	23
Figura 11 – Características de Wifi	24
Figura 12 – Diagrama de comunicação	27
Figura 13 – Robotino View 2: Programa para navegação	28
Figura 14 – API 1: Documentação em Doxygen	29
Figura 15 – Interface Web - Robotino	30
Figura 16 – Captura de intensidade do sinal do Robotino	31
Figura 17 – Diagrama de comunicação de sistemas	32
Figura 18 – Integração com Defense IA	34
Figura 19 – Acesso remoto ao Robotino via ssh	35
Figura 20 – Wireshark - Visualização de pacotes	36
Figura 21 – Wireshark - exemplo	36
Figura 22 – Solicitação de estado do Bumper	38
Figura 23 – Fluxo de requisição à API SOAP do Robotino	38
Figura 24 – Fluxograma de funções da ESP32	39
Figura 25 – Fluxograma de rotinas do Middleware	40
Figura 26 – Diagrama de cenário CFTV	41
Figura 27 – Diagrama de notificação de evento no Defense IA	42
Figura 28 – Pessoa cadastrada no Defense IA	42
Figura 29 – S.L.I.	43
Figura 30 – Blocos do Middleware	45
Figura 31 – Conexão MQTT	46
Figura 32 – Função <i>messageHandler</i>	47
Figura 33 – <i>onMessageReceived</i>	48
Figura 34 – Funções da Token	49
Figura 35 – Função <i>generateSoapToken</i>	50
Figura 36 – Função <i>keepAlive</i>	51
Figura 37 – Função <i>move</i>	53

Figura 38 – Movendo o Robotino para frente	54
Figura 39 – Função <i>verifyApproach</i>	55
Figura 40 – Função <i>girar180</i>	56
Figura 41 – Final da movimentação	56
Figura 42 – Função <i>sendEventToDefense</i>	57
Figura 43 – Função <i>pushEventToDefense</i>	58

LISTA DE TABELAS

Tabela 1 – Endpoints	37
Tabela 2 – Medições de valores RSSI ao longo de 6 metros	44
Tabela 3 – Atribuição de dispositivos a índices a partir de eventos	47
Tabela 4 – Funções DeviceManager	49
Tabela 5 – Funções IOControl	52

LISTA DE ABREVIATURAS E SIGLAS

AGV	Veículo Autônomo Guiado
API	Application Programming Interface
CFTV	Círculo Fechado de Televisão
HTTP	HyperText Transform Protocol
IA	Inteligência Artificial
MQTT	Message Queuing Telemetry Transport
RSSI	Received Signal Strength Indicator
S.L.I.	Sistema de Localização Interno
WLAN	Wireless Local Area Network

SUMÁRIO

1	RESUMO	12
2	INTRODUÇÃO	13
2.1	Premissa	13
2.2	Objetivos do Projeto	15
3	TECNOLOGIAS	16
3.1	Defense IA	16
3.1.1	Conexão de dispositivos e monitoramento	17
3.1.1.1	<i>Protocolo CFTV</i>	18
3.1.2	Eventos baseados em inteligências	19
3.2	Robotino	20
3.2.1	Sensores de navegação	21
3.2.2	Unidade de processamento	22
3.3	ESP32	23
3.3.1	RSSI - Received Signal Strength Indicator	24
3.4	Middleware	25
3.4.1	Protocolo MQTT	25
3.4.2	API - Application Programming Interface	26
4	METODOLOGIA	27
4.1	Comunicação com Robotino	27
4.1.1	Robotino View 2	27
4.1.2	Open Robotino API	28
4.1.3	Open Robotino API 2	29
4.1.3.1	<i>Servidor web</i>	30
4.2	Configuração de dispositivos ESP32	31
4.2.1	Medição de RSSI	31
4.2.2	Comunicação via Broker MQTT	32
4.3	Integração de sistemas	32
4.3.1	Integração com o Sistema de Localização	33
4.3.2	Integração com Defense IA	33
5	DESENVOLVIMENTO	35
5.1	Controle do Robotino remotamente	35
5.1.1	Análise de pacotes de comunicação	35
5.1.2	Acesso a funções de comando	37
5.2	Construção de S.L.I. - Sistema de Localização Interno	39
5.2.1	Programação da ESP32	39
5.3	Desenvolvimento do Middleware	40
5.4	Inteligência Perimetral	41
5.4.1	Cadastro e Configuração de Eventos	41
6	RESULTADOS	43
6.1	Sistema de localização interno	43
6.2	Middleware	44
6.2.1	Clientes MQTT	45
6.2.1.1	<i>Processamento de mensagens recebidas - Broker Defense IA</i>	46
6.2.1.2	<i>Processamento de mensagens recebidas - Broker RSSI</i>	48
6.2.2	Clientes API SOAP	48

6.2.2.1	<i>Serviço de conexão - DeviceManager</i>	49
6.2.2.2	<i>Serviço de controle - IOControl</i>	51
6.2.3	Cliente API REST - Defense IA	57
6.3	Acionamento do Robotino em operação	58
6.4	Conclusão sobre resultados	59
7	CONSIDERAÇÕES FINAIS	60
7.1	Comunicação com Robotino	60
7.1.1	Latência na comunicação	60
7.1.2	Integração da web-cam	60
7.1.3	Segurança de dados durante comunicação	60
7.2	Navegação do Robotino	61
	REFERÊNCIAS	62
	APÊNDICES	63

1 RESUMO

Este projeto tem como objetivo principal a implementação de um Veículo Autônomo Guiado (AGV), especificamente o Robotino, para aprimorar a inteligência perimetral em um sistema integrado de vigilância, Defense IA, utilizando um Circuito Fechado de Televisão (CFTV). Com objetivos de automatizar o controle do Robotino em resposta a eventos externos, direcioná-lo autonomamente a pontos de interesse, gerar notificações digitais (eventos) por meio do Defense IA aplicado ao CFTV e integrar todos os sistemas via rede, a integração será realizada por meio de um middleware, com capacidades de gerenciamento e processamento de dados, permitindo a troca eficiente de informações entre todos os dispositivos envolvidos no sistema. Utilizando protocolos de comunicação MQTT e HTTP (comandos API), em conjunto com dispositivos ESP32 posicionados estratégicamente como âncoras ao redor do cenário do CFTV, o resultado almejado é um protótipo de robô autônomo capaz de deslocar-se em direção a pontos de interesse em resposta a eventos de inteligência perimetral, contribuindo significativamente para a automação e eficiência na validação de segurança.

2 INTRODUÇÃO

Sistemas inteligentes e autônomos têm como objetivo reduzir a carga de trabalho humana em atividades designadas; apresentam maior confiabilidade, repetibilidade e, geralmente, maior precisão. A integração dessas tecnologias na vida cotidiana torna-se uma necessidade cada vez mais urgente, influenciando diretamente a segurança pública e o bem-estar da sociedade.

À medida em que robôs autônomos tornam-se protagonistas em diversos setores, desde a indústria, à assistência médica, suas aplicações começam a remodelar a maneira como enfrentamos desafios cotidianos. A ascensão desses auxiliares robóticos promete impactar positivamente a vida das pessoas, proporcionando eficiência, inovação e segurança.

Setores diversos já colheram benefícios significativos com a implementação desses sistemas. A logística, por exemplo, testemunhou melhorias substanciais na gestão de estoques e distribuição. Na saúde, cirurgias assistidas por robôs estão elevando os padrões de precisão e recuperação. Em paralelo, a indústria automotiva vislumbra um futuro de veículos autônomos que prometem não apenas revolucionar o transporte, mas também otimizar a segurança viária.

À medida que essas inovações tecnológicas se integram à rotina, torna-se viável enxergar o investimento em segurança pública como uma oportunidade estratégica e necessária. Ambientes de alto tráfego, como shoppings centers, estádios, aeroportos e grandes eventos, estão adotando sistemas autônomos para aprimorar a vigilância e resposta a situações críticas. A constante busca por acertabilidade na segurança e a necessidade de garantir redundância em ambientes dinâmicos, onde a quantidade de informação e metadados é massiva, tornam-se desafios iminentes. Nesse contexto, será explorado como esses sistemas autônomos não apenas transformam a paisagem tecnológica, mas também demandam estratégias inovadoras para garantir a segurança pública de maneira eficaz e abrangente.

2.1 Premissa

A premissa do presente trabalho de conclusão de curso é fundamentada em uma interseção estratégica entre a engenharia mecatrônica e o cenário prático proporcionado pelo estágio na renomada empresa brasileira Intelbras, líder no mercado brasileiro de segurança eletrônica (INTELBRAS, 2023a). Originada a partir de uma oportunidade identificada durante essa experiência profissional, a decisão de integrar essas duas áreas culminou na definição do tema central do projeto: desenvolver uma integração para construir um sistema de segurança perimetral autônomo.

A escolha do Robotino, fornecido pela empresa alemã Festo e disponibilizado

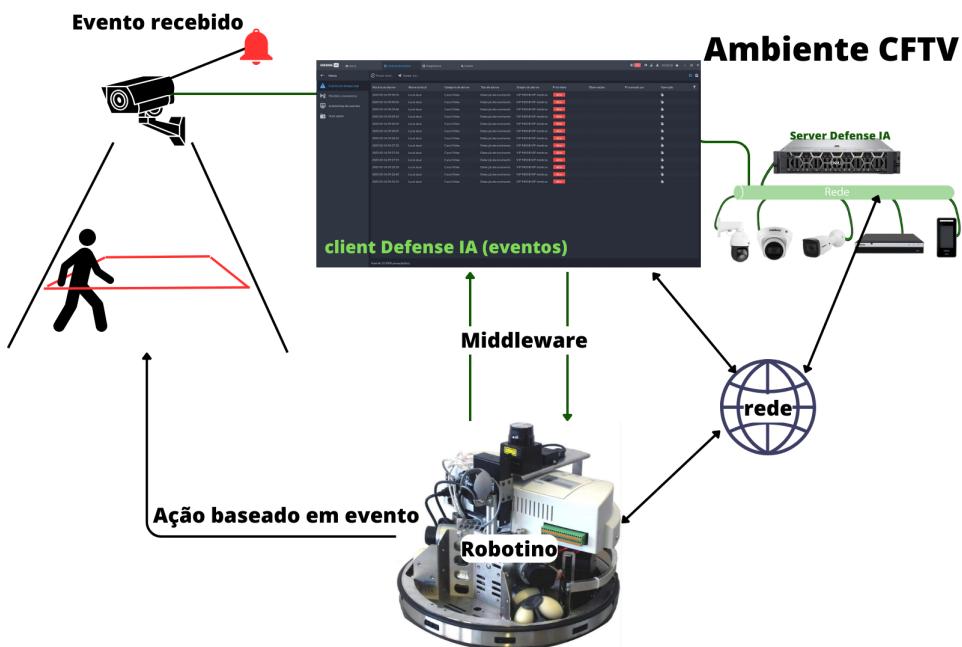
pela universidade para a execução do Trabalho de Conclusão de Curso, revelou-se como uma decisão estratégica. Reforçando essa escolha, Kracht e Nielsen (KRACHT; NIELSEN, 2007), já haviam explorado o potencial do Robotino em um contexto de robô autônomo interativo, aplicado no cotidiano das pessoas. Tal referência justifica a plataforma como ideal para experimentos relacionados ao desenvolvimento de um sistema de segurança perimetral autônomo.

Adicionalmente, a utilização do software Defense IA, desenvolvido pela Intelbras e familiar durante o estágio, agrega uma camada crucial de inteligência perimetral ao projeto. Este software está fortemente vinculado à segurança, como destacado pela Intelbras, ao incluir tecnologias que "identificam, categorizam e classificam os diferentes elementos presentes no local, sendo possível filtrar as ocorrências, apontando o que merece atenção e ignorando o que não representa risco"(INTELBRAS, 2023b).

Essa abordagem proporciona ao vigilante uma atuação mais assertiva, mitigando falhas humanas como alarmes falsos, e representa uma contribuição valiosa para a eficácia do sistema proposto.

A figura abaixo ilustra um cenário exemplar do tema em questão, demonstrando a sinergia entre o Robotino, atuando como um robô autônomo, e o Defense IA, desempenhando o papel de agente de inteligência perimetral. A imagem evidencia de que forma a integração entre esses componentes em conjunto à aplicação de diversas inteligências ao cenário podem ser eficazes, fluindo de maneira coordenada.

Figura 1 – Cenário exemplar



Fonte: Autor.

Um sistema intermediador (Middleware) pode assumir a responsabilidade pela comunicação entre os sistemas, processando informações provenientes de diversas tecnologias e utilizando diferentes protocolos. Dessa forma, ele desempenha o papel crucial de integrar sistemas diversos.

Com base nessas premissas, é possível estabelecer objetivos, geral e específicos, para a execução do trabalho, delineando metas a serem alcançadas ao longo do desenvolvimento do protótipo.

2.2 Objetivos do Projeto

Os objetivos do projeto, geral e específicos, são elencados a seguir.

Objetivo Geral:

- Aplicar um AGV (Robotino) no contexto de inteligência perimetral em um sistema de vigilância integrado a um CFTV para efetuação de rondas realizadas a partir de eventos específicos.

Objetivos Específicos:

- Controlar o Robotino de maneira automática a partir de eventos externos;
- Comandar o Robotino para alcançar pontos de interesse dentro do ambiente de vigilância de maneira autônoma ao longo de um eixo;
- Gerar notificações digitais (Eventos) a partir do Defense IA aplicado a um cenário de CFTV (Círculo Fechado de Televisão);
- Integrar todos os sistemas via rede com objetivo de enviar o robô móvel automaticamente para validar eventos notificados pela plataforma de segurança.

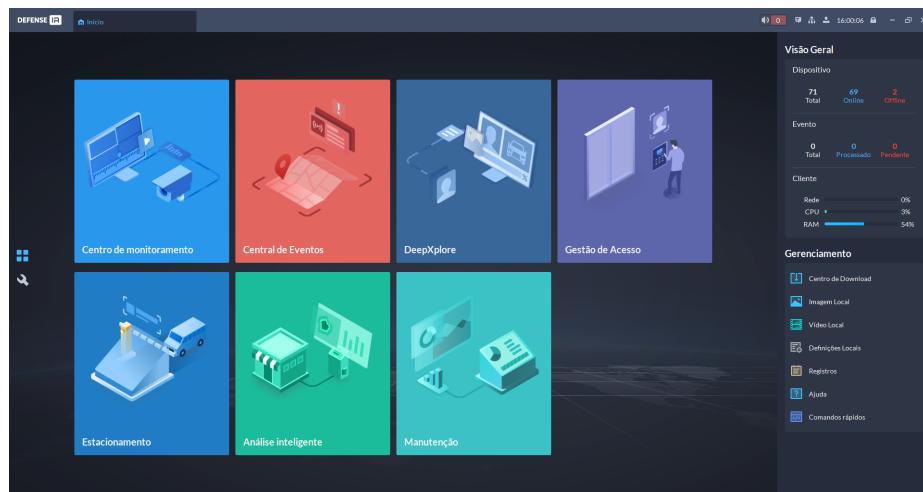
3 TECNOLOGIAS

Neste capítulo, são definidas as tecnologias utilizadas durante o projeto, abordando suas capacidades, destacando potenciais de aplicação no meio de segurança eletrônica, assim, justificando a escolha de cada.

3.1 Defense IA

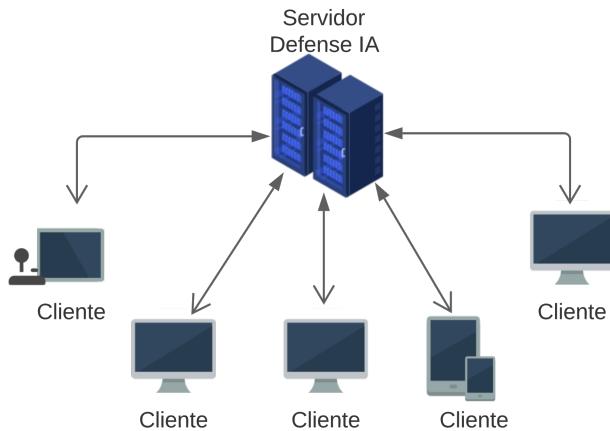
Desenvolvido pela Intelbras®, o software Defense IA caracteriza-se por apresentar um gerenciamento centralizado de segurança; o sistema dispõe interfaces de controle para monitoramento de vídeo, controle de acesso, eventos e alarmes, administração de dispositivos, recursos de Inteligência Artificial, entre outras funcionalidades que compõem o ecossistema de segurança. A plataforma destaca-se por apresentar uma ampla capacidade de integração entre seus recursos, permitindo sua aplicação em diferentes cenários.

Figura 2 – Defense IA V3



Fonte: Autor (2023).

Baseado em uma estrutura cliente-servidor, o Defense IA apresenta uma arquitetura de rede descentralizada, ou seja, múltiplos clientes podem se conectar ao servidor central, acessando seus serviços e recursos.

Figura 3 – Arquitetura Cliente x Servidor

Fonte: Autor (2023).

Isso torna a estrutura de informação escalável e eficiente, permitindo a distribuição da execução de tarefas entre múltiplos dispositivos enquanto apresenta uma centralização da utilização de recursos para gerenciamento e processamento de serviços. (INTELBRAS, 2023c)

3.1.1 Conexão de dispositivos e monitoramento

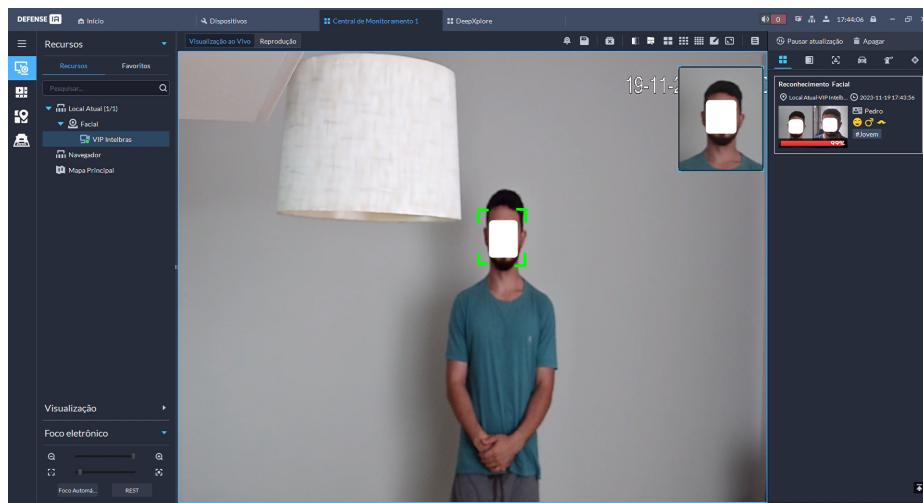
O Defense IA desempenha um papel central no sistema, oferecendo funcionalidades avançadas de monitoramento por vídeo e integração com dispositivos de segurança eletrônica. A plataforma permite a conexão e gestão eficiente de dispositivos de vigilância, sendo que, para este projeto, optou-se pela utilização exclusiva de câmeras IP, as quais podem ser adicionadas através de diferentes métodos, tais como endereço IP, nome de domínio ou protocolo de registro automático.

As câmeras IP selecionadas para o protótipo pertencem à mesma fabricante que o software, neste caso, a Intelbras. Essa afinidade é estratégica, pois permite a utilização de um protocolo nativo do Defense IA para acessar as inteligências incorporadas nos dispositivos Intelbras. Entre as principais funcionalidades oferecidas por esse protocolo, destacam-se:

- Detecção e reconhecimento facial
- Detecção de movimento
- Leitura de placas de veículos
- Detecção de veículos e pessoas

A integração harmoniosa entre o Defense IA e as câmeras Intelbras proporciona uma eficaz coleta de dados e a ativação de funcionalidades avançadas para fortalecer a segurança. A figura abaixo apresenta um canal de vídeo inteligente ao vivo conectado à plataforma, realizando reconhecimento facial.

Figura 4 – Visualização de vídeo ao vivo - Reconhecimento facial



Fonte: Autor (2023).

A Central de Monitoramento, uma das interfaces do Defense IA, representa o ponto central de controle e supervisão do sistema. Através deste menu, o operador pode visualizar os canais de vídeo provenientes dos dispositivos adicionados à plataforma, além de interagir com diversas funcionalidades, tais como gravação de vídeo, captura de imagens, acionamento de alarmes e comunicação bidirecional de áudio. Esta interatividade amplia significativamente a capacidade de resposta do sistema, permitindo uma gestão mais eficiente e personalizada da segurança.

3.1.1.1 Protocolo CFTV

O sistema de CFTV, ou Circuito Fechado de Televisão, desempenha um papel crucial no ecossistema de segurança eletrônica, especialmente no contexto do software Defense IA. Projetado para viabilizar a transmissão segura e criptografada de dados, o protocolo CFTV possibilita a aquisição eficiente de informações, como transmissões de vídeo ao vivo, áudio e a capacidade de realizar inteligência na borda, sem a necessidade de enviar todos os dados para uma central de processamento remota.

Essa eficiência na transmissão permite que dados cruciais, incluindo inteligências na borda, transmissões ao vivo e históricos, sejam acessados de maneira otimizada, contribuindo para operações avançadas de monitoramento e inteligência no

âmbito da segurança eletrônica, com a integração harmoniosa entre câmeras IP e o Defense IA.

3.1.2 Eventos baseados em inteligências

Além da funcionalidade central de monitoramento, o software de segurança eletrônica apresenta um menu dedicado aos eventos na sua interface, a Central de Eventos. Este menu tem como função consolidar e gerenciar todos os eventos recebidos pela plataforma. A figura abaixo apresenta o histórico de eventos ocorridos na plataforma.

Figura 5 – Histórico de eventos

Número	Hora do Al...	Nome do L...	Categoria...	Tipo de Alarme	Origem do ...	Prioridade	Comentários	Processad...	Status
1	2023-11-19...	Local Atual	Dispositivos	O dispositivo está desconectado	Facial	Alta			Não pr
2	2023-11-18...	Local Atual	Dispositivos	O dispositivo está desconectado	Facial	Alta			Não pr
3	2023-11-17...	Local Atual	Dispositivos	O dispositivo está desconectado	Facial	Alta			Não pr
4	2023-11-17...	Local Atual	Dispositivos	O dispositivo está desconectado	Facial	Alta			Não pr
5	2023-11-17...	Local Atual	Dispositivos	O dispositivo está desconectado	Facial	Alta			Não pr
6	2023-11-17...	Local Atual	Dispositivos	O dispositivo está desconectado	Facial	Alta			Não pr
7	2023-11-16...	Local Atual	Robotino	1	22	Alta			Não pr
8	2023-11-16...	Local Atual	Robotino	1	2	Alta			Não pr
9	2023-11-16...	Local Atual	Robotino	11	22	Alta			Não pr
10	2023-11-16...	Local Atual	Robotino	1	2	Alta			Não pr
11	2023-11-16...	Local Atual	Robotino	11	22	Alta			Não pr
12	2023-11-16...	Local Atual	Robotino	1	2	Alta			Não pr
13	2023-11-16...	Local Atual	Robotino	1	2	Alta			Não pr
14	2023-11-16...	Local Atual	Robotino	1	2	Alta			Não pr
15	2023-11-16...	Local Atual	Robotino	1	2	Alta			Não pr
16	2023-11-16...	Local Atual	Robotino	11	22	Alta			Não pr
17	2023-11-16...	Local Atual	Robotino	1	2	Alta			Não pr

Fonte: Autor (2023).

Na central de eventos é possível configurar regras de alarmes, atribuindo usuários da plataforma a serem notificados quando algum evento registrado ocorrer. Além disso, essa interface também permite integrar outras funções do software a partir da ocorrência de um evento, como apresentar uma janela de vídeo como *pop-up*, acionar alarmes sonoros e/ou visuais, acionar saídas digitais de dispositivos na plataforma, enviar um e-mail, enviar comandos utilizando o protocolo HTTP, entre outros.

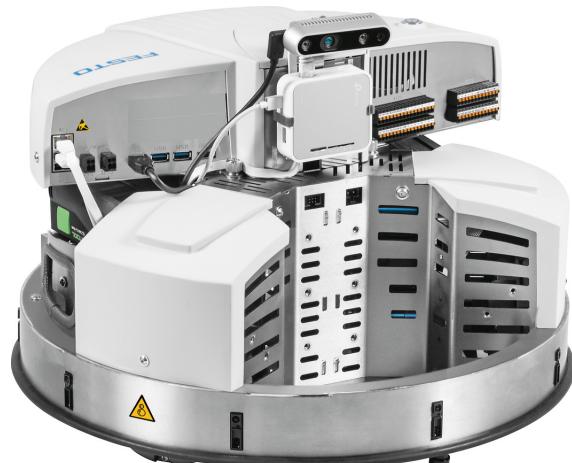
Uma grande capacidade do software Defense IA é a integração com sistemas externos, permitindo a criação de eventos personalizados a partir de informações externas à plataforma.

3.2 Robotino

O Robotino, um dispositivo robótico móvel desenvolvido pela Festo®, destaca-se por possuir uma variedade de sensores e atuadores para fins didáticos e de pesquisa. Equipado com um sistema de navegação omnidirecional e um computador embarcado, o Robotino possui a capacidade de se deslocar de forma autônoma por ambientes.

Utilizado em espaços internos, suas aplicações caracterizam-se por empregar o Robotino como um robô utilitário, interagindo não só com o ambiente ao seu redor, mas também com pessoas. Beelen, Bernard et al. (BEELEN; BERNARD; PAPENMEIER, 2016) exploram a aplicação do Robotino em um experimento que aborda o controle remoto do Robotino a partir de uma interface amigável para crianças. O conjunto de hardware e software disponíveis na estrutura do Robotino o torna um ótimo dispositivo para interação com humanos, uma vez que é capaz de interpretar seus arredores de maneira eficiente e interagir com este de forma precisa e útil. A figura abaixo apresenta a última versão do Robotino produzida pela Festo®.

Figura 6 – Robotino 4



Fonte: (FESTO, 2023).

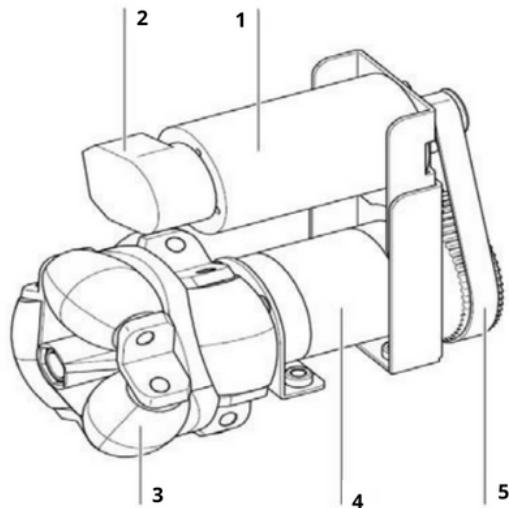
Como explorado por Ruiz (RUIZ, 2016), o Robotino possui as seguintes características físicas:

- Diâmetro: 370 mm
- Altura total: 210 mm
- Peso aproximado: 11 kg

Sua navegação é de responsabilidade de três motores DC alimentados por uma bateria de 12 V @ 4 Ah. Tais motores são acoplados à rodas omnidirecionais,

formando um ângulo de 120º entre cada. A figura abaixo apresenta a estrutura de um conjunto desses.

Figura 7 – Módulo motor



Fonte: (RUIZ, 2016).

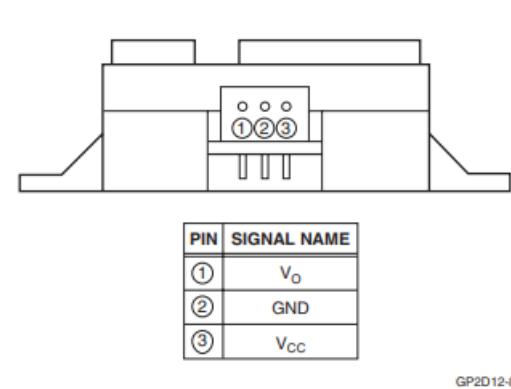
De acordo com os índices da figura, os componentes podem ser listados em:

1. Motor DC;
2. Caixa de redução 16:1;
3. Rodas omnidirecionais;
4. Encoder incremental;
5. Correia dentada.

Além de acessórios adicionais presentes no catálogo da Festo®, o Robotino também possui em sua estrutura outros sensores e dispositivos que serão discutidos nos tópicos subsequentes.

3.2.1 Sensores de navegação

Ao redor de seu chassi, o robô apresenta 9 sensores infravermelhos de proximidade, os sensores formam um ângulo de 40º entre si. A imagem a seguir, retirada de seu *datasheet*, apresenta o sensor GP2D12.

Figura 8 – Sensor IR GP2D12

Fonte: (SHARP, 2023).

Além dos sensores de proximidade, também ao redor do Robotino encontra-se o sensor de colisão, *bumper*, uma fita de borracha, que ao ser pressionada transmite um sinal à unidade de controle.

3.2.2 Unidade de processamento

Acoplados em sua unidade de processamento encontram-se uma web-cam conectada via USB, e uma interface I/O, com entradas e saídas digitais e analógicas. Sua unidade de processamento integrada, demonstrada na imagem abaixo, opera a partir da arquitetura PC/104.

Figura 9 – PC/104

Fonte: (RUIZ, 2016).

Equipado com um processador AMD LX800 de 400 MHz, uma memória SDRAM de 1 GB e um disco Flash de 1 MB que contém o sistema operacional, esse PC oferece uma base sólida para o funcionamento do Robotino. O suporte aos

sistemas operacionais Linux com Kernel em tempo real (RTAI) proporciona um ambiente flexível para a execução de aplicações, enquanto bibliotecas disponíveis enriquecem a funcionalidade do sistema.

Essa configuração técnica versátil permite a execução de uma variedade de aplicações. Além disso, a presença da placa de controle de entradas e saídas amplia a capacidade do Robotino ao facilitar a interação com sensores, motores, entradas digitais e analógicas. O Access Point sem fio complementa essa estrutura, proporcionando conectividade e mobilidade ao sistema, tornando possível conectar-se ao Robotino a partir de uma *WLAN* própria.

O Robotino apresenta compatibilidade com diferentes tipos de interfaces para sua configuração e programação, algumas serão exploradas durante o capítulo 4. Metodologia.

3.3 ESP32

O dispositivo ESP32, desenvolvido pela Espressif Systems®, é um componente-chave neste projeto, desempenhando um papel central na implementação de um sistema de referência espacial a partir de uma rede *wireless*. Este microcontrolador Wi-Fi e Bluetooth de baixo custo oferece uma gama abrangente de recursos técnicos, destacando-se por seu processador dual-core potente e conectividade sem fio integrada. As figuras abaixo, retiradas do *Datasheet* do fabricante, apresentam dados técnicos sobre condições de operação recomendadas e características de conexão wifi, respectivamente.

Figura 10 – Condições de operação recomendadas

Symbol	Parameter	Min	Typ	Max	Unit
V _{D33}	Power supply voltage	3.0	3.3	3.6	V
I _{VDD}	Current delivered by external power supply	0.5	—	—	A
T	Operating ambient temperature	85 °C version 105 °C version	-40	—	85 105 °C

Fonte: (ESPRESSIF, 2023).

Figura 11 – Características de Wifi

Name	Description	
Center frequency range of operating channel		2412 ~ 2484 MHz
Wi-Fi wireless standard		IEEE 802.11b/g/n
Data rate	20 MHz	11b: 1, 2, 5.5, 11 Mbps 11g: 6, 9, 12, 18, 24, 36, 48, 54 Mbps 11n: MCS0-7, 72.2 Mbps (Max)
	40 MHz	11n: MCS0-7, 150 Mbps (Max)
Antenna type		PCB antenna, external antenna ²

Fonte: (ESPRESSIF, 2023).

Ao incorporar o ESP32, ganha-se a capacidade de realizar configurações iniciais cruciais por meio da função *setup*, além de executar operações contínuas essenciais através da função *loop*. Contudo, apesar de irrelevante à aplicação neste projeto, é fundamental considerar suas limitações, como a capacidade de processamento limitada em comparação com alternativas mais robustas. A seleção do ESP32 para este projeto é respaldada pela combinação única de custo acessível e recursos substanciais, estabelecendo uma base sólida para a implementação de soluções eficazes em segurança eletrônica. Neste contexto, o próximo tópico explorará como o ESP32 será utilizado com as demais tecnologias, alinhando suas capacidades às demandas específicas do projeto.

3.3.1 RSSI - Received Signal Strength Indicator

O RSSI, ou Indicador de Força do Sinal Recebido (Received Signal Strength Indicator), é uma métrica crucial em redes sem fio que quantifica a intensidade do sinal de rádio entre dispositivos. Ele descreve a potência do sinal recebido por um dispositivo receptor, fornecendo uma medida da qualidade da comunicação sem fio. Em ambientes internos, onde a interferência é uma consideração importante, o RSSI desempenha um papel significativo na determinação da distância entre dispositivos.

O RSSI (Indicador de Força do Sinal Recebido) é uma métrica vital em redes sem fio, quantificando a intensidade do sinal entre dispositivos. Em ambientes internos, onde a interferência é crítica, o RSSI desempenha um papel significativo na determinação da distância entre dispositivos. Bellecier, Jabour et al. (BELLECIERI; JABOUR; JABOUR, 2016) destacam a eficácia do RSSI, propondo um método bidirecional para estimar distâncias. A adaptação desse método a diferentes ambientes reforça a escolha do RSSI como uma tecnologia fundamental para sistemas de localização indoor, demonstrando sua relevância na busca por soluções precisas e adaptáveis.

3.4 Middleware

Um Middleware, em um contexto computacional, é uma camada de software que atua como intermediário entre sistemas heterogêneos, possibilitando a comunicação e a interação eficientes entre eles. A implementação de um Middleware justifica-se por sua capacidade de interagir com sistemas onde a diversidade de linguagens de programação e hardwares é presente. É imperativo que o desenvolvimento do middleware tenha como objetivo primário a harmonização da comunicação entre sistemas distribuídos, independente das tecnologias subjacentes. Essa abordagem estratégica simplifica significativamente a integração de aplicações desenvolvidas em linguagens diversas e executadas em plataformas distintas, fomentando, assim, a interoperabilidade e a eficiência na troca de dados. "Por exemplo, uma aplicação de frontend do Windows envia e recebe dados de um servidor de backend do Linux, mas os usuários da aplicação desconhecem a diferença."(AMAZON, 2023b)

Em um cenário onde a diversidade tecnológica é a presente, o Middleware assume uma posição central ao facilitar a construção de sistemas distribuídos coesos. Um exemplo prático dessa integração é a utilização do protocolo MQTT para a troca eficiente de mensagens entre dispositivos, aliado à implementação de clientes API, utilizando protocolo HTTP para o envio de comandos, proporcionando interfaces padronizadas para diferentes linguagens de programação. Essa combinação estratégica não apenas ilustra a versatilidade do Middleware, mas também ressalta como tecnologias como MQTT e clientes API podem ser integradas de maneira sinérgica para otimizar a comunicação entre componentes diversos. Essa sinergia contribui substancialmente para a eficácia e flexibilidade na integração de diferentes tecnologias em um ambiente distribuído.

3.4.1 Protocolo MQTT

O protocolo MQTT (Message Queuing Telemetry Transport) é uma tecnologia essencial para a comunicação eficiente em ambientes IoT (Internet of Things), proporcionando uma solução robusta para a troca de mensagens entre dispositivos conectados. Baseado em um modelo de publicação e subscrição, o MQTT simplifica a comunicação assíncrona, permitindo que dispositivos enviem e recebam mensagens de forma eficaz, independentemente de suas disparidades em termos de linguagens de programação e capacidades de hardware.

Ao adotar o MQTT, é possível contar com uma infraestrutura sólida e padronizada, que facilita a integração de dispositivos heterogêneos, promovendo uma comunicação eficiente e escalável em ambientes diversificados. (AMAZON, 2023c)

O MQTT é um protocolo que atua na camada de aplicação em uma rede. A informação navega entre a fonte (o emissor) e o broker (o receptor) e a implementação

de mecanismos de segurança durante esta comunicação é essencial para a construção de um sistema isolado e seguro.

3.4.2 API - Application Programming Interface

As APIs (Interfaces de Programação de Aplicações) desempenham um papel fundamental na interconexão de sistemas e serviços, oferecendo um conjunto de regras e protocolos que permitem a comunicação eficiente entre diferentes aplicações. A padronização proporcionada pelas APIs simplifica a integração de funcionalidades e dados, independentemente das linguagens de programação ou das arquiteturas de software utilizadas. No contexto de desenvolvimento de software, as APIs são elementos-chave para alcançar a interoperabilidade, possibilitando a interação entre sistemas heterogêneos. Ao adotar boas práticas de design e implementação de APIs, desenvolvedores podem criar interfaces eficazes, promovendo a modularidade, a reusabilidade de código e facilitando a integração de novos componentes. A ênfase na importância das APIs torna-se ainda mais evidente ao considerar seu papel essencial na construção de ecossistemas digitais interconectados e na promoção da inovação tecnológica.

APIs utilizam o protocolo HTTP para efetuarem requisições utilizando URLs. O Protocolo de Transferência de Hipertexto (HTTP - Hypertext Transfer Protocol) é um protocolo de comunicação utilizado na internet para a transferência de informações, como texto, gráficos, áudio, vídeo e outros arquivos multimídia. O HTTP opera no modelo cliente-servidor, onde um cliente envia solicitações (geralmente em formato de URLs) e o servidor responde com os recursos solicitados. (AMAZON, 2023a)

O protocolo HTTP (Hypertext Transfer Protocol) é um padrão de comunicação utilizado para transferência de dados na World Wide Web. Funcionando na camada de aplicação, o HTTP permite a troca de informações entre um cliente, como um navegador web, e um servidor web. A comunicação ocorre por meio de requisições enviadas pelo cliente e respostas retornadas pelo servidor. A implementação de medidas de segurança, como SSL/TLS, é crucial para garantir a integridade e a confidencialidade dos dados durante a transmissão. Com sua ampla adoção e flexibilidade, o HTTP desempenha um papel fundamental na interoperabilidade e acessibilidade da web moderna.

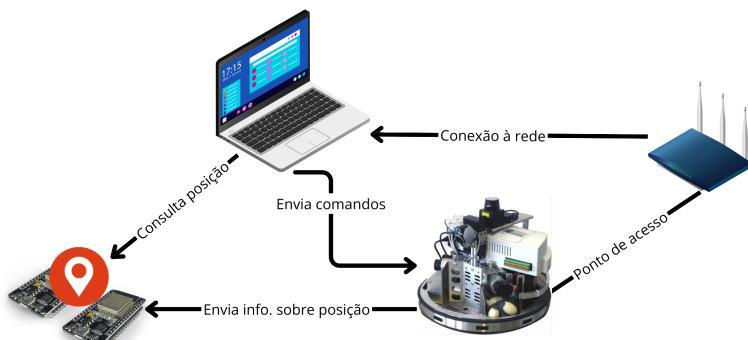
4 METODOLOGIA

Ao decorrer deste capítulo, as atividades desenvolvidas durante a realização do projeto são apresentadas, demonstrando como a comunicação e configuração dos sistemas listados durante a fundamentação teórica foram estudadas com objetivo de alcançar os resultados esperados.

4.1 Comunicação com Robotino

O Robotino é um robô com um computador integrado e sistema operacional Linux, apresentando softwares próprios e APIs para programação. Os softwares próprios oferecem interface gráfica intuitiva, enquanto as APIs são compatíveis com C, C++ e Java. Para alcançar o objetivo final, o diagrama abaixo apresenta o papel da interface, que deve ser capaz de receber comandos externos (contendo pacotes de dados).

Figura 12 – Diagrama de comunicação



Fonte: Autor (2023).

A interface deve estar hospedada no Robotino e ser capaz de trocar dados tanto com os dispositivos ESP32, quanto com o Defense IA, assim é possível configurar um Middleware para gerenciar esse fluxo de informações a fim de controlar o robô a partir de gatilhos externos.

4.1.1 Robotino View 2

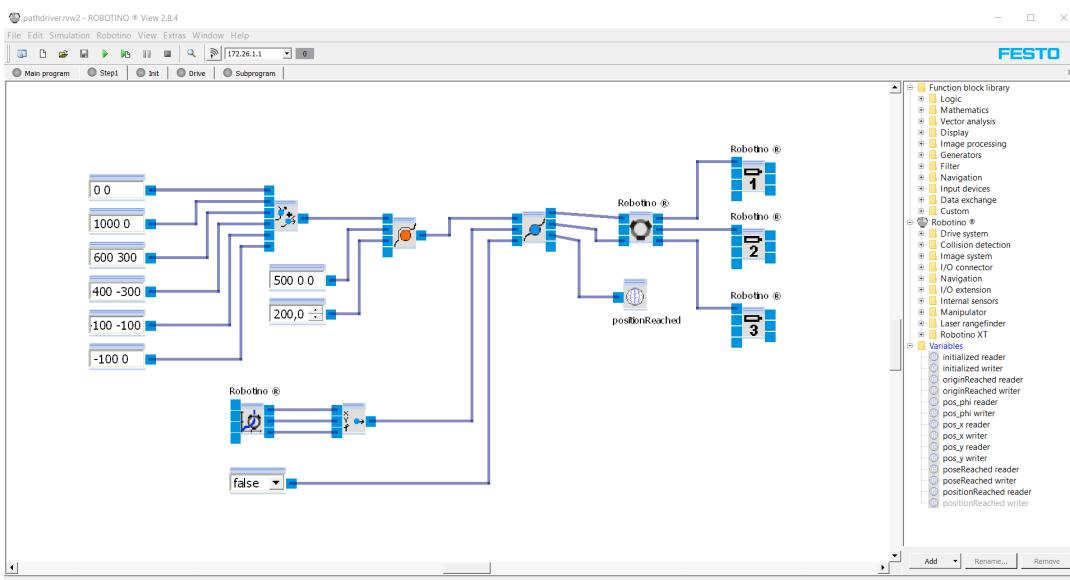
O Robotino View 2 é um software desenvolvido pela Festo Didactic para programação de robôs da linha Robotino. A aplicação oferece uma interface gráfica intuitiva e fácil de usar, permitindo a criação de programas complexos com um baixo grau de dificuldade.

O software apresenta uma ampla gama de blocos de programação, que abrangem as mais variáveis funções; blocos para cálculos matemáticos, sistemas

lógicos computacionais, variáveis globais, acesso a sensores e atuadores, hierarquia entre programas, são algumas funcionalidades disponíveis para programação utilizando o Robotino View 2.

A imagem apresentada a seguir mostra um programa para navegação do Robotino durante um caminho pré-definido, isto é feito utilizando blocos disponíveis como, bloco de composição de caminho (permite configuração de até 6 coordenadas para formar um caminho), bloco de obstáculo (define um obstáculo a ser evitado durante o caminho) e blocos de navegação (blocos que acessam motores e a odômetria).

Figura 13 – Robotino View 2: Programa para navegação



Fonte: Autor (2023).

A interface é extremamente versátil e útil para criar programas complexos, uma vez que tem acesso a todos os sensores e atuadores do robô, além de apresentar várias funções matemáticas e de navegação. É uma ótima opção para prototipagem, uma vez que apresenta integração com outros softwares da Festo Didatic, como Robotino SIM e Robotino Factory.

Um grande obstáculo que o software apresenta é a falta de documentação sobre sua API e como enviar e receber dados pela plataforma, isso aumenta o grau de complexidade ao utilizá-lo na topologia desejada.

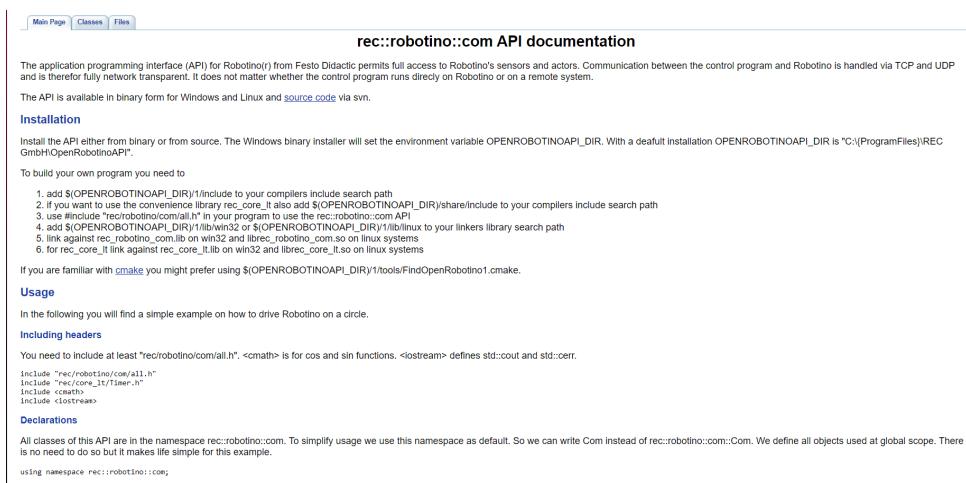
4.1.2 Open Robotino API

No diretório local do sistema operacional do Robotino (Linux Ubuntu), há disponível duas versões de API. A API 1 apresenta bibliotecas de códigos em C, C++ e Java, contendo alguns exemplos e clients para acesso à funcionalidades, como controle

remoto, visualização de imagem da web-cam ao vivo, navegação em círculos, entre outros.

A API 1 está toda documentada utilizando doxygen, uma biblioteca para gerar documentação de códigos, permitindo a análise e estudo dos códigos que acessam os sensores e atuadores do Robotino de maneira didática. À seguir é apresentada a página inicial em .html da documentação da API 1.

Figura 14 – API 1: Documentação em Doxygen



Fonte: Autor (2023).

A documentação traz exemplos utilizando a linguagem de programação C++, apresentando como realizar e finalizar a conexão com o robô, além de descrever todas as classes e arquivos que compõem os códigos de gerenciamento de seus atuadores e sensores.

4.1.3 Open Robotino API 2

A API 2 é uma arquitetura mais moderna comparada à API 1. Ela não contém exemplos ou documentação integrada aos códigos, mas sua estrutura é mais organizada e eficiente.

A API 2 é baseada no protocolo SOAP, que é um padrão de comunicação para serviços web. O SOAP é um protocolo baseado em XML que utiliza o formato WSDL para descrever a estrutura de suas solicitações. Este protocolo funciona da seguinte forma:

1. Um cliente envia uma solicitação HTTP para o servidor;
2. O servidor recebe a solicitação e processa-a;
3. O servidor envia uma resposta HTTP para o cliente.

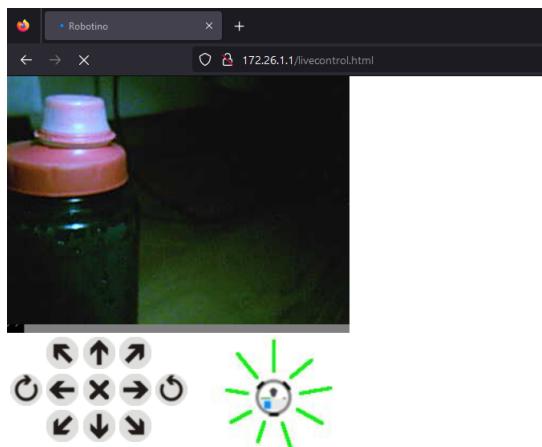
A solicitação HTTP é composta por um cabeçalho e um corpo. O cabeçalho contém informações sobre a solicitação, como o tipo de mensagem, o método e o URI do serviço web. O corpo contém os dados da solicitação.

A resposta HTTP também é composta por um cabeçalho e um corpo. O cabeçalho contém informações sobre a resposta, como o código de status HTTP e o tipo de mensagem. O corpo contém os dados da resposta.

4.1.3.1 Servidor web

A estrutura da API 2 é disponibilizada num servidor web instalado no Robotino, alocado na porta padrão web 80; essa porta possui uma interface onde é possível acessá-la, enviando e recebendo solicitações HTTP.

Figura 15 – Interface Web - Robotino



Fonte: Autor (2023).

A interface de usuário é dividida entre 3 quadros, um espaço para visualização da web-cam conectada ao robô, apresentando imagens em formato .jpg que são atualizadas a uma taxa de 15 FPS. Um segundo espaço apresenta o estado dos sensores de proximidade e colisão do Robotino, enquanto o terceiro quadro contém botões para controle da navegação do Robô.

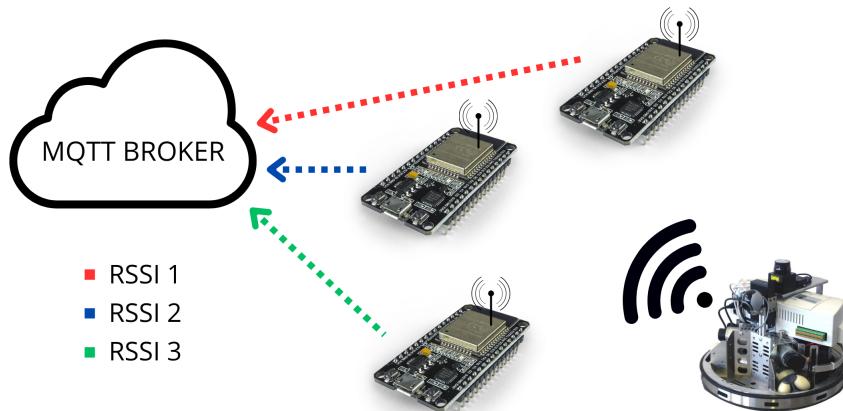
Estes comandos são enviados pela própria interface web, a partir de funções descritas em Javascript. São limitados e não possuem camadas de segurança, tampouco estrutura para troca de dados.

Para utilizar a interface de programação da API 2 para acessar os módulos do Robotino a partir da porta 80, deve-se realizar uma autenticação e manter a conexão utilizando serviços ativos. Durante o próximo capítulo será explicado como a conexão com a API 2 é realizada e como seus serviços são consumidos.

4.2 Configuração de dispositivos ESP32

Um sistema de posicionamento local será desenvolvido utilizando parâmetros de rede, como a intensidade do sinal entre o robô e dispositivos âncoras, para que o Robotino consiga navegar para uma direção conhecida.

Figura 16 – Captura de intensidade do sinal do Robotino



Fonte: Autor (2023).

Para a implementação do sistema de posicionamento, foram escolhidas as ESP32 como dispositivos âncoras devido à sua capacidade de integração com outros dispositivos através de uma rede wireless. As ESP32s são microcontroladores versáteis que oferecem suporte a Wi-Fi e Bluetooth, tornando-as ideais para a comunicação e a medição de intensidade do sinal (RSSI) entre elas e o Robotino. A imagem acima apresenta o fluxo de informação do sistema de posicionamento, onde dispositivos ESP32 conectam-se na (Wireless Local Area Network) do Robotino, cada dispositivo é responsável por capturar a intensidade do sinal recebido (RSSI) e o enviar a um broker MQTT.

4.2.1 Medição de RSSI

A medição da intensidade do sinal (RSSI) é um componente essencial deste sistema de posicionamento. As ESP32s serão configuradas para medir o RSSI do sinal de Wi-Fi emitido pelo Robotino. O RSSI é uma métrica que indica a potência do sinal recebido e é fundamental para determinar a proximidade do robô em relação às âncoras. A partir dessas medições, será possível estimar a posição do Robotino em relação às ESP32s e assim construir um sistema de localização interno para uma navegação orientada.

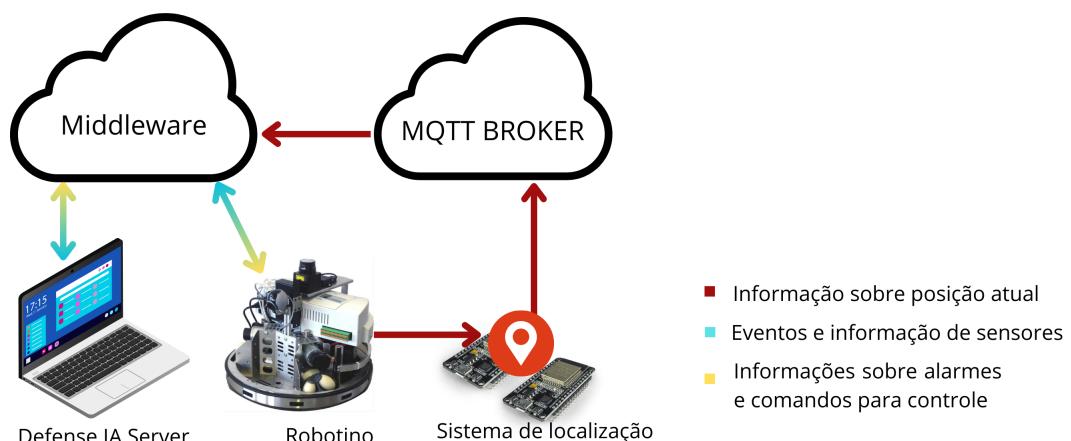
4.2.2 Comunicação via Broker MQTT

Para consolidar as medições de RSSI e permitir que o Robotino tome decisões com base nessas informações, será implementado um sistema de comunicação via MQTT (Message Queuing Telemetry Transport). Cada ESP32 atuará como um cliente MQTT que envia as leituras de RSSI para um broker MQTT centralizado. Isso permitirá que o Robotino acesse as informações de localização de forma eficiente e em tempo real.

4.3 Integração de sistemas

No âmbito deste estudo, a integração com o software de segurança eletrônica, Defense IA, é um componente fundamental. Conforme estabelecido pela estrutura lógica do software, é imperativo a implementação de um middleware para permitir o fluxo de dados entre a plataforma do Defense IA e os demais sistemas envolvidos, que incluem o Robotino e o sistema de localização interno. A Figura a seguir ilustra o esquema de intercâmbio de dados entre esses sistemas e o Defense IA por meio do Middleware.

Figura 17 – Diagrama de comunicação de sistemas



Fonte: Autor (2023).

Para atender às exigências do projeto, que incluem a utilização de elementos como clientes de broker MQTT, API SOAP, e de API REST, a escolha da linguagem de programação é de extrema importância. Com base em sua notável capacidade de desempenho, que inclui a capacidade de executar múltiplas threads simultaneamente, bem como sua rica coleção de bibliotecas destinadas à integração entre sistemas, optou-se pela linguagem GoLang para o desenvolvimento do middleware.

A escolha da GoLang é justificada pelo seu desempenho de alto nível e a capacidade de manipular eficientemente a comunicação entre os sistemas envolvidos.

Sua eficácia no gerenciamento de threads e recursos torna-a uma escolha ideal para as demandas de integração entre sistemas complexos como os que compõem este projeto. Além disso, a linguagem GoLang oferece uma ampla gama de bibliotecas e ferramentas prontamente disponíveis, o que facilitará significativamente o processo de desenvolvimento e manutenção do middleware. Essas características fazem da GoLang a linguagem ideal para atender aos requisitos da integração de sistemas proposta neste estudo.

4.3.1 Integração com o Sistema de Localização

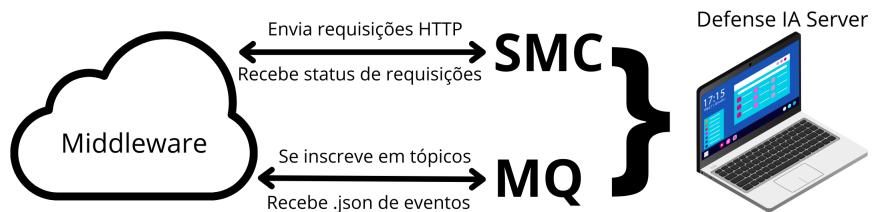
Como mencionado anteriormente, as informações sobre a intensidade do sinal entre o Robotino e âncoras estrategicamente posicionadas serão transmitidas através de um broker MQTT. Esses dados serão processados pelo Middleware para referenciar a posição do Robotino no espaço.

Com o objetivo de simplificar o desenvolvimento, o Middleware não deve incorporar algoritmos de navegação complexos, mas sim adotar uma lógica de referenciamento simples, utilizando apenas um eixo espacial como trajetória de navegação. Essa abordagem permitirá validar a capacidade de navegação autônoma do Robotino a partir do acionamento remoto por eventos do Defense IA.

4.3.2 Integração com Defense IA

Na discussão sobre a integração com o Defense IA, é crucial entender a estrutura modular do software, destacando-se por uma ampla variedade de funções e comandos. A arquitetura cliente-servidor do Defense IA reforça a necessidade de uma API para facilitar a comunicação entre cliente e servidor. A API, baseada no modelo RESTFUL, converte cada solicitação do cliente em um comando correspondente, utilizando o protocolo HTTP para efetuar a comunicação de dados com os serviços da plataforma. A comunicação via comandos API desempenha um papel crucial na interoperabilidade e na capacidade de integrar funcionalidades ao Defense IA, aspectos fundamentais abordados na metodologia.

A integração entre outros sistemas e o software Defense IA ocorre em duas etapas distintas, implementadas durante o desenvolvimento do middleware. Uma etapa envolve o uso de um cliente HTTP para enviar e receber informações do serviço de gerenciamento do sistema do Defense IA. A estruturação de informações, conforme representada na imagem a seguir, permite gerar outputs desejados (eventos) no sistema.

Figura 18 – Integração com Defense IA

SMC (Service Management Platform) - Serviço de gerenciamento da plataforma Defense IA
MQ (Message Queue) - Serviço de notificações por push MQ de eventos que ocorrem na plataforma

Fonte: Autor (2023).

Já a segunda etapa, um cliente MQTT deve ser desenvolvido com objetivo de conectar-se ao broker próprio do Defense IA, desta forma, é possível acompanhar o fluxo de todos os eventos CFTV ou digitais que ocorrem durante a plataforma.

Ao final das duas etapas, será então possível receber e publicar eventos ao software de segurança eletrônica de forma automática a partir de comandos do Middleware.

5 DESENVOLVIMENTO

Ao decorrer deste capítulo, pontos críticos sobre o desenvolvimento do projeto são abordados, expondo suas dificuldades e justificando ações tomadas visando os objetivos específicos.

5.1 Controle do Robotino remotamente

Após ligado, ao inicializar, o sistema operacional do Robotino habilita uma WLAN (Wireless Local Area Network) própria, permitindo conexões externas a seu sistema via Wi-Fi, também é possível conectar-se diretamente a sua interface de rede através de um conector rj45 presente. Ao conectar-se à rede sem fio, a partir de um computador é possível acessar o sistema do Robô; o Robotino permite conexões via protocolos como ssh e telnet para execução de comandos internos remotamente.

Figura 19 – Acesso remoto ao Robotino via ssh

```
robotino@robotino: ~
C:\Users\pedrossh robotino@172.26.1.1
robotino@172.26.1.1's password:
Linux robotino 2.6.28.9.robotino-rtai-3.7.1-gcc-4.3 #1 Thu Apr 15 09:12:45 CEST 2010 i586

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

To access official Ubuntu documentation, please visit:
http://help.ubuntu.com/
Last login: Wed Jan  1 01:01:12 2003 from 172.26.1.100
robotino@robotino:~$ pwd
/home/robotino
robotino@robotino:~$ -
```

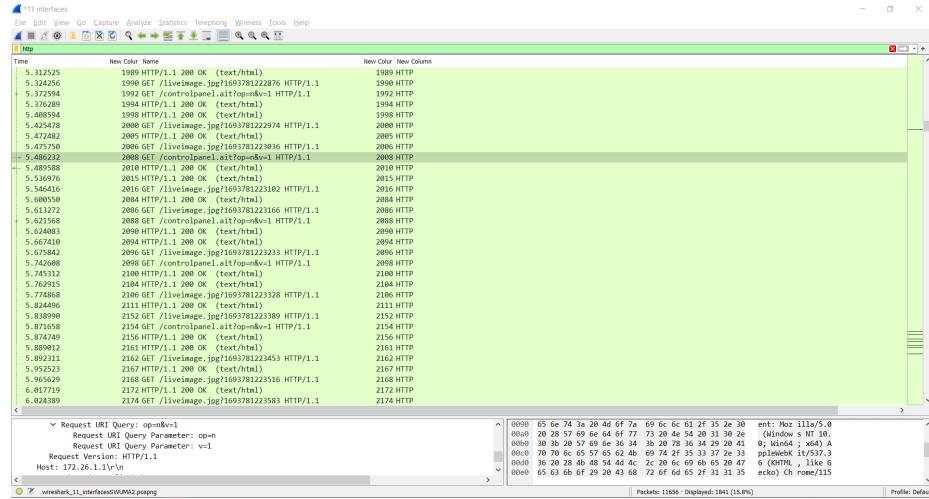
Fonte: Autor (2023).

Assim como apresentado no capítulo anterior, acessando seu sistema operacional é possível inicializar um servidor web nativo. À disposição há uma interface básica em html para troca de comandos básicos utilizando protocolo HTTP.

5.1.1 Análise de pacotes de comunicação

Estando conectado e comunicando-se com o Robô, é possível utilizar ferramentas de análise de tráfego de rede, como por exemplo *Wireshark* para analisar os pacotes de comunicação entre o servidor (Robotino) e a interface de usuário (Interface web). A figura abaixo apresenta a tela de captura de pacotes de rede do *Wireshark* durante comunicação entre o usuário e o Robotino.

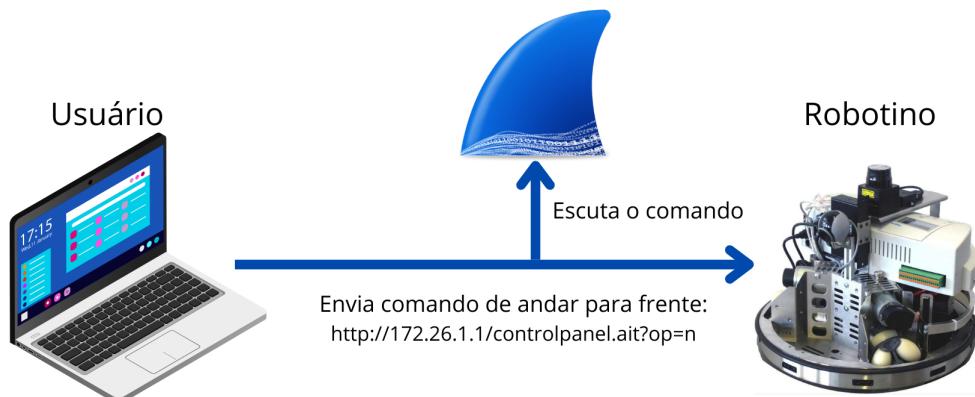
Figura 20 – Wireshark - Visualização de pacotes



Fonte: Autor (2023).

Rastreando os pacotes de comunicação, é possível colher os *endpoints* de cada comando, ou seja, o que foi pronunciado pelo usuário ao executar cada comando. Todos os comandos são uma requisição HTTP do tipo GET, inclusive comandos do painel de controle de navegação. A imagem a seguir exemplifica a captura de pacotes pelo *Wireshark*. A tabela abaixo lista todos os comandos encontrados a partir da análise de tráfego.

Figura 21 – Wireshark - exemplo



Fonte: Autor (2023).

Requisição	Comando	Ação
GET	/controlpanel.ait?op=x	Comando para cessar qualquer movimentação em motores
GET	/controlpanel.ait?op=n	Ativa os motores para navegar sentido norte
GET	/controlpanel.ait?op=s	Ativa os motores para navegar sentido sul
GET	/controlpanel.ait?op=w	Ativa os motores para navegar sentido oeste
GET	/controlpanel.ait?op=e	Ativa os motores para navegar sentido leste
GET	/controlpanel.ait?op=nw	Ativa os motores para navegar sentido noroeste
GET	/controlpanel.ait?op=ne	Ativa os motores para navegar sentido nordeste
GET	/controlpanel.ait?op=sw	Ativa os motores para navegar sentido sudoeste
GET	/controlpanel.ait?op=se	Ativa os motores para navegar sentido sudeste
GET	/controlpanel.ait?op=cl	Ativa os motores para girar sentido horário
GET	/controlpanel.ait?op=ccl	Ativa os motores para girar sentido anti-horário

Tabela 1 – Endpoints

Com a lista de comandos é possível executar comandos básicos e limitados a partir de um navegador, de maneira remota, desde que conectado ao servidor web do robotino, mas além disso, nota-se que tais comandos estão vinculados a funções presentes em bibliotecas e serviços alocados no servidor web. Estes arquivos podem ser acessados a partir do sistema operacional do Robotino.

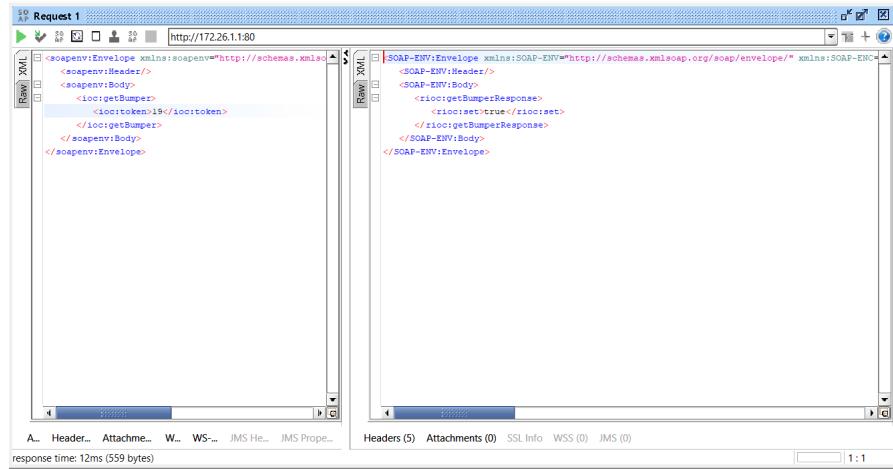
5.1.2 Acesso a funções de comando

Durante a análise da estrutura de arquivos do servidor web do Robotino, identificaram-se três arquivos .wsdl, conhecidos como Web Services Description Language (WSDL). Esses arquivos desempenham o papel de definir a estrutura e sintaxe necessárias para acessar os serviços do servidor do Robotino por meio de APIs SOAP, possibilitando a comunicação e o controle específico do robô.

A função principal do Middleware é estabelecer a comunicação com a API do Robotino, permitindo a troca de comandos. No entanto, para fins de teste e validação, utilizou-se o aplicativo de teste de serviço SoapUI como uma interface para enviar requisições à API. Esse aplicativo é capaz de importar arquivos .wsdl, interpretando e convertendo seu conteúdo em requisições para o servidor SOAP, facilitando a validação de comandos no protótipo. Abaixo, é apresentada uma solicitação referente ao estado

do sensor Bumper.

Figura 22 – Solicitação de estado do Bumper

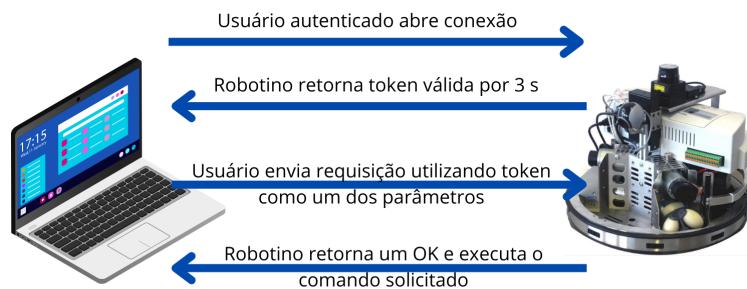


Fonte: Autor (2023).

À esquerda da imagem é possível visualizar o pacote estruturado da requisição, enquanto à direita, visualiza-se a resposta da solicitação. No caso, o sensor estava propositalmente obstruído para validação da resposta como "true".

Para efetuar requisições à API do Robotino, é necessário apresentar uma *token* gerada, um valor inteiro retornado ao abrir conexão com a API. Esta conexão é passível de uma camada de segurança com usuário e senha. O fluxo a seguir explica brevemente o processo de uma requisição à API.

Figura 23 – Fluxo de requisição à API SOAP do Robotino



Fonte: Autor (2023).

A *token* possui uma validade curta, expirando em poucos segundos. Para manter a conexão ativa e a token válida, o comando 'keepAlive' deve ser requisitado, renovando a conexão por mais alguns segundos, assegurando um fluxo contínuo de comunicação.

5.2 Construção de S.L.I. - Sistema de Localização Interno

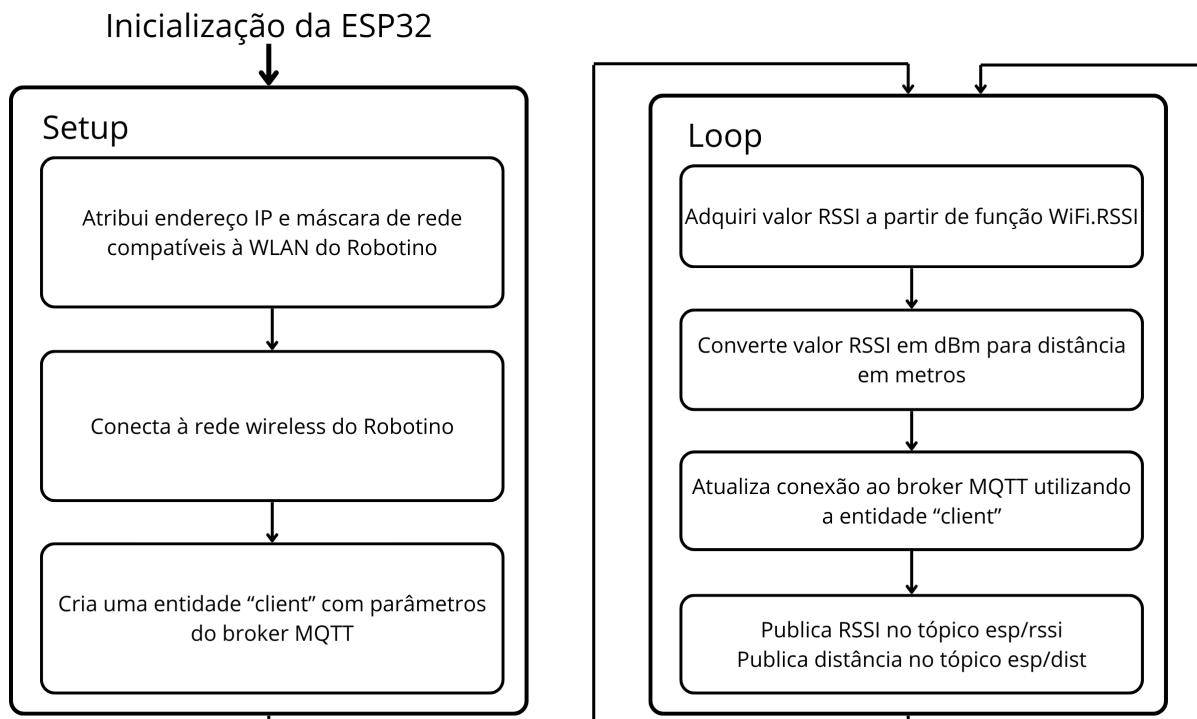
Durante etapas iniciais do projeto, o sistema de localização interno foi criado a partir da configuração de dois dispositivos ESP32, o objetivo desse sistema é criar uma referência espacial ao Robtino a partir da intensidade de conexão wireless entre os dispositivos e o robô.

Uma etapa crucial neste desenvolvimento é a aquisição do RSSI (*Received Signal Strength Indicator*), este valor é constantemente adquirido por cada ESP32 e continuamente atualizado em um broker MQTT.

5.2.1 Programação da ESP32

Configuradas de maneira uniforme através do ambiente de desenvolvimento Arduino IDE, cada ESP32 possui duas funções essenciais em seu sistema lógico: uma função de *setup* e uma função de *loop*. A primeira é acionada durante a inicialização do dispositivo, abrangendo operações como a conexão à rede sem fio e ao broker MQTT. A segunda função opera de forma contínua enquanto a ESP32 está ativa, realizando a aquisição do parâmetro RSSI, sua conversão para um valor em metros, e a publicação dos dados obtidos em tópicos específicos no broker MQTT. O fluxograma subsequente ilustra a estrutura programada em cada ESP32.

Figura 24 – Fluxograma de funções da ESP32



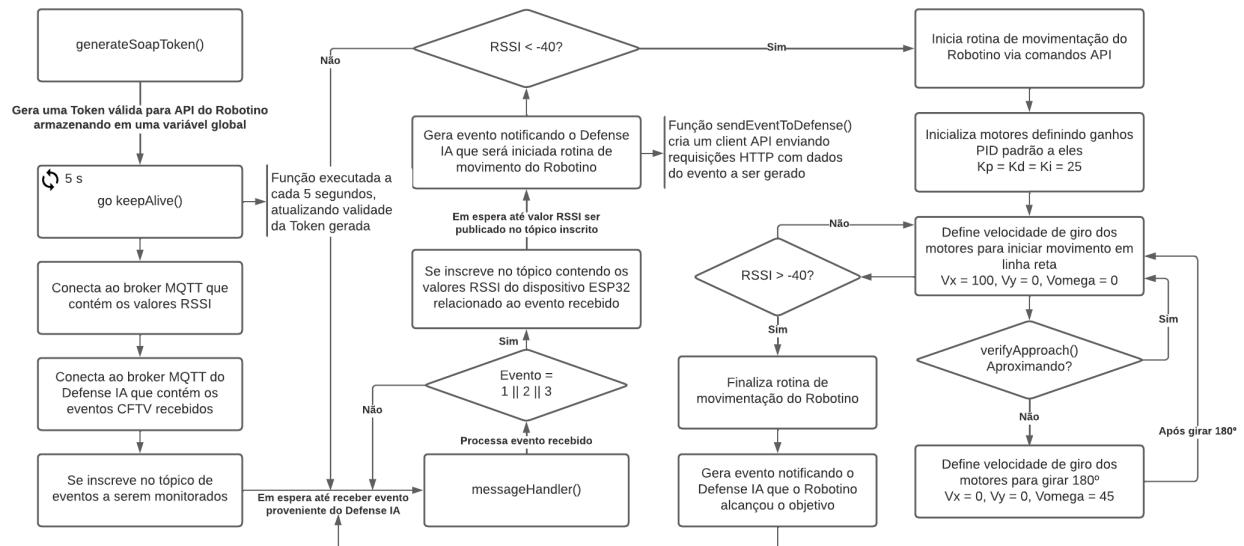
Fonte: Autor (2023).

As principais funções utilizadas provêm de bibliotecas do microcontrolador ESP32, como "WiFi.h", responsável por funções de conexão de rede e aquisição de RSSI, e "PubSubClient.h", responsável por funções de conexão e comunicação MQTT.

5.3 Desenvolvimento do Middleware

Desenvolvido utilizando a linguagem de programação Go, criada pela Google®, o papel do Middleware é receber todos os dados utilizados durante o sistema, tratá-los e distribuir comandos a partir de tais *inputs*. A figura abaixo apresenta a ordem de execução das rotinas, representando o comportamento do sistema.

Figura 25 – Fluxograma de rotinas do Middleware



Fonte: Autor (2023).

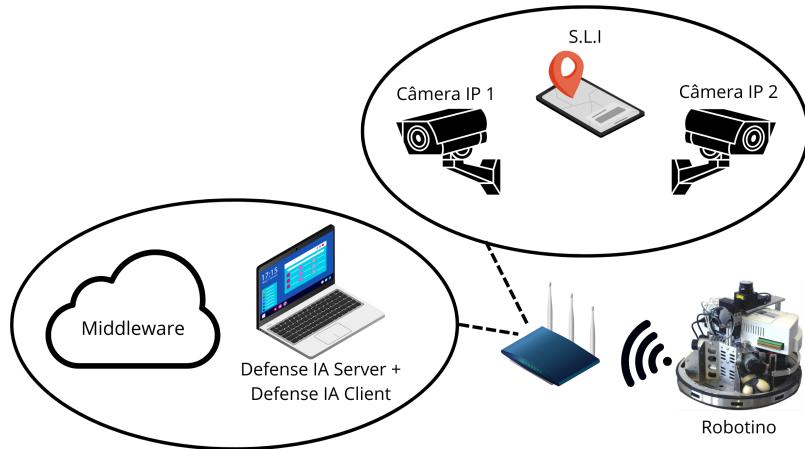
O Middleware inicia suas operações executando rotinas cruciais. Isso inclui a geração de uma token de conexão para possibilitar o acesso às funcionalidades do Robotino por meio de comandos API. Além disso, estabelece conexões com os brokers MQTT, do sistema de localização e do Defense IA. Essa abordagem permite a obtenção de informações vitais, como os valores RSSI provenientes das ESP32, bem como o acesso aos eventos registrados pela plataforma Defense IA por meio de câmeras de monitoramento.

Ao centralizar essas informações no Middleware, torna-se viável a criação de rotinas de navegação para o Robotino, baseadas no processamento lógico desses dados. Essa integração estratégica representa um ponto crucial no desenvolvimento, permitindo a otimização da tomada de decisões e aprimorando a eficiência operacional do sistema como um todo.

5.4 Inteligência Perimetral

A fim de gerar notificações de eventos ao software de segurança eletrônica, Defense IA, um cenário simples de CFTV foi construído, representando uma aplicação real de inteligência perimetral. O cenário construído é composto pelo Servidor do Defense IA, um cliente do Defense IA, câmeras IP de monitoramento e um roteador de rede.

Figura 26 – Diagrama de cenário CFTV



Fonte: Autor (2023).

Todos os dispositivos devem se conectar ao roteador, que também deve estar endereçado na mesma rede que o Robotino, o sistema de localização interno e o Middleware, desta forma é possível configurar as câmeras de monitoramento para enviar eventos de CFTV para notificação no Defense IA e direcioná-los ao Middleware.

Para o protótipo, o sistema de localização interno (S.L.I.) foi vinculado às câmeras, no qual um dispositivo ESP32 representa a posição de uma câmera IP. O Middleware é executado na mesma máquina onde executa-se o servidor do Defense IA e o cliente do Defense IA. Todos os dispositivos conectam-se à WLAN fornecida pelo Robotino.

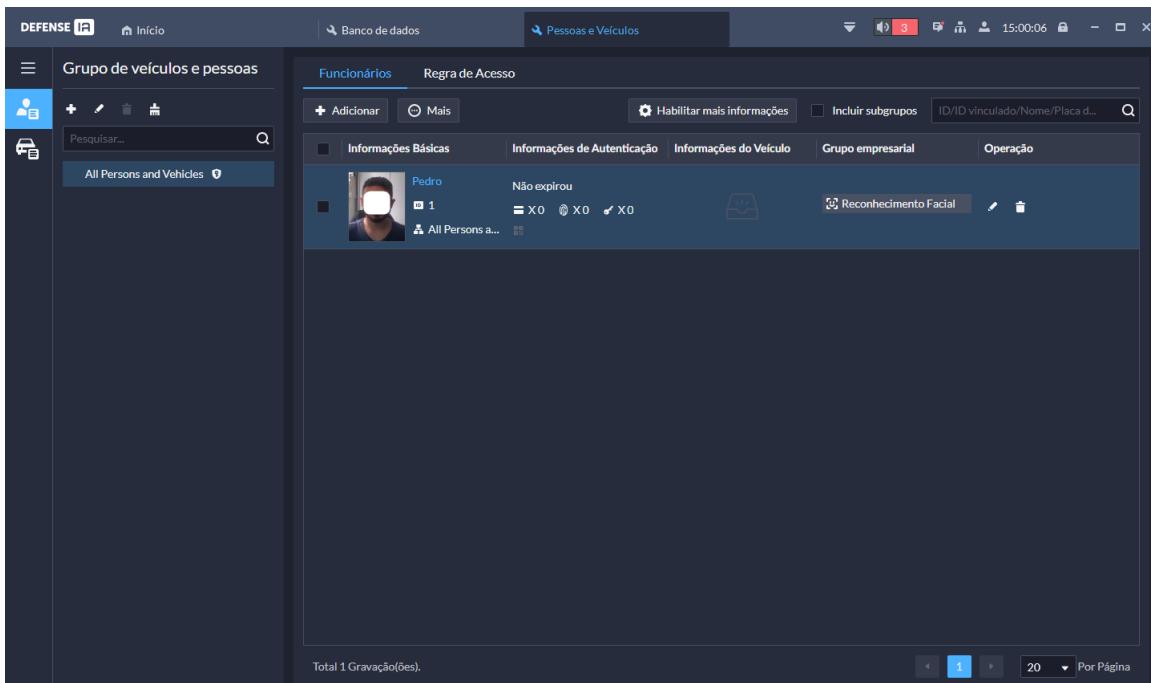
5.4.1 Cadastro e Configuração de Eventos

Eventos recebidos exigem um cadastramento por parte do operador na plataforma. Esse processo permite que a plataforma identifique qual inteligência específica deve ser acionada em resposta a determinados eventos provenientes dos dispositivos conectados. A figura a seguir demonstra o fluxo de eventos entre uma câmera IP e o Defense IA.

Figura 27 – Diagrama de notificação de evento no Defense IA

Fonte: Autor (2023).

No cliente Defense IA, é possível configurar um cenário em que, a partir de um reconhecimento facial específico, a plataforma notifique o operador. Para implementar esse cenário, a primeira etapa envolve o registro da face a ser monitorada, incorporando-a a um banco de dados facial nativo à plataforma. A figura abaixo apresenta uma face cadastrada no banco de dados do Defense IA.

Figura 28 – Pessoa cadastrada no Defense IA

Fonte: Autor (2023).

Nota-se que o cadastro relaciona-se a um grupo, "Reconhecimento Facial", o qual foi indicado durante o cadastro. Este grupo pode ser enviado ao dispositivo a fim de realizar a inteligência descrita, gerando um evento e enviando-o à plataforma.

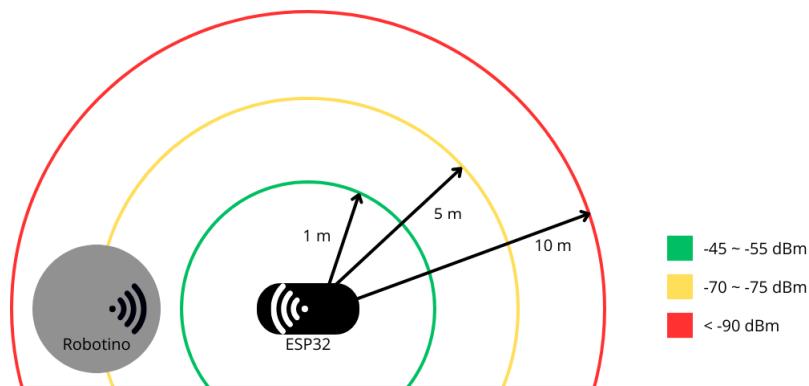
6 RESULTADOS

Neste capítulo são apresentados os resultados obtidos a partir da metodologia empregada, bem como a discussão dos mesmos após o desenvolvimento.

6.1 Sistema de localização interno

Composto por dois ESP32, um broker MQTT e o Robotino, o sistema de localização interno envolve a utilização de valores RSSI como referência para identificar a posição do robô em relação aos dispositivos ESP32. Identificando se o Robotino está próximo o suficiente ou não de um dispositivo ESP32, o sistema baseia-se em um valor RSSI de referência. O valor de referência foi definido como -40 dBm, compreendendo uma área com raio de menos de 1 metro em torno do dispositivo âncora. A figura abaixo demonstra a realação entre os valores adquiridos e a respectiva distância em metros entre o Robotino e um ESP32.

Figura 29 – S.L.I.



Fonte: Autor (2023).

O valor de RSSI não é linear e é afetado pelo ambiente ao seu redor, isso faz com que os dados adquiridos sejam variáveis dentro de uma faixa, assim como mostra a imagem. O sistema de localização interno não permite definir uma posição exata do robô e o dispositivo âncora. A tabela abaixo apresenta valores de RSSI adquiridos durante 6 medições, distanciando o Robotino do ESP32 1 metro a cada medição.

Distância (m)	RSSI 1 (dBm)	RSSI 2 (dBm)	RSSI 3 (dBm)
1	-48	-55	-52
2	-55	-56	-60
3	-69	-62	-66
4	-69	-70	-73
5	-70	-70	-75
6	-82	-86	-88

Tabela 2 – Medições de valores RSSI ao longo de 6 metros

Os resultados apresentados na tabela foram adquiridos em um ambiente com baixa interferência (com baixa transmissão de ondas de rádio). Foi notado que ao ultrapassar de 6 metros os resultados começaram a apresentar um alto grau de incerteza, uma vez que os valores em dBm tornam-se instáveis.

Em linhas gerais, o sistema de localização em ambientes internos utilizando o sinal RSSI cumpriu seu papel. Apesar dos dados adquiridos apresentarem leves variações (insignificantes para o atingimento do objetivo), o sistema que utiliza informações provenientes de dispositivos âncoras consegue identificar com uma boa acertividade se o Robotino está aproximando-se ou afastando-se da referência.

Durante a discussão sobre os resultados do Middleware, será apresentado como os valores RSSI adquiridos são processados.

6.2 Middleware

A operação resume-se em um fluxo contínuo iniciado a partir do recebimento um evento proveniente do Defense IA pelo Middleware. Durante esse fluxo, caso o evento seja esperado, o Robotino é acionado para direcionar-se ao dispositivo responsável pela notificação. O sistema de localização interno é utilizado para constatar a aproximação do Robotino em relação ao dispositivo em questão, possibilitando a correção da direção de navegação caso necessário. Assim que o Robotino aproxima-se o suficiente, sua movimentação é cessada e um evento de validação, indicando o fim da rotina, é enviado à plataforma Defense IA. O Middleware é o elemento responsável por centralizar e tratar todos os dados envolvidos durante toda a operação.

A aplicação desenvolvida comprehende as seguintes funções e estruturas:

- Cliente MQTT para receber dados RSSI;
- Cliente MQTT para receber dados de eventos do Defense IA;
- Funções lógicas para tratamento de dados.
- Cliente API SOAP para utilizar funções de conexão a dispositivos do Robotino;

- Cliente API SOAP para utilizar funções de controle do Robotino;
- Cliente API REST para enviar eventos ao servidor do Defense IA;

A figura abaixo representa com três blocos genéricos como as funções listadas relacionam-se entre si a partir da arquitetura do Middleware.

Figura 30 – Blocos do Middleware



Fonte: Autor (2023).

O bloco de funções que compreendem a utilização do protocolo MQTT é responsável por realizar a conexão com os dois brokers utilizados, sua função principal é a aquisição de dados provenientes de elementos externos ao Middleware. O bloco central é responsável por realizar o processamento dos dados recebidos, é a parte lógica do Middleware. O terceiro bloco é responsável pela conexão às APIs, assemelha-se ao bloco MQTT, porém com a função principal de enviar os dados processados.

6.2.1 Clientes MQTT

São utilizados dois brokers para transferência de dados ao Middleware. Um broker é implantado a partir da instalação dos serviços do Defense IA, neste broker são publicados todos os eventos que ocorrem na plataforma de segurança eletrônica, ao conectar-se a ele é possível receber estes eventos.

Já o segundo broker, implantado em uma máquina virtual, compõe o S.L.I., neste broker são publicados os valores de RSSI capturados a partir dos dispositivos ESP32. A figura abaixo apresenta um trecho de código onde é realizada a definição de parâmetros e conexão a ambos os brokers MQTT.

Figura 31 – Conexão MQTT

```

340     clientID := "mqtt-client-id"
341     //Broker RSSI
342     broker := "tcp://172.26.1.100:1883"
343     //Topics RSSI
344     topics := [3]string{"esp_1/rssi", "esp_2/rssi", "esp_3/rssi"}
345     //Conexao ao broker RSSI
346     opts := MQTT.NewClientOptions().AddBroker(broker).SetClientID(clientID)
347     client = MQTT.NewClient(opts)
348     if token := client.Connect(); token.Wait() && token.Error() != nil {
349         log.Fatalf("Error connecting to RSSI MQTT broker: %v\n", token.Error())
350     }
351     //Broker Defense
352     defenseBroker := "mqtts://172.26.1.150:1884"
353     //Credenciais broker Defense
354     MQpass := "7POXEN13n4z92DIk"
355     MQuser := "admin"
356     //Topico Eventos
357     defTopic := "mq/alarm/msg/topic/1"
358     config := &tls.Config{
359         InsecureSkipVerify: true,
360     }
361     defOpts := MQTT.NewClientOptions().AddBroker(defenseBroker).SetClientID(clientID)
362     defOpts.SetTLSConfig(config)
363     defOpts.Password = MQpass
364     defOpts.Username = MQuser
365     defClient = MQTT.NewClient(defOpts)
366     //Conexao ao broker Defense
367     if token := defClient.Connect(); token.Wait() && token.Error() != nil {
368         log.Fatalf("Error connecting to Defense MQTT broker: %v\n", token.Error())
369     }

```

Fonte: Autor (2023).

Ambas as conexões são realizadas no início do código e retornam um erro caso a conexão não seja efetuada com sucesso.

6.2.1.1 Processamento de mensagens recebidas - Broker Defense IA

Logo após a conexão ao broker, o Middleware executa a função de inscrição ao tópico de eventos do Defense IA, já que o recebimento de uma notificação pela plataforma é requisito para a sequência do fluxo.

Ao receber um evento, a função *messageHandler* é executada, apresentando no terminal de console informações sobre o pacote recebido e processando o evento. A figura abaixo apresenta um trecho do código em que isso ocorre.

Figura 32 – Função *messageHandler*

```

86 func messageHandler(defClient MQTT.Client, event MQTT.Message) {
87     currentDefPayload = string(event.Payload())
88     fmt.Printf("Current payload: %s\n", currentDefPayload)
89     fmt.Printf("Topic: %s\n", event.Topic())
90
91     var evento payloadDefense
92
93     // Use json.Unmarshal to decode JSON into the struct
94     err := json.Unmarshal(event.Payload(), &evento)
95     if err != nil {
96         fmt.Printf("Error decoding JSON: %v\n", err)
97         return
98     }
99
100    // Evento facial -> num = 1 (ESP 1), vai ate a camera
101    if evento.Info[0].AlarmType == "100001" && evento.DeviceName == "Facial" {
102        num = 1
103        fmt.Println("num 1")
104        // Signal that a new message has arrived and can be printed
105        select {
106            case printch <- struct{}{}:
107                default:
108            }
109        }

```

Fonte: Autor (2023).

O pacote publicado no broker MQTT pelo Defense IA possui o formato .json, sendo necessário decodificá-lo para extrair informações necessárias; são essas, o tipo de evento e o nome do dispositivo que o notificou na plataforma.

A partir de tais informações, é possível filtrar o evento, verificando se este é um evento de interesse. Caso seja, o Middleware continua o fluxo, atribuindo um índice ao evento recebido. Este índice é utilizado para definir qual será o objetivo de navegação do Robotino.

Para cada índice passível de atribuição, há um dispositivo ESP32 relacionado a este de forma sequencial, a tabela a seguir exemplifica a lógica de atribuição utilizada.

Evento	Índice	Dispositivo
Reconhecimento facial proveniente de câmera X	1	ESP 1
Acionamento de botão proveniente da plataforma	2	ESP 2
Desconexão de rede de dispositivo Y	3	ESP 3

Tabela 3 – Atribuição de dispositivos a índices a partir de eventos

Considerando que cada ESP32 relacionado a um índice seja fisicamente

atrelado a uma câmera, ou local de interesse, isso torna o sistema perpetuamente, dentro de suas capacidades, expansível.

6.2.1.2 Processamento de mensagens recebidas - Broker RSSI

Seguindo a lógica de atribuição de eventos a dispositivos, o Middleware é capaz de se inscrever no tópico referente ao dispositivo ESP32 de interesse. De tal forma, o software adquiri acesso aos valores de RSSI, assim possuindo a informação necessária para comandar o Robotino à direção correta. Ao se inscrever em um tópico do broker RSSI, sempre que este receber um dado, a função apresentada na figura abaixo é executada.

Figura 33 – *onMessageReceived*

```

78 func onMessageReceived(client MQTT.Client, message MQTT.Message) {
79     //Armazena payload
80     currentPayload = string(message.Payload())
81     //Converte payload string -> int
82     payloadAsInt, err := strconv.Atoi(currentPayload)
83     fmt.Printf("%d", payloadAsInt)
84     if err != nil {
85         fmt.Printf("could not convert payload to integer.\n")
86         return
87     }
88     select {
89     case printCh2 <- struct{}{}:
90     default:
91     }
92 }
```

Fonte: Autor (2023).

A função *onMessageReceived* executa uma conversão da informação de RSSI recebida em formato string (um vetor de caracteres) para formato de número inteiro. Isso ocorre para que o valor seja comparado ao valor referência de proximidade (-40), também um número inteiro. Essa é a próxima etapa executada pelo Middleware.

Ao comparar o valor recebido com o valor referência, caso o robô não esteja próximo o suficiente do objetivo, ou seja, caso o RSSI recebido seja um número menor que -40 (quanto mais positivo, mais próximo o robô está da âncora), o robô inicia a rotina de movimentação pela função *move*, é nesta etapa em que o robô é controlado utilizando comandos de API.

6.2.2 Clientes API SOAP

Em seus arquivos locais, o Robotino apresenta 3 serviços disponíveis para consumação pela OpenRobotino API 2, estes serviços foram convertidos em arquivos .go para utilização no Middleware utilizando a biblioteca *gowsdl*:

- *DeviceManager*
- *IOControl*
- *Camera*

O serviço *DeviceManager* é responsável por estruturar funções de gerenciamento de dispositivos do sistema do Robotino, permitindo a criação e manutenção de conexões entre um cliente e um dispositivo do sistema. O serviço *IOControl* possui funções de acionamento e consulta de status de sensores, atuadores, entradas e saídas (digitais e analógicas) do robô. Já o terceiro serviço, *Camera*, possui funções de acesso e gerenciamento da web-cam conectada a ao Robotino, possibilitando a configuração e transmissão de imagens em tempo real. Este último serviço não foi utilizado durante o desenvolvimento do projeto.

6.2.2.1 Serviço de conexão - *DeviceManager*

O serviço *DeviceManager* apresenta as seguintes funções listadas como comandos:

Nome do comando	Descrição
List	Lista os dispositivos existentes no sistema do Robotino.
Open	Abre uma conexão API a um dispositivo especificado, retornando uma Token.
OpenAuthenticated	Abre uma conexão API com credenciais a um dispositivo especificado, retornando uma Token.
KeepAlive	Renova o tempo de validade da Token.
Close	Encerra a conexão API.

Tabela 4 – Funções *DeviceManager*

No Middleware foram utilizadas os comandos *Open* e *KeepAlive*. O comando *List* foi utilizado uma única vez para identificar o valor de Id que indica o dispositivo Robotino. As três figuras a seguir apresentam os trechos de código em que os comandos do *DeviceManager* foram utilizados para gerar e manter a Token necessária para requisitar comandos API ao Robotino.

Figura 34 – Funções da Token

```

335 func main() {
336     generateSoapToken()
337     go keepAlive()

```

Fonte: Autor (2023).

As funções *generateSoapToken* e *keepAlive* são as duas primeiras do Middleware, ou seja, logo quando este é inicializado, já há a geração de uma Token de acesso à API do Robotino, seguida da restauração constante de sua validade.

Figura 35 – Função *generateSoapToken*

```
134 func generateSoapToken() {
135     accessMode := devservice.EAccessSERVICEREADWRITE
136
137     // Create an instance of the Open request
138     openRequest := &devservice.Open{
139         Id:          2, // cam - 1  robotino - 2
140         AccessMode: &accessMode,
141     }
142
143     // Make the SOAP call to open the connection
144     openResponse, err := devservice.NewDeviceManagerPortType(devclient).Open(openRequest)
145     if err != nil {
146         log.Fatalf("Error opening connection to SOAP: %v", err)
147     }
148
149     // Process the response
150     token = openResponse.Token
151     fmt.Printf("SOAP Token: %d\n", openResponse.Token)
152 }
```

Fonte: Autor (2023).

A função *generateToken* comporta a requisição do comando API *open*. Como parâmetros da requisição, é necessário indicar um Id que refere-se ao dispositivo o qual deseja abrir a conexão e o modo de acesso. O valor de Id é igual a 2, indicando que deseja-se abrir conexão API ao Robotino e o modo de acesso indica que a conexão possui permissão de escrita e leitura.

Por fim, processa-se a resposta do comando API, salvando o valor da Token gerada em uma variável global e apresentando-a no terminal de console.

Figura 36 – Função *keepAlive*

```

154 func keepAlive() {
155     ticker := time.NewTicker(5 * time.Second)
156     defer ticker.Stop()
157
158     for {
159         select {
160             case <-ticker.C:
161                 //fmt.Println("Keep-alive: Sending a keep-alive signal...")
162                 keepAliveRequest := &devservice.KeepAlive{
163                     Token: token,
164                 }
165                 _, err := devservice.NewDeviceManagerPortType(devclient).KeepAlive(keepAliveRequest)
166                 if err != nil {
167                     fmt.Printf("Error keeping token alive: %v\n", err)
168                     break
169                 }
170             case <-exitCh:
171                 return
172         }
173     }
174 }
```

Fonte: Autor (2023).

A função *keepAlive* comporta a requisição do comando API *KeepAlive*. Como parâmetro da requisição, é necessário indicar o valor da Token a qual deseja renovar a conexão. Este comando não possui conteúdo em sua resposta. A função *keepAlive* foi configurada para ser executada a cada 5 segundos.

Com a Token sempre "viva" é possível utilizar comandos API durante a mesma sessão, reciclando a Token, sem a necessidade de gerar uma a cada nova requisição. A variável global token é utilizada durante todas as requisições do serviço *IOControl*.

6.2.2.2 Serviço de controle - *IOControl*

O serviço *IOControl*, utilizado em funções do Middleware para controlar os motores a partir de comandos API, apresenta as seguintes funções listadas como comandos:

Nome do comando	Descrição
getActualMotorVelocity	Adquire a velocidade dos motores separadamente
getActualVelocity	Adquire a velocidade total do Robotino
getAnalogInput	Adquire o valor do terminal analógico indicado
getBumper	Adquire o valor do status do <i>bumper</i> (<i>true ou false</i>)
getDigitalInput	Adquire o valor do terminal digital indicado
getDistanceSensor	Adquire o valor do status do sensor de distância indicado (<i>true ou false</i>)
getMotorConstants	Adquire o valor das constantes de controle dos motores.
getOdometry	Adquire o valor da odometria dos motores.
motorActualPosition	Adquire o valor do encoder de um motor específico.
motorCurrent	Adquire o valor da corrente de um motor específico.
numAnalogInput	Lista entradas analógicas.
numDigitalInput	Lista entradas digitais.
numDigitalOutput	Lista saídas digitais.
numDistanceSensors	Lista sensores de distância.
numMotors	Lista os motores.
resetPosition	Redefine a odometria a zero.
setBrake	Freia um motor específico.
setDigitalOutput	Define valor à saída digital.
setMotorConstants	Define valores Kp, Kd e Ki a um motor específico.
setMotorVelocity	Define velocidade em rpm a um motor específico.
setOdometry	Define uma posição específica à odometria.
setRelay	Altera o status de um terminal de contato seco.
setVelocity	Define velocidade em rpm ao Robotino.
systemCurrent	Adquire a corrente total em ampères consumida pela bateria.
systemVoltage	Adquire a tensão em volts atual da bateria.
update	Contém todos parâmetros de definição de valores.

Tabela 5 – Funções IOControl

Para o protótipo, foram utilizados os comandos *setMotorConstants* e *setVelocity*. Ambos os comandos foram utilizados na função *move*, que é executada durante a rotina de movimentação do Robotino.

A rotina de movimentação do Robotino consiste na inicialização dos motores, seguido do acionamento destes, movimentando o robô em linha reta para frente. Há constantemente uma verificação da aproximação do Robotino ao alvo, caso esteja se afastando, executa um giro de 180º, encarando a direção oposta, e assim, prossegue em linha reta para frente novamente; este fluxo é executado até o robô chegar ao objetivo.

As figuras abaixo apresentam os trechos de código que representam a rotina de movimentação do Robotino.

Figura 37 – Função move

```

222 func move(obj int) {
223     //indices de afastamento e aproximacao
224     afastando = 0
225     aprox = 0
226     //initialize motors
227     if !motorsInit {
228         for i := 0; i < 3; i++ {
229             setMotorConstants := &iocservice.SetMotorConstants{
230                 Token: token, // Use the token received from devservice
231                 Motor: byte(i),
232                 Kp:    25,
233                 Kd:    25,
234                 Ki:    25,
235             }
236             _, err := iocservice.NewIOControlPortType(iocclient).SetMotorConstants(setMotorConstants)
237             if err != nil {
238                 log.Fatalf("Error setting motor %d constants: %v", i, err)
239             }
240         }
241         fmt.Printf("motors initialized.\n")
242         motorsInit = true
243     }
244     //zera contador de aproximacao
245     contApproach := 0

```

Fonte: Autor (2023).

Ao início da função *move*, há a declaração de duas variáveis (aproximando e afastando) que servem como índices utilizados em conjunto com a função de verificação de aproximação executada posteriormente. Em seguida, há a inicialização de motores, atribuindo valores às constantes de controle de cada motor (1, 2 e 3). Por padrão, é indicado pela Festo o valor de 25. Por fim, há o zeramento de uma variável auxiliar para contagem (contApproach), incrementa-se 1 a seu valor a cada verificação de aproximação como apresentado na figura abaixo.

Figura 38 – Movendo o Robotino para frente

```

256 //go forward
257 for {
258     // Construct the setVelocity struct
259     setVelocity := &iocservice.SetVelocity{
260         Token: token, // Use the token received from devservice
261         VX:    100,
262         VY:    0,
263         VOmega: 0,
264     }
265     // Send the setVelocity struct to set the velocity
266     _, err := iocservice.NewIOControlPortType(iocclient).SetVelocity(setVelocity)
267     if err != nil {
268         log.Fatalf("Error setting velocity: %v", err)
269     }
270     //Verifica se esta se aproximando ou afastando do alvo
271     verifyApproach(refPayload, payloadAsInt)
272     contApproach++
273     if contApproach > 30 {
274         if aprox > afastando {
275             fmt.Printf("Robo se aproximando")
276             afastando = 0
277             aprox = 0
278         }
279         if afastando > aprox {
280             fmt.Printf("Robo se afastando, efetuar manobra")
281             refPayload = payloadAsInt
282             afastando = 0
283             aprox = 0
284             girar180()
285         }
286     }
287 }
```

Fonte: Autor (2023).

Assim que variáveis auxiliares e os motores são inicializados, define-se a velocidade do Robotino como 100 mm/s para frente; por característica da API do Robotino, este comando deve ser executado incessantemente para que sua velocidade permaneça constante. Enquanto comanda-se o Robotino, ocorre a verificação se este aproxima-se ou afasta-se do objetivo pela função *verifyApproach*. Após 30 verificações de aproximação, executa-se a tomada de decisão de girar 180º ou permanecer à frente. A figura abaixo apresenta a lógica de verificação de aproximação.

Figura 39 – Função verifyApproach

```
197 // ref = reference cur = current
198 func verifyApproach(ref int, curr int) {
199     reff := float64(ref)
200     curr := float64(curr)
201     //afastando
202     if reff/curr < 1 {
203         afastando++
204         return
205     }
206     //aproximando
207     if reff/curr > 1 {
208         aprox++
209         return
210     }
211 }
```

Fonte: Autor (2023).

A função de verificação de aproximação têm como parâmetros um valor de referência que é adquirido ao início da rotina de movimentação e o valor de RSSI atual que é adquirido constantemente. Caso o quociente dos dois valores seja menor que 1, indica que o RSSI atual é um número maior que o RSSI de referência, indicando que o Robotino está mais distante, de tal forma, acrescenta-se 1 à variável índice "afastando", caso o contrário, acrescenta-se 1 à variável índice "aprox". Na Figura 38 é possível ver que após 30 execuções da função, ocorre a verificação de aproximação onde os índices são comparados, e caso o índice "afastando" seja maior que o índice "aprox", executa-se a manobra de giro.

Figura 40 – Função girar180

```

175 // girar o robo 180 graus
176 func girar180() {
177     i := 0
178     for i < 40 {
179         // Construct the setVelocity struct
180         setVelocity := &iocservice.SetVelocity{
181             Token: token, // Use the token received from devservice
182             VX: 0,
183             VY: 0,
184             VOmega: 45,
185         }
186         // Send the setVelocity struct to set the velocity
187         _, err := iocservice.NewIOControlPortType(iocclient).SetVelocity(setVelocity)
188         if err != nil {
189             log.Fatalf("Error setting velocity to spin 180: %v", err)
190         }
191         duration := 100 * time.Millisecond
192         time.Sleep(duration)
193         i++
194     }
195 }
```

Fonte: Autor (2023).

A função responsável por executar a manobra de giro executa o mesmo comando de API *setVelocity*, porém com parâmetros diferentes. Ao invés de definir 100 mm/s para frente, define-se 45 graus/s durante 40 vezes a cada 100 ms. Ou seja, a cada 100 ms, o Robô gira 4,5º, após 40 execuções, este terá alcançado um giro de 180º. Permitindo agora aproximar-se de seu objetivo, alcançando a última etapa da função *move*.

Figura 41 – Final da movimentação

```

286     //encerra o movimento caso proximo ao alvo (RSSI < -40)
287     if payloadAsInt > -40 {
288         //envia um evento ao defense indicando chegada
289         sendEventToDefense("11", "22")
290         fmt.Println("Robotino chegou ao objetivo.")
291         //finaliza loop de movimento
292         return
293     }
294     duration := 150 * time.Millisecond
295     time.Sleep(duration)
296 }
```

Fonte: Autor (2023).

Durante a última etapa da função de movimentação, compara-se o valor atual de RSSI à referência fixa. Caso a intensidade recebida seja maior que a referência

fixa de -40, finaliza-se o loop de movimentação, encerrando o envio de comandos aos motores e retornando a função principal do Middleware. Além disso, também há o envio de um evento à plataforma do Defense IA a partir da função *sendEventToDefense*, indicando que o Robotino chegou a seu alvo.

6.2.3 Cliente API REST - Defense IA

O desenvolvimento da comunicação via API do Defense IA possui duas funções principais, *sendEventToDefense* e *pushEventToDefense* a primeira é responsável por realizar a conexão com o servidor e o serviço que comporta a API de recebimento de eventos, chamando a segunda função que é responsável por estruturar o evento e executar o comando *http* específico contendo as informações necessárias. As figuras abaixo apresentam os trechos de código que contêm ambas as funções.

Figura 42 – Função *sendEventToDefense*

```

504 func sendEventToDefense(eventType string, eventSource string) {
505     //Informações de conexão ao Defense IA
506     defenseIP = "172.26.1.150"
507     defensePort = 8000
508     defenseAccessKey = "5mHo9a2713i9U9x2Bo5Qz4V9"
509     defenseSecretKey = "EdCORWY6N2hqv29W1630gYu="
510     nowTimestamp := time.Now()
511     nowTimestamp.UnixMilli()
512     data := []byte(`{
513         "accessKey": "` + defenseAccessKey + `",
514         "signature": "` + HMAC256(fmt.Sprintf("%d", nowTimestamp.UnixMilli()), defenseSecretKey) + `",
515         "timestamp": "` + fmt.Sprintf("%d", nowTimestamp.UnixMilli()) + `"
516     }`)
517     //Conexão ao Defense IA
518     url := fmt.Sprintf("http://%s:%d/ecos/api/v1.1/%s", defenseIP, defensePort, "account/authorize")
519     fmt.Printf("url: %s\n", url)
520     body, _ := defenseRequest(http.MethodPost, url, bytes.NewBuffer(data), "")
521     var defenseTokenResponse defenseTokenResponseStruct
522     json.Unmarshal(body, &defenseTokenResponse)
523     fmt.Printf("Body: %s\n", body)
524     //Envio de evento
525     pushEventToDefense(defenseTokenResponse.Data.Token, eventType, eventSource, "1233", "evento do robotino")
526 }
```

Fonte: Autor (2023).

A função *sendEventToDefense*, envia um comando *http* ao servidor do Defense IA, abrindo uma conexão autenticada e gerando uma token que é utilizada como parâmetro na função *pushEventToDefense*.

Figura 43 – Função *pushEventToDefense*

```

536 func pushEventToDefense(token string, eventTypeCode string, eventSourceCode string, eventId string, remark string) {
537     nowTimestamp := time.Now()
538     nowTimestamp.Unix()
539     //Estrutura pacote
540     data := []byte(`{
541         "eventId": "` + eventId + `",
542         "eventTime": ` + fmt.Sprintf("%d", nowTimestamp.Unix()) + `,
543         "eventSourceCode": "` + eventSourceCode + `",
544         "eventTypeCode": "` + eventTypeCode + `",
545         "remark": "` + remark + `"
546     }`)
547     //Comando para enviar evento
548     url := fmt.Sprintf("http://%s:%d/ecos/api/v1.1/%s", defenseIP, defensePort, "bridge/event/push")
549     body, err := defenseRequest(http.MethodPost, url, bytes.NewBuffer(data), token)
550     fmt.Printf("Body: %s\n", body)
551     if err != nil {
552         fmt.Printf("err: %s\n", err)
553     }
554 }
```

Fonte: Autor (2023).

A função *pushEventToDefense* estrutura o pacote com informações do evento a ser enviado à plataforma, seguido da execução do comando *http*, contendo tais informações estruturadas em seu *body*; de tal forma, enviando o evento ao Defense IA.

6.3 Acionamento do Robotino em operação

Com todas as funções estruturadas, é possível realizar a integração de todos os sistemas, uma vez que todas as tecnologias se comunicam utilizando o Middleware. O fluxo, executado em etapas é apresentado a seguir.

- Etapa 1:

Todos os sistemas, conectados à mesma rede, são inicializados.

O Middleware inicializado gera uma token de conexão ao Robotino e a mantém, esperando um evento proveniente do Defense IA.

- Etapa 2:

Evento é gerado na plataforma. Eventos cadastrados no Middleware:

1. Reconhecimento Facial;
2. Acionamento por software;
3. Desconexão de dispositivo.

Cada evento é vinculado a um dispositivo ESP32, estabelecendo 3 (três) pontos de interesse no espaço.

- Etapa 3:

Evento é recebido pelo Middleware e Robotino inicia rotina de movimentação.

- Etapa 4:

Robotino alcança objetivo e envia evento ao Defense IA.

O Middleware mantém a token de conexão com Robotino, esperando um novo evento proveniente do Defense IA.

6.4 Conclusão sobre resultados

Ao avaliar os objetivos delineados, observa-se que os resultados obtidos foram satisfatórios, alcançando todas as metas estabelecidas. Embora persistam itens passíveis de melhorias, o sistema demonstrou eficácia ao cumprir os requisitos predefinidos. Destaca-se o protagonismo do Middleware no contexto do protótipo, desempenhando a função crucial de integrar e conectar todas as tecnologias envolvidas. Essa peça central revelou-se como o elemento unificador, consolidando a harmonia entre as diferentes partes do sistema.

A comunicação do sistema ocorre por meio de uma rede wireless, e durante o desenvolvimento, identificou-se um gargalo significativo. Durante a rotina de navegação do Robotino, observou-se um volume elevado de dados sendo transmitidos pela rede sem fio, resultando em latência considerável no sistema. Como consequência, a execução da rotina de navegação não se mostrou fluida, frequentemente apresentando interrupções.

Possíveis melhorias ao protótipo são discutidas no capítulo seguinte.

7 CONSIDERAÇÕES FINAIS

Este projeto visou o desenvolvimento de um protótipo de integração entre sistemas utilizando diversas tecnologias e abordagens. O objetivo final não vincula-se a apenas o desenvolvimento aplicado. Além do aprofundamento dos aplicados, diferentes métodos e materiais poderiam ser utilizados para alcançar os objetivos. Alguns foram estudados e serão aqui explorados.

7.1 Comunicação com Robotino

A comunicação com os atuadores e sensores do Robotino foram realizados utilizando a OpenRobotinoAPI 2, considerada a opção mais atualizada e compatível com os objetivos. A API apresenta uma alta gama de opções ao comunicar-se com o Robotino, nem todas exploradas durante o desenvolvimento.

7.1.1 Latência na comunicação

Para abordar a problemática da latência na comunicação, sugere-se a inclusão de um roteador no sistema. Esse roteador pode assumir a responsabilidade de gerenciar e direcionar o fluxo de dados de todos os dispositivos, substituindo assim o papel do Robotino nesse aspecto específico. Essa medida pode otimizar a eficiência da comunicação, reduzir a latência e melhorar o desempenho geral da rotina de navegação.

7.1.2 Integração da web-cam

O Robotino apresenta conectado a seu computador uma web-cam compatível com a API disponível. No serviço web *camera* encontram-se comandos para execução de transmissões de vídeo à portas de rede virtuais do robô, com a exploração e aplicação de tais comandos, é possível adquirir a transmissão de vídeo pela rede utilizando os protocolos RTSP ou RTMP. Ambos estes protocolos são passíveis de conversão para ONVIF, protocolo de CFTV nativo ao Defense, permitindo a conexão do canal de vídeo do Robotino como um dispositivo ao Defense IA. Assim, permitindo acesso às capacidades de inteligências e integrações do software de vídeo monitoramento.

7.1.3 Segurança de dados durante comunicação

Com o intuito de tornar o protótipo mais robusto e eficiente, sugere-se a utilização de técnicas e rotinas de segurança de dados durante todos os pontos de comunicação do sistema. Para elevar o nível de segurança, é recomendável incorporar medidas específicas, começando pela implementação de protocolos de criptografia. A aplicação de algoritmos de criptografia assegurará a confidencialidade dos dados,

impedindo acessos não autorizados e protegendo informações sensíveis durante as transmissões pela rede. Tais rotinas foram implementadas à comunicação com o Defense IA, durante a conexão ao broker MQTT e à API REST. Já durante a comunicação entre o Middleware, o sistema de localização interno e o Robotino, não há medidas de segurança robustas o suficiente para tornar o sistema isolado.

A autenticação também desempenha um papel crucial na segurança do sistema. Um processo sólido de autenticação assegura que apenas usuários autorizados e dispositivos confiáveis possam interagir com o sistema.

A utilização de tais técnicas, como a criptografia de credenciais de acesso e protocolos de segurança durante a conexão trazem um nível adicional de robustez ao protótipo, fortalecendo sua resiliência contra potenciais ameaças. Ao incorporar medidas de autenticação, estabelece-se uma barreira protetora que restringe o acesso não autorizado, garantindo a integridade do sistema.

7.2 Navegação do Robotino

Os resultados do protótipo demonstraram eficazmente o acionamento do Robotino em resposta a eventos gerados pelo Defense IA, indicando uma integração bem-sucedida entre os sistemas. Contudo, para validar conceitualmente a ideia, restringiu-se a navegação do Robotino a apenas um eixo. Essa limitação, embora útil para os propósitos iniciais, não explora completamente o potencial do Robotino em termos de movimentação tridimensional.

Com o intuito de aprimorar a capacidade de navegação do Robotino em três eixos, seria vantajoso explorar a implementação de algoritmos de navegação. Utilizar algoritmos adequados permitiria ao Robotino calcular trajetórias eficientes em espaços tridimensionais, otimizando rotas em tempo real. Essa abordagem não apenas expandiria significativamente as capacidades do protótipo, possibilitando uma navegação mais versátil e adaptável, mas também elevaria a precisão e eficiência na execução de tarefas específicas.

Ao incorporar algoritmos de navegação, o protótipo se beneficiaria de uma abordagem flexível, permitindo sua adaptação a diferentes contextos e ambientes. Essa escolha estratégica não apenas enriqueceria a experiência de navegação do Robotino, mas também fortaleceria a utilidade prática do sistema, tornando-o mais eficaz em ambientes dinâmicos e complexos.

REFERÊNCIAS

- AMAZON. *api*. 2023. Disponível em: <https://aws.amazon.com/pt/what-is/api/>. 26
- AMAZON. *middleware*. 2023. Disponível em: <https://aws.amazon.com/pt/what-is/middleware/>. 25
- AMAZON. *mqtt*. 2023. Disponível em: <https://aws.amazon.com/pt/what-is/mqtt/>. 25
- BEELEN, T.; BERNARD, T.; PAPENMEIER, A. A robot control system for child-robot interaction. HMI - University of Twente, 2016. Disponível em: https://andrea.papenmeier.io/res/papers/2016_HMIP.pdf. 20
- BELLECIERI, Y.; JABOUR, F. C.; JABOUR, E. G. Localização indoor baseada na leitura bidirecional do rssi. IF Sudeste MG, 2016. Disponível em: https://www.academia.edu/66369578/Localiza%C3%A7%C3%A3o_Indoor_Baseada_na_Leitura_Bidirecional_do_RSSI. 24
- ESPRESSIF. *Datasheet - ESP32*. 2023. Disponível em: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32e_esp32-wroom-32ue_datasheet_en.pdf. 23, 24
- FESTO. *Robotino*. 2023. Disponível em: https://www.festo.com/br/pt/p/sistema-de-robo-movel-id_PROD_DID_8101344/?page=0. 20
- INTELBRAS. *Intelbras Líder*. 2023. Disponível em: <https://www.intelbras.com/pt-br/noticia/lider-brasileira-intelbras-expande-fronteiras-na-america-latina-marcando-presenca-em-evento.13>
- INTELBRAS. *Inteligência perimetral*. 2023. Disponível em: <https://www.intelbras.com/pt-br/linha-future>. 14
- INTELBRAS. *Manual Defense IA*. 2023. Disponível em: manuais.intelbras.com.br/manual-software-de-seguranca-eletronica-defense-ia/. 17
- KRACHT, S.; NIELSEN, C. *Robots in Everyday Human Environments*. 130 p. Dissertação (Mestrado em Sistemas autônomos inteligentes) — Aalborg University - AAU, 2007. Disponível em: <https://core.ac.uk/download/pdf/60678919.pdf>. 14
- RUIZ, A. M. Estudi d'estratègies de seguiment de línia per al robot mòbil robotino. UPC - Escola d'Enginyeria de Terrassa, 2016. Disponível em: <http://hdl.handle.net/2117/107423>. 20, 21, 22
- SHARP. *Datasheet - GP2D12*. 2023. Disponível em: https://engineering.purdue.edu/ME588/SpecSheets/sharp_gp2d12.pdf. 22

APÊNDICES

APÊNDICE A – CÓDIGO DO MIDDLEWARE

```

1 #include <stdio.h>
2 package main
3
4 import (
5     "bufio"
6     "bytes"
7     "crypto/hmac"
8     "crypto/sha256"
9     "crypto/tls"
10    "encoding/hex"
11    "encoding/json"
12    "fmt"
13    "io"
14    "log"
15    "net/http"
16    "os"
17    "os/signal"
18    "strconv"
19    "sync"
20    "time"
21
22    "cliente/devservice"
23    "cliente/iocservice"
24
25    MQTT "github.com/eclipse/paho.mqtt.golang"
26    "github.com/hooklift/gowsdl/soap"
27 )
28
29 var client MQTT.Client
30 var defClient MQTT.Client
31 var currentTopic string
32 var currentDefTopic string
33 var currentPayload string
34 var currentDefPayload string
35 var printCh = make(chan struct{})
36 var printCh2 = make(chan struct{})
37 var exitCh = make(chan struct{})
38 var exitCh2 = make(chan struct{})
39 var wg sync.WaitGroup
40 var num int
41 var token int32
42 var motorsInit bool = false
43 var afastando int
44 var aprox int
45
46 // const serviceURL = "http://172.26.1.1" // Endereço do SOAP
47 var devclient = soap.NewClient("http://172.26.1.1")
48 var iocclient = soap.NewClient("http://172.26.1.1")
49
50 type payloadDefense struct {
51     Method string `json:"method"`
52     Info   []struct {
53         DeviceCode      string  `json:"deviceCode"`
54         ChannelSeq     int     `json:"channelSeq"`
55         UnitType       int     `json:"unitType"`

```

```
56     UnitSeq          int      'json:"unitSeq"'
57     NodeType         string   'json:"nodeType"'
58     NodeCode         string   'json:"nodeCode"'
59     AlarmCode        string   'json:"alarmCode"'
60     AlarmStat        string   'json:"alarmStat"'
61     AlarmType        string   'json:"alarmType"'
62     AlarmGrade       string   'json:"alarmGrade"'
63     AlarmPicture     string   'json:"alarmPicture"'
64     AlarmDate        string   'json:"alarmDate"'
65     Memo              string   'json:"memo"'
66     ExtData           string   'json:"extData"'
67     LinkVideoChannels []any   'json:"linkVideoChannels"'
68     UserIds          []any   'json:"userIds"'
69     AlarmSourceName  string   'json:"alarmSourceName"'
70     RuleThreshold    int     'json:"ruleThreshold"'
71     StayNumber       int     'json:"stayNumber"'
72     PlanTemplateID  string   'json:"planTemplateId"'
73     DeviceName        string   'json:"deviceName"'
74     LinkedOutput      string   'json:"linkedOutput"'
75     MapIds            []string 'json:"mapIds"'
76 } 'json:"info"'
77 }
78
79 func onMessageReceived(client MQTT.Client, message MQTT.Message) {
80     currentPayload = string(message.Payload())
81     select {
82     case printCh2 <- struct{}{}:
83     default:
84     }
85 }
86
87 func messageHandler(defClient MQTT.Client, event MQTT.Message) {
88     currentDefPayload = string(event.Payload())
89     fmt.Printf("Current payload: %s\n", currentDefPayload)
90     fmt.Printf("Topic: %s\n", event.Topic())
91
92     var evento payloadDefense
93
94     // Use json.Unmarshal to decode JSON into the struct
95     err := json.Unmarshal(event.Payload(), &evento)
96     if err != nil {
97         fmt.Printf("Error decoding JSON: %v\n", err)
98         return
99     }
100
101    // Evento facial -> num = 1 (ESP 1), vai ate a camera
102    if evento.Info[0].AlarmType == "100001" && evento.Info[0].DeviceName == "Facial"
103    {
104        num = 1
105        fmt.Println("num 1")
106        // Signal that a new message has arrived and can be printed
107        select {
108            case printCh <- struct{}{}:
109            default:
110        }
111    }
112    // Evento softtrigger -> num = 2 (ESP 2), volta pra posicao inicial
```

```
113 if evento.Info[0].AlarmType == "11000001" && evento.Info[0].DeviceName == "Facial"
114     " {
115         num = 2
116         fmt.Println("num 2")
117         // Signal that a new message has arrived and can be printed
118         select {
119             case printCh <- struct{}{}:
120                 default:
121             }
122     }
123     // Evento desconexao -> num = 3 (ESP 3), vai ate o dispositivo ficticio
124     if evento.Info[0].AlarmType == "16" && evento.Info[0].DeviceName == "Teste" {
125         num = 3
126         fmt.Println("num 3")
127         // Signal that a new message has arrived and can be printed
128         select {
129             case printCh <- struct{}{}:
130                 default:
131             }
132     }
133
134 func generateSoapToken() {
135     accessMode := devservice.EAccessSERVICEREADWRITE
136
137     // Create an instance of the Open request
138     openRequest := &devservice.Open{
139         Id:           2, // cam - 1 robotino - 2
140         AccessMode:  &accessMode,
141     }
142
143     // Make the SOAP call to open the connection
144     openResponse, err := devservice.NewDeviceManagerPortType(devclient).Open(
145         openRequest)
146     if err != nil {
147         log.Fatalf("Error opening connection to SOAP: %v", err)
148     }
149
150     // Process the response
151     token = openResponse.Token
152     fmt.Printf("SOAP Token: %d\n", openResponse.Token)
153 }
154
155 func keepAlive() {
156     ticker := time.NewTicker(5 * time.Second)
157     defer ticker.Stop()
158
159     for {
160         select {
161             case <-ticker.C:
162                 //fmt.Println("Keep-alive: Sending a keep-alive signal...")
163                 keepAliveRequest := &devservice.KeepAlive{
164                     Token: token,
165                 }
166                 _, err := devservice.NewDeviceManagerPortType(devclient).KeepAlive(
167                     keepAliveRequest)
168                 if err != nil {
169                     fmt.Printf("Error keeping token alive: %v\n", err)
```

```
168         break
169     }
170     case <-exitCh:
171     return
172   }
173 }
174 }

// girar o robo 180 graus
176 func girar180() {
177   i := 0
178   for i < 40 {
179     // Construct the setVelocity struct
180     setVelocity := &iocservice.SetVelocity{
181       Token: token, // Use the token received from devservice
182       VX:    0,
183       VY:    0,
184       V0mega: 45,
185     }
186     // Send the setVelocity struct to set the velocity
187     _, err := iocservice.NewIOControlPortType(iocclient).SetVelocity(setVelocity)
188     if err != nil {
189       log.Fatalf("Error setting velocity to spin 180: %v", err)
190     }
191     duration := 100 * time.Millisecond
192     time.Sleep(duration)
193     i++
194   }
195 }
196 }

// ref = reference cur = current
198 func verifyApproach(ref int, cur int) {
199   reff := float64(ref)
200   curr := float64(cur)
201   //afastando
202   if reff/curr < 1 {
203     afastando++
204     return
205   }
206   //aproximando
207   if reff/curr > 1 {
208     aprox++
209     return
210   }
211 }
212 }

213 func move(obj int) {
214   //indices de afastamento e aproximacao
215   afastando = 0
216   aprox = 0
217   //armazena o valor de RSSI recebido no inicio do movimento e o converte para
218   //inteiro
219   refPayload, err := strconv.Atoi(currentPayload)
220   fmt.Printf("%d", refPayload)
221   if err != nil {
222     fmt.Printf("could not convert reference payload to integer.\n")
223     return
224 }
```

```
225 //initialize motors
226 if !motorsInit {
227     for i := 0; i < 3; i++ {
228         setMotorConstants := &iocservice.SetMotorConstants{
229             Token: token, // Use the token received from devservice
230             Motor: byte(i),
231             Kp:    25,
232             Kd:    25,
233             Ki:    25,
234         }
235         _, err := iocservice.NewIOControlPortType(iocclient).SetMotorConstants(
236             setMotorConstants)
237         if err != nil {
238             log.Fatalf("Error setting motor %d constants: %v", i, err)
239         }
240     }
241     fmt.Printf("motors initialized.\n")
242     motorsInit = true
243 }
244 //zera contador de aproximacao
245 contApproach := 0
246 //go foward
247 for {
248     // Construct the setVelocity struct
249     setVelocity := &iocservice.SetVelocity{
250         Token: token, // Use the token received from devservice
251         VX:    100,
252         VY:    0,
253         VOmega: 0,
254     }
255     // Send the setVelocity struct to set the velocity
256     _, err := iocservice.NewIOControlPortType(iocclient).SetVelocity(setVelocity)
257     if err != nil {
258         log.Fatalf("Error setting velocity: %v", err)
259     }
260
261     //converte todo RSSI recebido para inteiro
262     payloadAsInt, err := strconv.Atoi(currentPayload)
263     fmt.Printf("%d", payloadAsInt)
264     if err != nil {
265         fmt.Printf("could not convert payload to integer.\n")
266         return
267     }
268
269     //Verifica se esta se aproximando ou afastando do alvo
270     verifyApproach(refPayload, payloadAsInt)
271     contApproach++
272     if contApproach > 40 {
273         if aprox > afastando {
274             fmt.Printf("Robo se aproximando")
275             afastando = 0
276             aprox = 0
277         }
278         if afastando > aprox {
279             fmt.Printf("Robo se afastando, efetuar manobra")
280             refPayload = payloadAsInt
281             afastando = 0
282             aprox = 0
283 }
```

```
282     girar180()
283 }
284 contApproach = 0
285 }
286
287 //encerra o movimento caso proximo ao alvo (RSSI < -40)
288 if payloadAsInt > -45 {
289     //envia um evento ao defense indicando chegada
290     sendEventToDefense("11", "22")
291     fmt.Println("Robotino chegou ao objetivo.")
292     //finaliza loop de movimento
293     return
294 }
295 duration := 150 * time.Millisecond
296 time.Sleep(duration)
297 }
298 }
299
300 func selectNumber() {
301     reader := bufio.NewReader(os.Stdin)
302     fmt.Println("numero:")
303     _, err := fmt.Fscanf(reader, "%d\n", &num)
304     if err != nil || num < 1 || num > 3 {
305         fmt.Printf("Invalid input.")
306         return
307     }
308 }
309
310 func subscribeToTopic(topic string) {
311     if currentTopic != "" {
312         // Unsubscribe from the previous topic
313         if token := client.Unsubscribe(currentTopic); token.Wait() && token.Error() != nil {
314             log.Printf("Error unsubscribing from topic %s: %v\n", currentTopic, token.Error())
315         }
316     }
317
318     fmt.Printf("Subscribing to topic: %s\n", topic)
319     if token := client.Subscribe(topic, 0, onMessageReceived); token.Wait() && token.Error() != nil {
320         log.Fatalf("Error subscribing to topic: %v\n", token.Error())
321     }
322     fmt.Printf("Subscribed to topic: %s\n", topic)
323     currentTopic = topic
324 }
325
326 func subscribeDefense(defTopic string) {
327     fmt.Printf("Subscribing to topic: %s\n", defTopic)
328     if token := defClient.Subscribe(defTopic, 0, messageHandler); token.Wait() && token.Error() != nil {
329         log.Fatalf("Error subscribing to topic: %v\n", token.Error())
330     }
331     fmt.Printf("Subscribed to topic: %s\n", defTopic)
332     currentDefTopic = defTopic
333 }
334
335 func main() {
```

```
336 generateSoapToken()
337 go keepAlive()
338
339 clientID := "mqtt-client-id"
340 //Broker RSSI
341 broker := "tcp://172.26.1.100:1883"
342 //Topicos RSSI
343 topics := [3]string{"esp_1/rssi", "esp_2/rssi", "esp_3/rssi"}
344 //Conexao ao broker RSSI
345 opts := MQTT.NewClientOptions().AddBroker(broker).SetClientID(clientID)
346 client = MQTT.NewClient(opts)
347 if token := client.Connect(); token.Wait() && token.Error() != nil {
348     log.Fatalf("Error connecting to RSSI MQTT broker: %v\n", token.Error())
349 }
350 //Broker Defense
351 defenseBroker := "mqtts://10.0.0.10:1884"
352 //Credenciais broker Defense
353 MQpass := "7POXEN13n4z92DIk"
354 MQuser := "admin"
355 //Topico Eventos
356 defTopic := "mq/alarm/msg/topic/1"
357 config := &tls.Config{
358     InsecureSkipVerify: true,
359 }
360 defOpts := MQTT.NewClientOptions().AddBroker(defenseBroker).SetClientID(clientID)
361 defOpts.SetTLSConfig(config)
362 defOpts.Password = MQpass
363 defOpts.Username = MQuser
364 defClient = MQTT.NewClient(defOpts)
365 //Conexao ao broker Defense
366 if token := defClient.Connect(); token.Wait() && token.Error() != nil {
367     log.Fatalf("Error connecting to Defense MQTT broker: %v\n", token.Error())
368 }
369
370 //end
371
372 //subscribe to # topic at Defense broker
373 subscribeDefense(defTopic)
374
375 fmt.Println("Esperando evento do Defense")
376
377 wg.Add(1)
378 go func() {
379     defer wg.Done()
380     fmt.Print("Em espera...")
381     for {
382         select {
383             case <-printCh:
384                 selectedTopic := topics[num-1]
385                 subscribeToTopic(selectedTopic)
386                 select {
387                     case <-printCh2:
388                         //manda evento ao Defense indicando que esta iniciando rotina (pop-up de
389                         //camera)
390                         sendEventToDefense("1", "2")
391                         // Print the payload
392                         fmt.Printf("Current payload: %s\n", currentPayload)
393                         payloadAsInt, err := strconv.Atoi(currentPayload)
```

```
393     fmt.Printf("%d", payloadAsInt)
394     if err != nil {
395         fmt.Printf("could not convert payload to integer.\n")
396         return
397     }
398     if payloadAsInt < -40 {
399         //if currentPayload != "-40" || currentPayload != "-39" ||
currentPayload != "-38" {
400             fmt.Printf("Moving until get close.\n")
401             move(num)
402         }
403         num = 0
404         fmt.Printf("Rotina, finalizada.\n")
405         break
406     }
407
408     case <-exitCh:
409         // Exit the goroutine when requested
410         return
411     }
412 }
413 }()
414 // Wait for Ctrl+C to exit
415 c := make(chan os.Signal, 1)
416 signal.Notify(c, os.Interrupt)
417 <-c
418
419 // Signal the exit to the message processing goroutine
420 close(exitCh)
421 wg.Wait()
422
423 // Disconnect from the MQTT broker
424 client.Disconnect(250)
425 fmt.Println("Disconnected from MQTT broker")
426 }
427
428 // DEFENSE IA comm
429
430 type defenseTokenResponseStruct struct {
431     Code int      `json:"code"`
432     Desc string   `json:"desc"`
433     Data struct {
434         Token   string `json:"token"`
435         Duration string `json:"duration"`
436     } `json:"data"`
437 }
438
439 var defenseIP string
440 var defensePort int
441 var defenseAccessKey string
442 var defenseSecretKey string
443
444 func sendEventToDefense(eventType string, eventSource string) {
445     //Informacoes de conexao ao Defense IA
446     defenseIP = "10.0.0.10"
447     defensePort = 8000
448     defenseAccessKey = "5mHo9a2713i9U9x2Bo5Qz4V9"
449     defenseSecretKey = "EdCORWY6N2hqv29W1630gYu="
```

```
450 nowTimestamp := time.Now()
451 nowTimestamp.UnixMilli()
452 data := []byte(`{
453     "accessKey": "` + defenseAccessKey + `",
454     "signature": "` + HMAC256(fmt.Sprintf("%d", nowTimestamp.UnixMilli()), 
455     defenseSecretKey) + `",
456     "timestamp": "` + fmt.Sprintf("%d", nowTimestamp.UnixMilli()) + `"
457 }`)
458 //Conexao ao Defense IA
459 url := fmt.Sprintf("http://%s:%d/ecos/api/v1.1/%s", defenseIP, defensePort, "account/authorize")
460 fmt.Printf("url: %s\n", url)
461 body, _ := defenseRequest(http.MethodPost, url, bytes.NewBuffer(data), "")
462 var defenseTokenResponse defenseTokenResponseStruct
463 json.Unmarshal(body, &defenseTokenResponse)
464 fmt.Printf("Body: %s\n", body)
465 //Envio de evento
466 pushEventToDefense(defenseTokenResponse.Data.Token, eventType, eventSource, "1233
467     ", "evento do robotino")
468 }
469
470 func HMAC256(payload string, secret string) string {
471     sig := hmac.New(sha256.New, []byte(secret))
472     sig.Write([]byte(payload))
473     sha1_hash := hex.EncodeToString(sig.Sum(nil))
474     return sha1_hash
475 }
476
477 func pushEventToDefense(token string, eventTypeCode string, eventSourceCode string,
478     eventId string, remark string) { //device 2 abertura
479     nowTimestamp := time.Now()
480     nowTimestamp.Unix()
481     //Estrutura pacote
482     data := []byte(`{
483         "eventId": "` + eventId + `",
484         "eventTime": ` + fmt.Sprintf("%d", nowTimestamp.Unix()) + `,
485         "eventSourceCode": "` + eventSourceCode + `",
486         "eventTypeCode": "` + eventTypeCode + `",
487         "remark": "` + remark + `"
488     }`)
489     //Comando para enviar evento
490     url := fmt.Sprintf("http://%s:%d/ecos/api/v1.1/%s", defenseIP, defensePort, "bridge/event/push")
491     body, err := defenseRequest(http.MethodPost, url, bytes.NewBuffer(data), token)
492     fmt.Printf("Body: %s\n", body)
493     if err != nil {
494         fmt.Printf("err: %s\n", err)
495     }
496
497     func CreateDeviceOnBridge(eventSourceCode string, eventSourceName string, token
498         string, defenseIp string, defensePort int) {
499
500         url := fmt.Sprintf("http://%s:%d/ecos/api/v1.1/bridge/1/event/type", defenseIp,
501             defensePort)
502         data := []byte(`{
503             "eventTypeCode": " " + eventSourceCode + " ",
504             "eventName": " " + eventSourceName + " "
505         }`)
506         body, err := defenseRequest(http.MethodPost, url, bytes.NewBuffer(data), token)
507         fmt.Printf("Body: %s\n", body)
508         if err != nil {
509             fmt.Printf("err: %s\n", err)
510         }
511     }
512 }
```

```
501     `)
502 body, _ := defenseRequest(http.MethodPost, url, bytes.NewBuffer(data), token)
503 //fmt.Println(err, "err")
504 //fmt.Println("[SUCCESS] Device was created")
505 fmt.Println("This event creates a new device on bridge", string(body))
506 // end creation
507
508 }
509
510 func contains(arr []bool) bool {
511     for _, u := range arr {
512         if u == false {
513             return false
514         }
515     }
516     return true
517 }
518
519 func defenseRequest(requestType string, requestURL string, requestData *bytes.
520     Buffer, token string) ([]byte, error) {
521     var body []byte
522     var err error
523     client := &http.Client{}
524     req, err := http.NewRequest(requestType, requestURL, requestData)
525     if err != nil {
526         fmt.Println(err, "Erro1")
527     }
528     req.Header.Add("Content-Type", "application/json")
529     if token != "" {
530         req.Header.Add("X-Subject-Token", token)
531     }
532
533     resp, err := client.Do(req)
534     if err != nil {
535         fmt.Println(err, "Erro2")
536     } else {
537
538         body, err = io.ReadAll(resp.Body)
539     }
540     return body, err
541 }
```

Código A.1 – Middleware