UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ – UTFPR
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
CURSO DE BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO

HELENA SENN ARINS
LUIZ HENRIQUE PEREIRA
VINICIUS AMILGAR BRENNER

# READY - PROPOSTA E PROTOTIPAÇÃO DE UMA PLATAFORMA PARA DESENVOLVIMENTO DE APLICAÇÕES PARA ROBÔS DOMÉSTICOS

TRABALHO DE CONCLUSÃO DE CURSO

**CURITIBA**

**2020**

HELENA SENN ARINS
LUIZ HENRIQUE PEREIRA
VINICIUS AMILGAR BRENNER

# READY - PROPOSTA E PROTOTIPAÇÃO DE UMA PLATAFORMA PARA DESENVOLVIMENTO DE APLICAÇÕES PARA ROBÔS DOMÉSTICOS

Trabalho de Conclusão de Curso apresentado aos Departamentos Acadêmicos de Informática (DAINF) e Eletrônica (DAELN) como requisito parcial para obtenção do grau de Bacharel no Curso Superior de Bacharelado em Engenharia de Computação da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. João Alberto Fabro

**CURITIBA**

**2020**

**HELENA SENN ARINS**

**LUIZ HENRIQUE PEREIRA**

**VINICIUS AMILGAR BRENNER**

# READY - PROPOSTA E PROTOTIPAÇÃO DE UMA PLATAFORMA PARA DESENVOLVIMENTO DE APLICAÇÕES PARA ROBÔS DOMÉSTICOS

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título em Engenharia de Computação da Universidade Tecnológica Federal do Paraná (UTFPR).

Data de aprovação: 09 de dezembro de 2020

_____

João Alberto Fabro
Mestre em Engenharia Elétrica, Doutor em Engenharia Elétrica e Informática Industrial
Universidade Tecnológica Federal do Paraná (UTFPR).

_____

Ronnier Frates Rohrich
Mestre em Engenharia Elétrica e Sistema de Potência e Inteligência Artificial Aplicada, Doutorado em Engenharia Elétrica e Informática Industrial
Universidade Tecnológica Federal do Paraná (UTFPR).

_____

Marco Aurélio Wehrmeister
Mestre em Ciência da Computação e Doutorado em Computação
Universidade Tecnológica Federal do Paraná (UTFPR).

**Curitiba**

**2020**

# Agradecimentos

Tornar um conceito abstrato, uma ideia, em um produto real e tangível é uma tarefa desafiadora, que exige planejamento, pesquisa e muita dedicação. Este processo desgastante muitas vezes pode se tornar desanimador e quando este evento ocorre nada é mais valioso do que ser estimulado pela a animação de alguém que nunca deixou de acreditar nesta ideia.

Portanto, gostaríamos de iniciar esta seção agradecendo ao Professor Fabro, nosso orientador, por nos inspirar através da persistência, sempre nos apoiando e encorajando para que não desistíssemos e, principalmente, sempre nos orientando em momentos nos quais sequer éramos capazes de definir os limites do projeto, além da paciência exigida ao corrigir as inúmeras versões geradas deste trabalho.

Nosso agradecimento também se estende a nossa família, por nos apoiar não só durante a elaboração deste trabalho, mas por toda as nossas vidas, nos dando a oportunidade a cada dia de nos tornarmos pessoas melhores, nos inspirando a continuar sempre aprendendo e por nos confortar durante as muitas dificuldades envolvidas em nossos caminhos.

E por fim a paciência de nossos amigos, que não se esforçaram em disfarçar a ansiedade de ver este projeto finalizado com a intenção de voltar a contar conosco para a realização dos mais diversos eventos e projetos.

# Resumo

O *Framework Ready* é uma ferramenta que tem como objetivo auxiliar e padronizar o desenvolvimento de aplicações para robôs domésticos, tornando esta ação mais simples e independente do fabricante. Esta ferramenta foi planejada para ser capaz de gerenciar recursos do sistema e ainda encapsular funções que podem ser uteis ao desenvolvimento deste gênero de aplicação, como por exemplo as relacionadas a comunicação com o usuário e a movimentação do robô. Compõe o *Framework* proposto um *middleware*, capaz de gerenciar os recursos do robô e prover funcionalidades às aplicações, e bibliotecas que permitem as aplicações se conectarem ao *middleware*. Este documento também contempla a descrição do desenvolvimento de três aplicações que utilizam as funcionalidades do *Framework Ready*, utilizadas para a demonstração da proposta.

**Palavras-chaves**: Robô Doméstico, Interface de Programação de Aplicações (API), *Framework*, *Robot Operating System* (ROS), *Middleware*.

# Abstract

Ready Framework is a tool that aims to assist and standardize the development of applications for housework, making this action simpler and independent of the manufacturer. This tool was designed to be able to manage system resources and also encapsular functions that can be useful for the development of this type of application, such as those related to communication with the user and the movement of the robot. The Framework composes a middleware, capable of managing the robot resources and providing functionality to the applications, and libraries that allow applications to connect to the middleware. This document also includes a description of the development of three applications that use the features of Ready Framework, used for the proposal demonstration.

**Key-words**: Domestic Robot, Application Program Interface (API), Framework, Robot Operating System (ROS), Middleware

# Lista de ilustrações

# Lista de abreviaturas e siglas

| | |
|---|---|
| API | *Application Program Interface* - Interface de Programação de Aplicações |
| BLE | *Bluetooth Low Energy* - *Bluetooth* de Baixa Energia |
| CLR | *Common Language Runtime* - Linguagem Comum em Tempo de Execução |
| DAINF | Departamento Acadêmico de Informática |
| DAELN | Departamento Acadêmico de Eletrônica |
| DDS | *Data Distribution Service* - Serviço de Distribuição de Dados |
| HTTP | *Hypertext Transfer Protocol* - Protocolo de Transferência de Hipertexto |
| IoT | *Internet of Things* - Internet das Coisas |
| JSON | *JavaScript Object Notation* - Notação de Objeto em *JavaScript* |
| LTS | *Long Term Support* - Suporte de Longo Prazo |
| MCP | *Master Control Program* - Programa de Controle Mestre |
| ML | *Machine Learning* - Aprendizado de Máquina |
| MVC | *Model View Controller* - Modelo Visão Controle |
| NPM | *Node Package Manager* - Gerenciador de Pacotes do Node |
| PCL | *Point Cloud Library* - Biblioteca de Nuvem de Pontos |
| POO | Programação Orientada a Objetos |
| REST | *Representational State Transfer* - Transferência Representacional de Estado |
| ROS | *Robot Operating System* - Sistema Operacional de Robôs |
| SDK | *Software Development Kit* - *Kit* de Desenvolvimento de Programas |
| SGDB | Sistema de Gerenciamento de Banco de Dados |
| SLAM | *Simultaneous localization and mapping* - Localização e Mapeamento Simultâneos |
| SO | Sistema Operacional |
| SOA | *Service-Oriented Architecture* - Arquitetura Orientada a Serviços |

| | |
|---|---|
| SQL | *Structured Query Language* - Linguagem de Consulta Estruturada |
| TCP/IP | *Transmission Control Protocol / Internet Protocol* - Protocolo de Controle de Transmissão / Protocolo de Internet |
| URL | *Uniform Resource Locator* - Localizador Uniforme de Recursos |
| UTFPR | Universidade Tecnológica Federal do Paraná |
| WSDL | *Web Services Description Language* - Linguagem de Descrição de Serviços *Web* |
| XML | *Extensible Markup Language* - Linguagem de Marcação Extensível |
| YAML | *YAML Ain't Markup Language* - YAML Não é Linguagem de Marcação |

# Sumário

# 1 Introdução

Atualmente robôs são utilizados nas mais diferentes áreas, como meios de aumentar a eficiência e a produtividade, bem como executar tarefas perigosas (CHIJINDU; INYIAMA, 2012), entretanto são raros os robôs que possuem complexidade suficiente para executar tarefas cotidianas de maneira gerais (MCGINN et al., 2014), sendo então mais comum dispositivos especializados. Este fato torna o desenvolvimento de *software* para robôs também uma atividade bastante específica, uma vez que cada robô possui características mecânicas ou de *hardware* únicas.

Para contornar este problema há esforços como o ROS (*Robot Operating System*), um conjunto de bibliotecas e ferramentas de *software* que visam padronizar o desenvolvimento de aplicações para robôs (ROS, 2020b), permitindo que diferentes modelos de robôs possam compartilhar partes de suas programações, assim melhorando o índice de reaproveitamento de código.

O ROS é descrito por seus desenvolvedores, como um *"meta-operating system"*, por englobar conceitos como abstração de *hardware*, controle de componentes eletrônicos, e ainda possuir como funcionalidade um meio que permite a troca de informações entre processos (TSARDOULIAS; MITKAS, 2017), sendo uma ferramenta para controlar recursos e diversos modelos de robôs de forma padronizada.

Entretanto, mesmo sendo composto por inúmeras ferramentas que padronizam o desenvolvimento de robôs, o ROS ainda é um sistema flexível o suficiente de maneira a permiti-lo ser usado em uma vasta gama de robôs, como, por exemplo, *drones*, robôs terrestres com diferentes maneiras de movimentação, entre outras variações, não limitando o desenvolvimento de programas para tipos específicos de robôs. Esta decisão torna os próprios desenvolvedores responsáveis em estabelecer a arquitetura de *software* utilizada pelo sistema, o que significa que para desenvolver novas funcionalidades é necessário ter conhecimento prévio desta organização, fato que exige treinamento para novos programadores que ingressarem à equipe, além de impedir que as soluções implementadas sejam usadas em robôs que possuam arquitetura de *software* diferente, mesmo utilizando o ROS.

Esta característica torna o desenvolvimento de programas para estes dispositivos ainda limitado aos fabricantes de robôs, que normalmente disponibilizam funcionalidades e aplicações específicas, e a programadores que conhecem o funcionamento e especificações do ROS, do *hardware* e do *software* que o compõe dispositivo, o que envolve os pacotes (*Software* ROS) utilizados pelo sistema, bem como a maneira como todos os componentes se interligam.

Além da limitação em relação à padronização da arquitetura de *software* utilizado pelo robô, não está entre os objetivos do ROS monitorar o gerenciamento de recursos do sistema,

fato que torna os desenvolvedores responsáveis em incorporarem estas políticas nas próprias aplicações de modo a evitar conflitos.

Visando diminuir as possíveis dificuldades encontradas por desenvolvedores de *software* para robôs que utilizam o ROS através de um sistema que gerencie os recursos e forneça diferentes funcionalidades, este trabalho propõe uma arquitetura de *software* baseada em um *Framework* denominado *Ready*, que visa a simplificação do desenvolvimento de *software* para robôs usados em ambientes domésticos, ou seja, robôs de serviço ou de companhia, os quais normalmente são terrestres e algumas vezes operados por comunicação em linguagem natural.

O termo *framework*, que será abordado com mais detalhes no capítulo 2, se refere a um conjunto de ferramentas de desenvolvimento de programas (TSARDOULIAS; MITKAS, 2017).

Foi escolhido como base para o sistema proposto o ROS, devido ao conjunto de ferramentas existentes compatíveis com esta plataforma, que existem atualmente e são compatíveis com o ROS. O *Ready* está sendo proposto com foco nos robôs domésticos, ou seja, robôs de serviço ou de companhia, criados para executar tarefas que auxiliam no cotidiano da pessoa. A descrição da estrutura que forma o *Ready*, encontra-se a seguir.

Iniciando pelo robô, entre os componentes importantes para o entendimento deste trabalho, primeiramente serão apresentados o *Hardware* e os *Drivers*, que são as denominações adotadas para definir os componentes físicos do robô, como motores, sensores, atuadores, e respectivamente o *software* por eles utilizados.

Para acessar os dados fornecidos pelos *Drivers* dos componentes físicos do robô existem os "Serviços", nome dado aos componentes de *softwares* responsáveis por capturar as informações do robô e entregar ao *middleware Ready* na formatação correta. Os serviços devem ser implementados pelo fabricante e se comunicam com as *Daemons*, explicadas adiante.

Ainda entre os componentes do robô, encontra-se o ROS, um meta sistema operacional, que foi escolhido como base do trabalho, por ser capaz de intermediar a troca de dados entre programas através de estruturas denominadas tópicos. Aproveitando este recurso, o ROS é usado como base do *Framework Ready,* permitindo que os serviços sejam capazes de trocar informações com o *middleware Ready*.

Além disso, o ROS também traz como vantagem a padronização da estrutura de tópicos de diferentes tipos de dados, como por exemplo odométrica, velocidade, entre outros. Esta característica facilita a implementação da comunicação entre os serviços e o *middleware*, permitindo que vários pacotes ROS possam ser utilizados ou adaptados como serviços *Ready*. Um pacote ROS é a denominação utilizada para definir um *software* desenvolvido com o intuito de fornecer dados ou controle sobre o robô.

O *Framework Ready*, possuí um *middleware* e um conjunto de bibliotecas responsável pela comunicação entre as aplicações e o *framework*. Estas bibliotecas encapsulam uma

comunicação baseada em API (*Application Program Interface*) REST (*Representational State Transfer*), utilizando conceitos de *webServices*, um estilo de arquitetura de *software* responsável por definir padrões de comunicação em dispositivos, independente da linguagem ou *hardware*. Esses padrões consistem em uma comunicação entre cliente e servidor, no caso do *Ready*, as aplicações são os clientes, e o *middleware* é o servidor.

Este modo de comunicação permite que o *middleware* identifique qual aplicação está realizando a requisição, possibilitando o gerenciamento da entrega de dados e o controle do robô apenas a quem tem direito, além de permitir que as bibliotecas que compõe o *framework* possam ser convertidas para diferentes linguagens de programação, desde que estas suportem *webServices*.

O *middleware Ready* tem como objetivo intermediar e gerenciar a comunicação entre as aplicações e o ROS e controlar o acesso das aplicações a alguns recursos do robô, bem como fornecer funcionalidades complexas, ou seja, funcionalidades que envolvem cálculos ou mais de uma operação, encapsulando-as em funções mais simples. É importante que esta camada isole as aplicações do resto do sistema, de modo a ser o único componente capaz de controlar o robô diretamente e evitar conflitos, pois, na hipótese de uma aplicação conseguir contornar o *middleware*, ela poderá interferir no funcionamento do sistema.

O *middleware Ready* é formada por classes *Singletons*, *Controllers* e *Daemons*. O nome *Singletons* é dado a padrão de projeto responsável por garantir que apenas uma instância da classe seja criada. No *Ready* as *Singletons* são responsáveis pelo gerenciamento das *Daemons*.

As *Controllers*, podem ser resumidas como as classes responsáveis em gerenciar e controlar as entradas das aplicações no *Ready*. Por fim, existem as *Daemons*, que recebem um destaque especial, pois é onde estão as definições das possíveis funcionalidades que poderão ser usadas pelas aplicações.

Entre as inúmeras *Daemons* que podem existir, para a realização desde trabalho, foram criadas apenas quatro, uma *Daemon* de navegação, uma de comunicação, de registros e a *AppManager*,

A *Daemon* de navegação é responsável por buscar, tratar e retornar valores referentes a locomoção do robô, como, por exemplo, as coordenadas da posição atual do robô, ou o valor da distância entre o robô e um determinado ponto de interesse. Essas informações podem ser utilizadas nas mais distintas aplicações que um desenvolvedor de aplicações para robôs possa imaginar, como, por exemplo, construir uma aplicação onde o robô deve ficar andando em círculos por pontos específicos ou aleatórios em um ambiente, para vigiar as atividades ao redor.

A *Daemon* de Iteração Humano-Computador tem a função de escutar e tratar a linguagem falada pelo ser humano e converter em código para o entendimento do robô, assim como retornar as devidas respostas.

Um desenvolvedor de aplicações, tem a liberdade de desenvolver qualquer aplicação que ele/ela consigam de pensar, utilizando as *Daemons* disponibilizadas pelo *Ready*, como, por exemplo, uma aplicação de lista telefônica, em que o usuário consiga solicitar ao robô para buscar, guardar ou remover algum contato de sua lista. Ou mesmo criar uma aplicação que envolva contextos de locomoção e comunicação, como uma aplicação que simule um garçom, onde é possível realizar o pedido, o robô, irá escutar, e buscar o que foi desejado.

Além destas, existem também uma *Daemon* denominada *AppManager*, responsável por gerenciar as tarefas relacionadas à execução das aplicações, como a instalação ou abertura de uma nova aplicação, ou mesmo controlar qual aplicação poderá utilizar os recursos críticos do sistema, entre outras responsabilidades.

Por fim, tem-se a *Daemon* de registros, a qual deve guardar as informações das *Daemons*, necessárias e relevantes para o funcionamento do *Framework Ready*, tais como, o nome e o número de identificação único de cada aplicação.

Ressalta-se que é possível implementar novas *Daemons*, conforme necessidades e demandas, sem interferir no funcionamento das *Daemons* existentes, tornando o *middleware Ready* ainda mais completo e expandindo cada vez mais as funcionalidades possíveis de um robô.

A Figura 1 Apresenta uma ilustração dos componentes que envolvem o trabalho e foram descritos nessa seção

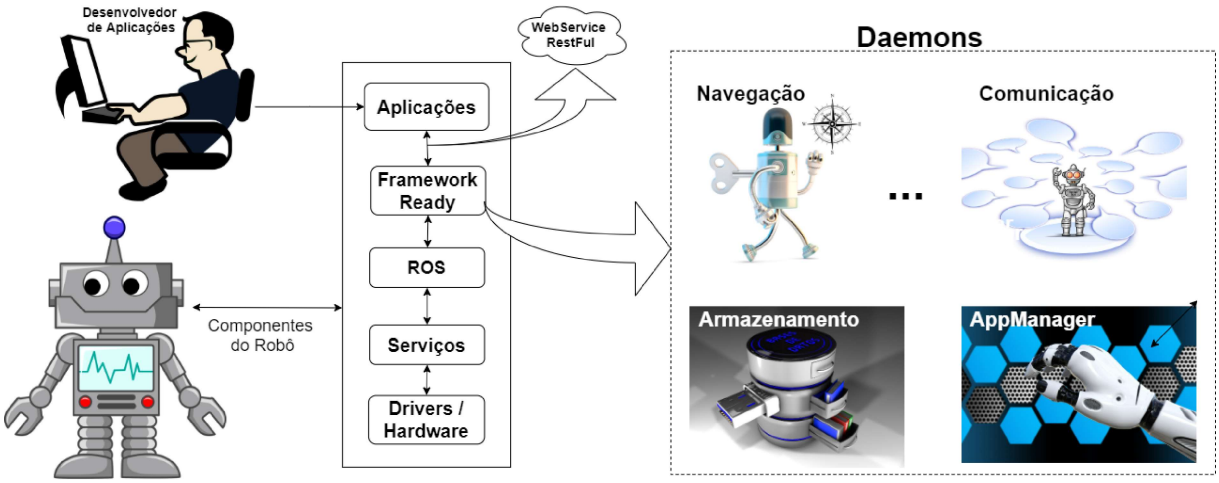



Figura 1 – Ilustração dos componentes que envolvem o trabalho *framework Ready*- Fonte: Autoria Própria.

Gerenciar a comunicação entre as aplicações e os recursos fornecidos pelo ROS, permite o *middleware Ready* monitorá-los e deste modo prevenir que diferentes aplicações entrem em conflitos ao acessar determinadas funcionalidades do robô. Pode-se citar como exemplo de conflito a movimentação inadequada causada pela disputa entre duas ou mais aplicações, que além do comportamento citado, pode comprometer o funcionamento das aplicações envolvidas.

O intermédio entre as aplicações e o ROS também permite que o o *middleware Ready* abstraia o funcionamento interno do sistema aos desenvolvedores, tornando o funcionamento das aplicações, mais previsível.

Além disso, o *framework* também irá padronizar o uso de outras ferramentas não necessariamente ligadas ao ROS, como, por exemplo, bibliotecas de síntese de voz ou então conversão de voz em texto, para fornecer outras funcionalidades aos desenvolvedores.

Assim, este trabalho propõe o *Framework Ready*, um meio de padronizar a arquitetura de *software* para robôs terrestres e domésticos, permitindo funcionalidades expansíveis a partir de aplicações independentes do fabricante, se aproximando da maneira que os aplicativos (Apps) são capazes de expandir a possibilidade do uso de aparelhos celulares, desde que este respeite certos requisitos da plataforma.

## 1.1 Motivação/Justificativa

Um mundo em que robôs passem a ocupar diferentes ambientes e aumentem a capacidade de interação com pessoas, entretendo-as, funcionando como um tipo de brinquedo, ou então melhorando a qualidade de vida, atuando como assistentes pessoais, tem se tornado cada vez mais possível graças aos novos modelos de robôs que estão sendo disponibilizados comercialmente.

Considerando modelos atuais, existem vários robôs domésticos que desempenham funções de companhia, interação, diversão e proteção ao proprietário, mas normalmente estas funções são restritas às especificações dos fabricantes, impossibilitando a implementação de uma nova aplicação em um determinado modelo de robô.

Para exemplificar é possível citar o robô *Cozmo*, que possui a habilidade de interagir com o usuário através de expressões faciais em uma tela, e conta com alguns jogos que estão disponíveis em sua memória. Além disso, o fabricante fornece uma SDK (*Software Development Kit*), um conjunto de ferramentas para aprendizado e pesquisa em robótica. Entretanto, o sistema pode ser utilizado somente no próprio dispositivo, sem a possibilidade de ser executado em outros modelos de robôs, e o mesmo não incentiva a distribuição de *software* (COZMO, 2019).

Outro robô que pode ser citado é o *Fribo*, um sistema que através de som, temperatura e outras características ambientais, analisa a rotina do recinto, aprendendo a identificar diferentes estímulos e detectando situações incomuns. Quando uma situação desta natureza é constatada, o dispositivo se comunica com uma rede de *Fribo* "amigos", notificando a irregularidade (ACKERMAN, 2018).

O fato dos robôs atuais terem suas habilidades restritas ao fornecido pelo do próprio fabricante, além de ser um limitante para as possibilidades de ações que ele poderia ter, também

deixa o proprietário de um dispositivo desses, a mercê de contratempos que podem acontecer com a fabricante, como por exemplo o caso do robô Jibo.

O robô Jibo, auto nomeado "Primeiro robô social" pela fabricante, apareceu na capa da revista *Time* por ser uma das 25 melhores invenções do ano de 2017, como pode ser visto na Figura 2 (TIME, 2017). Infelizmente, o robô Jibo irá perder a maior parte das funcionalidades, pois estas são dependentes dos servidores do fabricante que, por motivo de falência, os desligará (O GLOBO, 2017).


Figura 2 – Jibo na revista Time - *The 25 Best Inventions of 2017* (TIME, 2017)

Ter a possibilidade de expandir as funcionalidades de um robô de acordo com o desejo do proprietário, independente dos fabricantes, com certeza é um fator que influenciaria na tomada de decisão no momento de adquirir um robô, e aumentaria a procura e demanda deste produto no mercado. Um meio de permitir a expansão das funcionalidades de robôs é a criação de aplicações capazes de serem executadas em diferentes modelos, independente dos padrões impostos pelos fabricantes.

Entretanto, desenvolver aplicações que possam ser usadas em diferentes robôs, não é um desafio simples, pois geralmente este tipo de *software* é desenvolvido utilizando conhecimentos especializados de um determinado modelo.

Pensando em melhorar a experiência de quem possui um robô doméstico, possibilitando aumentar a gama de funções e atuação do mesmo, além de facilitar e padronizar o desenvolvimento de novas aplicações para robô, de modo a mantê-la independente de fabricante, este trabalho propõe a criação do *Framework Ready*, um conjunto de ferramentas para o desenvolvimento de aplicações para robôs, que utiliza o ROS como base e é capaz de gerenciar recursos

críticos do robô, além de entregar informação prontas e precisas ao desenvolvedor de aplicações, como a posição atual do robô, ou a distância que existe até um determinado ponto.

Os desenvolvedores de aplicações para robôs, poderão desenvolver novos programas sem necessitar conhecer detalhes de funcionamento do dispositivo para o qual esta será usado e ainda contarão com recursos que podem acelerar a implementação de softwares para este fim, deste que respeitando os requisitos necessários para o uso do *framework Ready*.

Com a padronização do desenvolvimento e do funcionamento das aplicações para robôs, espera-se melhorar a experiência do usuário final, que poderá contar com mais aplicações compatíveis para melhor adequar o dispositivo às próprias necessidades, e não apenas as imaginadas pelo fabricante.

A existência de um robô capaz de interagir com o usuário e, ao mesmo tempo, ser capaz de aceitar novas funções e aplicativos instigou a equipe, pois o desenvolvimento deste trabalho une muitos dos conhecimentos e das experiências obtidas pelos membros no decorrer do curso.

## 1.2 Objetivo Geral

O objetivo geral deste trabalho é propor, implementar e testar um *Framework* voltado ao desenvolvimento de aplicações para robôs, denominado *Ready*. Este *Framework* será a base de uma arquitetura de *software* que visa permitir a expansão de funcionalidades para este tipo de dispositivo por meio de aplicações robóticas desenvolvidas por programadores independentes do fabricante.

Pode-se definir um *framework* como um conjunto de objetos que colaboram com o objetivo de atender a um conjunto de responsabilidades para uma aplicação específica ou um domínio de aplicação (TSARDOULIAS; MITKAS, 2017). No escopo deste trabalho, o *framework Ready* tem a responsabilidade de gerir e intermediar os serviços de robótica, programas que permitem o monitoramento e o controle das funções do robô, com as aplicações descritas no parágrafo anterior.

Este *framework* será composto por um *middleware* e um conjunto de bibliotecas que permitirão a comunicação entre as aplicações e o *framework*. Um *middleware* pode ser definido como um *software* que se encontra entre duas camadas de abstração, neste caso entre os serviços e as aplicações, funcionando de maneira geral como uma camada oculta de tradução. De maneira geral, um *middleware* permite a comunicação e o gerenciamento de dados para aplicativos distribuídos (MICROSOFT AZURE, 2020).

Considerando isso, o propósito do trabalho é o desenvolvimento do *framework* para robô doméstico, o que inclui testá-lo utilizando um ambiente virtual, visando verificar a viabilidade de aplicação do conceito proposto.

Espera-se que objetivos descritos para este trabalho sejam capazes de atender o interesse

dos seguintes grupos de possíveis usuários da plataforma listados a seguir:

- Fabricante de Robôs: Podem se beneficiar com os padrões propostos por este trabalho por não precisarem dedicar tempo e recursos desenvolvendo o comportamento do dispositivo bem como a arquitetura de software interna deste, concentrando-se apenas na adaptação das funcionalidades do robô ao *middleware* proposto.
- Desenvolvedores de Software: Incentivar a criação de aplicações para robôs por desenvolvedores que conheçam programação mas não possuam familiaridade com a implementação de software voltados a este contexto, pode aumentar a distribuição deste tipo de software.
- Usuários Finais: A maior distribuição de aplicações para robôs tem o potencial de atrair mais consumidores deste tipo de dispositivo, uma vez que estes passariam a ser mais flexíveis e adaptáveis ao uso de diferentes perfis de usuários.

Com o potencial de massificação e facilitação de distribuição de *software*, poderia haver tanto uma demanda quanto pessoas dispostas a cumpri-la, podendo tornar robôs mais baratos e populares.

## 1.3 Objetivos Específicos

Os objetivos específicos deste trabalho são:

- Desenvolver um *middleware* capaz de gerenciar recursos disponibilizados pelos robôs.
- Desenvolver bibliotecas que permitam as aplicações se comunicarem com o *middleware* desenvolvido pela equipe.
- Desenvolver os serviços de Comunicação e Navegação para permitir que aplicações de testes sejam desenvolvidas.
- Desenvolver aplicações que utilizam os recursos fornecidos pelo *middleware* como prova de conceito.
- Testar o *Framework* desenvolvido em um ambiente virtual para verificar se os resultados do funcionamento esta de acordo com o previsto.

# 2 Revisão Bibliográfica

Este capítulo visa revisar e explicar conceitos e tecnologias empregadas durante o desenvolvimento deste trabalho, dividindo-o em duas partes, a primeira, “Propostas Similares”, apresentará tecnologias existentes similares à proposta por este trabalho, bem como um resumo do funcionamento e uma breve descrição do contexto que as envolvem. Na segunda “Tecnologias utilizadas”, serão descritas todas as ferramentas, tecnologias e elementos empregados na implementação da proposta do trabalho, objetivando tornar o leitor confortável com estes termos.

## 2.1 Propostas Similares

Os itens abaixo apresentam exemplos de projetos existente que possuem objetivos similares ao proposto pelo *Ready*.

- *CylonJS*

  *CylonJS* é um *Framework* robótico que possui como objetivo facilitar o desenvolvimento de robôs e outras aplicações IoT (*Internet of Things*). Ele é utilizado em conjunto com a linguagem *JavaScript*, podendo ser instalado por meio do gerenciador de pacotes da linguagem, o NPM (*Node Package Manager*). Atualmente o projeto suporta o *hardware* de plataformas de desenvolvimento diferentes, incluindo o *Arduino* e o *Raspberry*, além de possuir funcionalidades de *software* como o *Speech*, que utiliza o *eSpeak* para a síntese de voz e o suporte para a tecnologia de *Bluetooth Low Energy (BLE)* (CYLONJS, 2016b).

  A arquitetura do *CylonJS* possui o componente MCP (*Master Control Program)*, responsável por gerenciar a comunicação entre os robôs e a API (*Application Program Interface*). A API, componente que de fato se comunica com as aplicações, possui como características a capacidade de incorporar novas funcionalidades por meio de *plugins*, tornando os arquivos base do *CylonJS* mais leves além de preservar o sistema de ser composto por recursos desnecessários. Os *plugins* são os componentes da API que se comunicam diretamente com o MCP, o que os tornam capazes de controlarem e de obterem informações de robôs ou outros dispositivos conectados. A Figura 3 apresenta o digrama de blocos da arquitetura *CylonJS* (CYLONJS, 2016a).

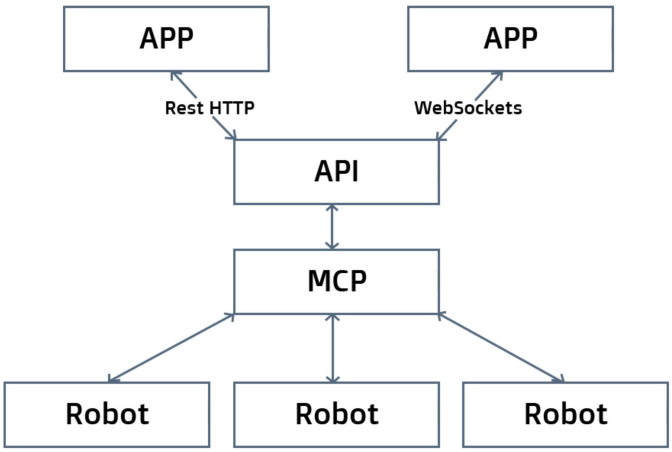



Figura 3 – Arquitetura do CylonJS - Fonte: (CYLONJS, 2016a)

- *Gobot*

  *Gobot* é um *Framework* de robótica, construído para tornar mais fácil o desenvolvimento de robôs, *drones* ou outros sistemas IoTs. Similar ao *Framework CylonJS*, o *Gobot* também fornece suporte para controlar vários dispositivos físicos de *Hardware*, como o *Arduino* e o *Raspberry PI*, porém utilizando a linguagem de programação GO (GOBOT, 2019).

  Programas escritos a partir do *Framework Gobot* são denominados *Robots*, e podem ser executados em um sistema operacional *Linux* ou em qualquer máquina conectada ao dispositivo desejado e suportado. Estes programas consistem em modelos que descrevem em *software* a implementação física do conjunto do *hardware* a ser controlado (GOBOT, 2019).

  Deste modo, o termo *Robots* é usado pelo *Gobot* para representar qualquer dispositivo conectado, robôs e *drones*, e tem conexão com vários adaptadores e dispositivos de controle. Esses dispositivos são os responsáveis por fornecer a interface necessária para a comunicação entre o *Gobot* e as plataformas de *hardware* como o *Arduino* e o *Raspberry PI* (GOBOT, 2019).

  Os *drivers* estão diretamente em contato com os adaptadores e providenciam suporte para comportamentos específicos do *Hardware*, como botões e *leds*. Logo abaixo dos *Drivers* estão os eventos, ou seja, representação de ocorrências, que são utilizados pelos *drivers*, que retornam ao robô por um desvio de caminho. A Figura 4 apresenta a arquitetura do funcionamento do *Framework Gobot* (GOBOT, 2019).

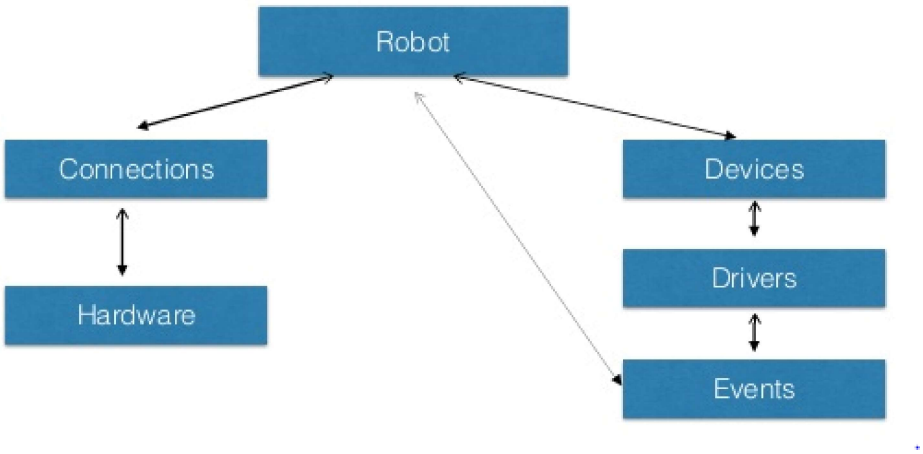



Figura 4 – Arquitetura do *Gobot* - Fonte: (GRAMMENS, JUSTIN, 2015)

- *Johnny-Five*

  O *framework Johnny-Five* é um projeto de código fonte aberto baseado no protocolo de comunicação *Firmata*, sendo portando compatível com as placas *Arduino*, *Electric Imp*, *Beagle Bone*, *Intel Galileo & Edison*, *Raspberry Pi* entre outras (JOHNNY-FIVE, 2012).

  O *Framework Johnny-Five* utiliza o *JavaScript* como meio de controlar dispositivos físicos como servos motores, potenciômetros entre outros. Para cumprir esta tarefa, os desenvolvedores do *framework* trouxeram não apenas a linguagem, mas também os paradigmas e as técnicas de programação que a envolvem visando facilitar tarefas robóticas, como por exemplo, tornar a movimentação de um braço robótico, semelhante a "animar um elemento em uma página *web*" (JOHNNY-FIVE, 2012).

  Para permitir a construção de programas capazes de controlar componentes de *hardware* da mesma maneira independente da plataforma escolhida, o *framework Johnny-Five* atua na abstração da comunicação (JOHNNY-FIVE, 2012).

- Avaliação das propostas similares

  Um fator comum entre todas as propostas apresentadas é o fato destas utilizarem linguagens de alto-nível, como por exemplo o *Go* e o *JavaScript*, inclusive a última sendo bastante utilizada em desenvolvimento Web.

  Entretanto, estas tecnologias operam utilizando apenas uma única linguagem de programação e são compatíveis com um determinado conjunto de dispositivos. Estas características, embora as tornem mais simples de serem utilizadas, podem ser um fator limitante em alguns projetos.

  Estas limitações inspiraram a criação do *framework Ready*, similar aos apresentados, porém capaz de combinar a compatibilidade com diferentes linguagens de programação e o extenso suporte de *software* e *hardware* já disponíveis para o ROS. Além disso, o *Ready* propõe um modelo de arquitetura que o permite gerenciar alguns recursos, tornando-o

menos flexível que o ROS, porém mais previsível aos desenvolvedores não familiarizados com o desenvolvimento de software para robôs.

## 2.2 Tecnologias utilizadas

Esta seção é composta pelas descrições teóricas dos componentes, biblioteca e termos utilizados na execução do trabalho. Para isso, primeiramente haverá uma breve explicação de alguns termos e conceitos voltados a área de computação e tecnologia, comuns no ambiente da informática e que são utilizados durante o desenvolvimento do trabalho, na subseção 2.2.1 "Termos e Conceitos de Programação".

Na sequência, a subseção 2.2.2 "Conceitos Tecnológicos", apresenta os conceitos necessários para a compreensão das tecnologias utilizadas para a implementação deste trabalho.

### 2.2.1 Termos e Conceitos de Programação

Esta seção contém itens com uma breve descrição dos mais importantes conceitos citados neste trabalho, para promover uma melhor compreensão. O primeiro item apresenta a definição de *Singleton*, também é introduzido o significado de *software Open Source*, termo que descreve a natureza de vários componentes envolvidos no trabalho. Finalmente, o termo MVC, ou *Model View Control*, será definido com maior detalhes, uma vez que é um conceito utilizado durante a implementação da comunicação das aplicações e do *middleware*

- *Singleton*

  Na área da lógica e matemática a definição para um *singleton* é dada como "um conjunto que contém só e somente só um elemento". Portanto não importa o quanto alguém tente conseguir mais de um elemento, se ele foi definido como um *singleton*, o elemento será sempre o mesmo (BADENHORST, 2017).

  Em uma aplicação orientada a objetos, uma classe *singleton* sempre retorna a mesma instância de si, fornecendo um ponto de acesso global para os recursos e objetos da classe. Um padrão de projeto chamado de padrão *singleton*, é responsável em garantir que as classes tenham uma única instância e providencie um ponto global de acesso (BADENHORST, 2017).

- *Open Source*

  "*Open Source* é um termo originalmente utilizado para se referir a *softwares* abertos ao público, ou seja é um *software* em que qualquer pessoa possa "vê-lo, modificá-lo e distribuí-lo conforme as próprias necessidades e as restrições impostas pela licença que o controla (REDHAT, 2020).

  Entretanto, atualmente o termo pode ser usado para referenciar um movimento tecnológico e uma forma de trabalho além da produção de *software* (REDHAT, 2020). Deste

modo, neste trabalho poderá ser encontrado também o termo "*Hardware Open Source*", o qual descreve um "artefato tangível", cujo projeto deve estar de acordo com os mesmos preceitos citados no parágrafo anterior (OSHWA, 2020).

- *Model View Controller* (MVC)

  O padrão de design MVC foi introduzido juntamente com a linguagem de programação *Smalltalk* na década de 70, e desde então se propagou como uma arquitetura popular para desenvolvimento de aplicações, principalmente com linguagens orientadas à objetos (DEACON, 2009). A arquitetura consiste em organizar o projeto em três seções distintas, denominadas *Model*, *View* e *Controller*, cada uma com suas responsabilidades no processo de interação do usuário (SELFA; CARRILLO; BOONE, 2006). Um exemplo da estrutura de interação da arquitetura pode ser visto na Figura 5.

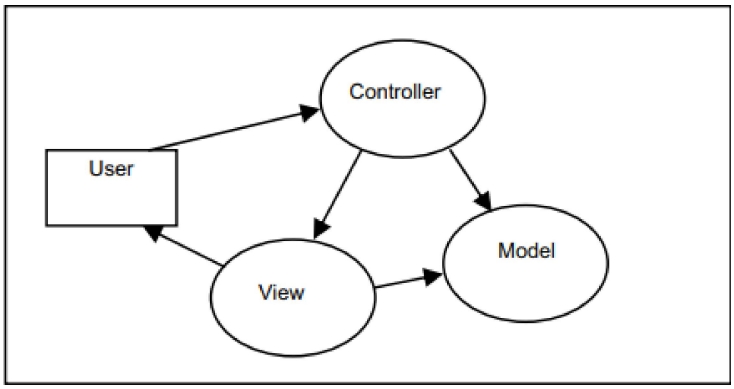



Figura 5 – Arquitetura MVC - Fonte: (SELFA; CARRILLO; BOONE, 2006)

O *Model* é a estrutura responsável pelo gerenciamento de regras de negócio e o estado da aplicação, bem como e a persistência da mesma em bases de dados. Normalmente o *Model* não possui acesso às outras estruturas da arquitetura, limitado apenas ao próprio escopo (VOORHEES, 2020).

A *View* é a responsável por exibir ou disponibilizar os dados organizados pelo *Model* ao usuário, organizando somente a estrutura em que estes dados serão entregues, normalmente não possuindo regras de processamento em seu escopo. A *View* também pode ser notificada em mudanças de estados no *Model* pela *Controller* (VOORHEES, 2020).

A *Controller* é a estrutura capaz de acessar tanto o *Model* quanto a *View*. Ela gerencia as entradas de usuário e altera os estados de aplicação a fim de que a requisição seja processada e o resultado seja entregue pela *View* (VOORHEES, 2020).

### 2.2.2 Conceitos Tecnológicos

Nesta subseção são descritos conceitos necessários para o entendimento das tecnologias utilizadas durante a implementação deste trabalho. Esta compreensão é importante pois as

tecnologias citadas influenciaram as escolhas envolvidas no desenvolvimento deste *Framework* além de orientar o desenvolvimento da própria arquitetura. Dentro destes conceitos cita-se por exemplo as linguagens de programações escolhidas, as bibliotecas utilizadas entre outras outras ferramentas exploradas.

Os próximos parágrafos fornecem uma definição para o termo *Framework* e como ele se encaixa na área da robótica. Logo em seguida, encontra-se a definição do que é um *Middleware* e a importância deste no contexto da robótica. O termo API (*API –Application Programming Interface*) bem com os principais componentes envolvidos neste trabalho, como os *WebServices REST*, apresentados também são explicados ao decorrer desta seção.

- *Framework*

  Um possível significado em português para a palavra *framework* consiste em defini-lo como uma estrutura de suporte. Sob a ótica da engenharia de *software*, pode se entender esta estrutura como um conjunto de objetos abstratos que apoiam o funcionamento de um determinado subsistema de aplicativos (TSARDOULIAS; MITKAS, 2017). Objetos abstratos são modelos usados para representar elementos do mundo real que possuem relevância para um determinado problema (RICARTE, 2001).

  Portanto, um *framework* é uma solução desenvolvida para ser utilizada em inúmeros programas, capaz de descrever o modo como estes são decompostos em conjuntos de objetos que interagem entre si (TSARDOULIAS; MITKAS, 2017). Um *Framework* visa auxiliar os desenvolvedores através de funcionalidades organizadas por responsabilidades similares, permitindo-o concentrar esforços de desenvolvimento nas especificidades do aplicativo a ser implementado (TSARDOULIAS; MITKAS, 2017).

  Aproximando a definição de *framework* do parágrafo anterior ao contexto do desenvolvimento de robôs obtêm-se a expressão "*Framework* Robótico", que ainda sintetiza a ideia de uma coleção de ferramentas de *software* e padronização de arquitetura de aplicação, entretanto estes conceitos passam a ser especificamente voltados ao desenvolvimento de tarefas e funções úteis ao desenvolvimento de *software* usados por robôs (TSARDOULIAS; MITKAS, 2017).

- *Middleware*

  Um *Middleware* pode ser definido como uma camada de *software* que é executado sobre um sistema operacional, porém em um nível abaixo da aplicação, como representado na Figura 6, fornecendo um grau de abstração maior do que o sistema operacional é capaz de proporcionar ao desenvolvedor, permitindo o desenvolvimento de *software* mais portável e com maior produtividade (BAKKEN, 2001).

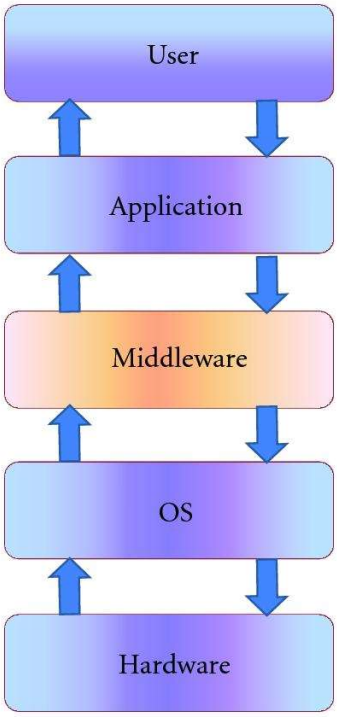



Figura 6 – Camadas a abstração de um *Middleware* - Fonte: (ELKADY; SOBH, 2012)

Aprofundando ao *middleware* robótico, este pode ser exemplificado como uma junção que une diferentes módulos de um sistema robótico. A tarefa mais básica de um *middleware* robótico é fornecer a infraestrutura de comunicação entre os nós de *software* em execução num sistema robótico (TSARDOULIAS; MITKAS, 2017). Uma das principais vantagens que um *middleware* robótico é capaz de fornecer, é a modularização de *softwares*, funcionando de forma independente das características especificas dos *hardwares*, ou seja, a portabilidade entre diferentes dispositivos deve ser acontecer apenas com mudanças nas configurações de sistema. Além disso, um *middleware* deve eficiente e flexível, com fácil uso e manutenção (ELKADY; SOBH, 2012).

Neste trabalho, o *Middleware Ready* está contido dentro do *Framework Ready* intermediando na comunicação com o ROS. A Figura 11 apresentada no capítulo 3 exemplifica as camadas de abstração existente no trabalho.

- *API –Application Programming Interface*

  Interface de Programação de Aplicações (API) é uma especificação de possíveis interações com um componente de *software*. Utilizando-a é possível interagir com um componente ou recurso de *software* separado, sem precisar necessariamente entender o processo de criação, ou funcionamento, apenas a forma de uso (FREEMAN, 2018).

  Uma API é capaz de permitir que desenvolvedores tenham acesso a várias informações, na forma de parâmetros, métodos, classes ou variáveis. APIs utilizam-se de requisições de

funções em programas, que acontece através de diversos protocolos e rotas, e recebe como retorno dados em formato XML *(Extensible Markup Language)* ou JSON *(JavaScript Object Notation)* (BOILLOT, 2012).

Entre os diversos tipos de API existentes, o *WebService* se destaca por utilizar tipicamente o protocolo HTTP *(Hypertext Transfer Protocol)* para comunicação (FREEMAN, 2018) e ser *open source*, ou seja, é um *software* que qualquer pessoa pode ver, modificar e distribuir (REDHAT, 2020). No próximo item, esse assunto será apresentado com mais detalhes.

- *WebServices*

  Um *WebService* é um tipo específico de API aberta, que atende a um conjunto de especificações, incluindo as especificadas na WSDL (*Web Services Description Language*), uma variante XML (*Extensible Markup Language*) (FREEMAN, 2018).

  Os *WebService* fornecem um modo de comunicação entre dispositivos eletrônicos ou diferentes aplicativos de *software* executados em uma variedade de plataformas e estruturas. Além disso, eles são caracterizados por sua grande interoperabilidade e extensibilidade, bem como por suas descrições processáveis por máquina, graças ao uso de XML. Os *WebService* permitem sua utilização de maneira combinada para realizar operações complexas, unindo vários serviços simples para fornecer serviços sofisticados com maior valor agregado (ORACLE, 2013).

  Uma das maiores vantagens dos *WebServices* é o fato da comunicação entre as aplicações independer do *hardware* ou da plataforma no qual a aplicação foi implementada e da linguagem de programação em que foi escrita, assim, as interfaces podem ser implementadas sem expor todos os detalhes da programação, como é natural que aconteça. Essa independência permite e incentiva os aplicativos baseados em *WebService* serem implementados a partir de diferentes tipos de tecnologias, e de forma modular, ou seja, a implementação ou alteração de um determinado componente não interfere nos demais componentes com o qual ele se relaciona (AFFONSO, 2015).

  Tais vantagens foram os principais motivos para escolha da utilização da arquitetura no desenvolvimento do trabalho. O *Ready*, utilizará a comunicação via *WebService* para interagir com as aplicações utilizando o estilo de arquitetura REST (*Representational State Transfer*). O item abaixo apresenta informações sobre a arquitetura REST.

  - REST - *Representational State Transfer*

    REST é um estilo de arquitetura de *software* projetado para sistemas distribuídos, particularmente destinado ao mundo da internet WWW (*World Wide Web*). O REST é conhecido por ser um conjunto de princípios de arquitetura para projetar *WebService*, que define um padrão para a transferência de dados de forma didática e representativa, voltado a realizar a comunicação de diferentes programas, capazes

de compartilhar dados entre si, de modo a aparentar ser um sistema único. Para isso, utiliza a estrutura de comunicação cliente e servidor e foca-se nos recursos do sistema, incluindo a maneira como os estados de recursos são endereçados e transferidos em chamadas HTTP (*Hypertext Transfer Protocol*) pelos clientes que se comunicam utilizando diversas linguagens de programação (HALILI; RAMADANI, 2018).

O REST tem a capacidade de estruturar dados que podem estar em formato XML (*Extensible Markup Language*), YAML (*YAML Ain't Markup Language*) ou qualquer outra opção de formato que uma máquina seja capaz de ler, entre eles, o JSON (*JavaScript Object Notatio*n) geralmente o mais usado. REST é conhecido como APIs RESTful ou serviços da *Web RESTful* e segue o paradigma de programação orientado a objetos (SOAPUI, 2020).

A comunicação entre o cliente e servidor via *WebService* acontece da seguinte maneira. O servidor, fornece *endpoints* que os clientes poderão acessar para visualizarem as respostas do servidor. Um *endpoint* é definido como a parte final da URL (*Uniform Resource Locator*) pela qual uma aplicação cliente é capaz de acessar ao serviço. O termo *Endpoint* é proveniente da língua inglesa, para o qual, uma tradução literal pode ser "Pontos de Extremidade". Quando aplicado a área da computação, o *endpoint* faz referência aos terminais de conexão entre uma API (*Application Program Interface*) e o cliente (TECHWALLA, 2018).

A arquitetura REST foi escolhida para como padrão para a comunicação entre as aplicações e o *Ready*, pois é fácil de ser desenvolvida e de baixo custo, além de ter uma manutenção simples (SOAPUI, 2020).

### 2.2.3 Ferramentas Utilizadas

As diferentes ferramentas utilizadas neste trabalho consistem em programas, linguagens e bibliotecas além de outros recursos em *software* existentes e que foram escolhidos para compor o trabalho. Neste escopo, pode-se citar a linguagem utilizada para a implementação do *middleware Ready*, as bibliotecas utilizadas por incorporarem recursos importantes ao sistema, *softwares* e o modelos de simulação do robô.

- ROS - *Robot Operating System*

  O ROS é um *framework open source* voltado a criação de *software* com aplicação em robótica, formado por um conjunto de bibliotecas e ferramentas que visam facilitar o desenvolvimento de *software* para robôs robustos e complexos (ROS, 2017).

  O desenvolvimento de *software* para este fim engloba inúmeros desafios, dentre eles, a necessidade de encontrar soluções que funcionem em diversos ambientes e dispositivos. Trabalhar nestas soluções tem sido suficientemente difícil de modo que nenhum indiví-

duo, laboratório ou instituição espera resolver sem o apoio de alguma comunidade. Deste modo o ROS também possui como objetivo incentivar o desenvolvimento colaborativo (ROS, 2017).

No início de 2010 o ROS realizou o lançamento da primeira versão chamada *ROS Box Turtle*. Atualmente já possui doze versões lançadas oficialmente. Neste trabalho a equipe escolheu trabalhar com a penúltima versão lançada, conhecida como, *ROS Kinetic Kame* por já ser uma versão LTS (*Long Term Support*), ou seja, com suporte estendido e por essa razão ser bastante difundida pela comunidade (ROS DISTRIBUTION, 2010).

O ROS é amplamente distribuído além de ser fortemente utilizado, possui uma integração perfeita com outras bibliotecas da comunidade de *software* e simuladores, como o *Gazebo 3D simulator*, *OpenCV*, PCL *(Point cloud library)*, entre outros. O *Gazebo* será abordado com mais detalhes na subseção 2.2.3.

A infraestrutura de comunicação do ROS é um *middleware* formada por uma interface para troca de mensagens entre processos, que pode gravar e reproduzir mensagens através da biblioteca *rosbagtool*, chamadas de procedimento remoto e um sistema de parâmetros distribuídos.

A organização do sistema ROS é baseada nos conceitos de "*Packet*", ou pacote, "*Node*", ou nó e "*Topics*", ou tópicos. O Pacote é a unidade de organização de arquivos utilizada pelo ROS, e consiste em um diretório que pode reunir executáveis, bibliotecas além de obrigatoriamente possuir um arquivo denominado "*package.xml*" o qual armazena todas as metainformações, além de conter as dependências necessárias para que o pacote funcione corretamente. As metainformações consistem em informações básicas do pacote como por exemplo versão, mantedor, licença, entre outros (ROS.ORG, 2018).

Um "Nó" é basicamente um programa executável contido em um "Pacote" e que tenha sido implementado utilizando a "*ROS client library*", ou seja, a biblioteca que o torna capaz de publicar ou receber mensagens através dos "Tópicos" ROS (ROS.ORG, 2019b).

Por fim os "Tópicos" podem ser resumidos como o canal em que os "Nós" utilizam para publicar e receber mensagens. Eles são construídos utilizando protocolos de rede e permitem que as mensagens sejam escritas de maneira anônima, com o objetivo de permitir desacoplamento entre as partes do sistema que geram dados e as partes do sistema que os consomem (ROS.ORG, 2019a).

Um ponto forte que pode ser citado sobre o ROS é o poderoso conjunto de ferramentas para desenvolvimento que ele possui. Entre elas estão vários recursos capazes de fornecer uma boa depuração, introspeção, plotagem e visualização de variáveis, procedimentos e até o estado do sistema robótico (TSARDOULIAS; MITKAS, 2017).

As duas ferramentas mais conhecidas são *rqt* (para visualização de dados e incorporação de módulos gráficos) e *rviz* (para visualização de experimentos), além de outras, ampla-

mente utilizadas como o *suchrosgraph* e o *rxplot* (TSARDOULIAS; MITKAS, 2017).

Além dos componentes já citados, o ROS possui uma gama de ferramentas capazes de acelerar o desenvolvimento de *software* robótico, alguns dos principais são (TSARDOULIAS; MITKAS, 2017):

- Definições de mensagens padrão para robôs;
- Biblioteca de geometria robótica – Linguagem de descrição do robô;
- Diagnóstico;
- Módulos de localização;
- Algoritmos de mapeamento;
- Módulos de navegação e criação de caminhos;

Em 2015 foi lançado a segunda versão do ROS, denominada "ROS 2" (ROS, 2020a). A principal diferença entre as duas versões consiste na primeira utilizar uma camada de comunicação personalizada que controla os nós baseada no protocolo de rede TCP/IP (*Transmission Control Protocol / Protocolo de Internet*), denominado TCPROS (ROS WIKI, 2013) e a versão mais recente utilizar o DDS (*Data Distribution Service*). O DDS, é um serviço de distribuição de dados que consiste em um padrão de *software* livre para *middleware* (OMG, 2019) e é utilizado pela indústria em aplicação voltadas às áreas críticas como a aviação e na energia nuclear (INFOQ, 2020).

Esta mudança traz ao ROS 2 a capacidade de ser usado em sistemas tempo-real, redes instáveis além de ser mais recomendado para aplicações de Robótica de Enxame (GENERATIONROBOTS, 2019).

Embora mais recente e com mais funcionalidades a equipe utilizará a primeira versão do ROS, pois esta possui uma documentação mais robusta e apresenta maior estabilidade, além disso, os principais conceitos entre as versões são os mesmos (GENERATIONROBOTS, 2019).

- Espeak

Como descrito no capítulo 1, alguns robôs são capazes de imitar a comunicação humana simulando a fala. Para adicionar esta funcionalidade ao sistema, será utilizada a ferramenta *Espeak*.

O *Espeak* é um *software Open Source* capaz de sintetizar fala com suporte a inúmeros idiomas. Este pode ser executado por meio de linha de comando e pode aceitar como parâmetros de entrada arquivos de texto ou uma variável com o conteúdo a ser sintetizado. Este programa utiliza o método "Síntese de Formantes", o que o torna bastante leve, mas produz um resultado pouco natural, por não utilizar amostras de som ou modelos obtidos da fala humana (SOURCEFORGE, 2016).

- *Deepspeech*

  O *Deepspeech* consiste em uma biblioteca desenvolvida em código livre pela *Mozilla* e está disponível na plataforma de versionamento *GitHub*. Possui como objetivo extrair e converter em texto a informação de voz obtida através de uma amostra de som (MOZILLA, 2017).

  A tecnologia *Deepspeech* compõe o sistema com o objetivo de complementar a funcionalidade de "imitar a comunicação humana", sendo a ferramenta utilizada para tornar o robô capaz de "escutar" e entender comandos de voz.

- SLAM - *Simultaneous localization and mapping*

  Alguns modelos de robôs podem executar tarefas que envolvem a locomoção do agentes, motivo pelo qual decidiu-se incluir na arquitetura do sistema a ferramenta SLAM (*Simultaneous Localization and Mapping*).

  O SLAM consiste em um conjunto de programas que torna um robô móvel capaz de mapear o ambiente no qual está inserido e estimar a própria posição enquanto se movimenta (BAILEY; DURRANT-WHYTE, 2006).

  Em outras palavras pode-se dizer que o robô, durante a movimentação, é capaz de atualizar a própria posição e a posição de obstáculos, permitindo-o, por exemplo, planejar trajetórias para determinados objetivos, evitar colisões e conhecer a posição de áreas de interesse no ambiente que o envolve.

  A navegação SLAM é provida pelo pacote *turtlebot3_slam* e a documentação utilizada para configurar e utilizar os pacotes citados nesta seção estão disponíveis na página "*Robotis* Emanual" (ROS, 2018). A utilização foi baseada na documentação disponibilizada em (ROBOTIS, 2020b).

- *TurtleBot*

  O *TurtleBot* é um *kit* de robô pessoal de baixo custo *open source*, voltado ao estudo e desenvolvimento da robótica. Criado em 2010 por Melonee Wise e Tully Foote o *kit* consiste em uma base móvel com sensor de distância 2D / 3D. Ele foi projetado para ter custo reduzido, ser fácil de construir e montar, com produtos prontos para o consumo e possuir peças que podem ser produzidas com material padrão (TURTLEBOT, 2014).

  Além disso, o *TurtleBot* é a plataforma de desenvolvimento padrão utilizada pela comunidade ROS (ROBOTIS, 2020b), que fornece amplo suporte, envolvendo documentação e *software* para o uso em robôs *TurtleBot*, sendo o motivo pelo qual esta plataforma foi escolhida para os testes e desenvolvimento do trabalho.

  Atualmente a plataforma está na 3ª versão, que consiste em três diferentes modelos de robôs, o *Burguer*, o *Waffle* e o *Waffle Pi*. A Figura 7 apresenta os modelos de robôs da família *TurtleBot*.

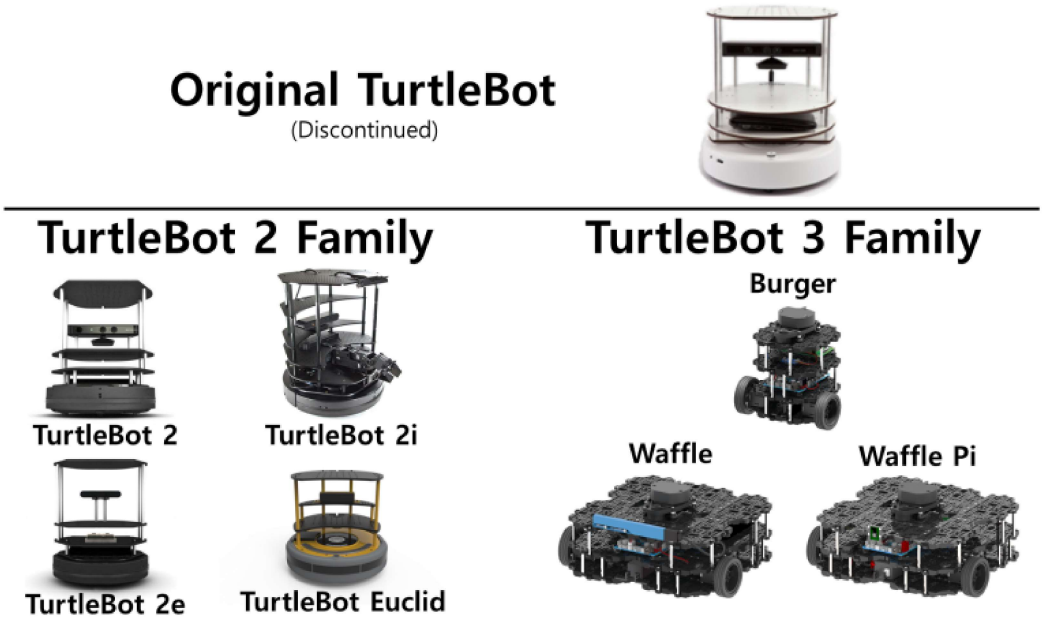



Figura 7 – TurtleBot Family - Fonte: (TURTLEBOT, 2019)

Para esse trabalho, foram utilizadas as ferramentas fornecidas pela comunidade ROS para realizar as simulações do robô *TurtleBot 3 Burger* no ambiente *Gazebo*, apresentado com mais detalhes no item "Gazebo" abaixo, permitindo a realização de testes e desenvolvido sem a necessidade de possuir um robô físico ou um grande espaço para que o robô atuasse.

– *Gazebo*

  O Gazebo é um simulador desenvolvido com o objetivo de atender a necessidade de simular robôs em diferentes ambientes e sob diferentes condições. O início do desenvolvimento ocorreu em 2002, sendo um trabalho desenvolvido pelo Dr. Andrew Howard e pelo estudante Nate Koening (GAZEBOSIM, 2020) e atualmente é mantido pela *Open Robotics* (OPENROBOTICS, 2020).

  Além de ser *open source* e estar em constante desenvolvimento, o Gazebo contém um sistema de simulação física e permite o monitoramento do experimento através de gráficos de alta qualidade (GAZEBOSIM, 2020). A Figura 8 apresenta a imagem da simulação do gazebo que está sendo utilizada neste trabalho.

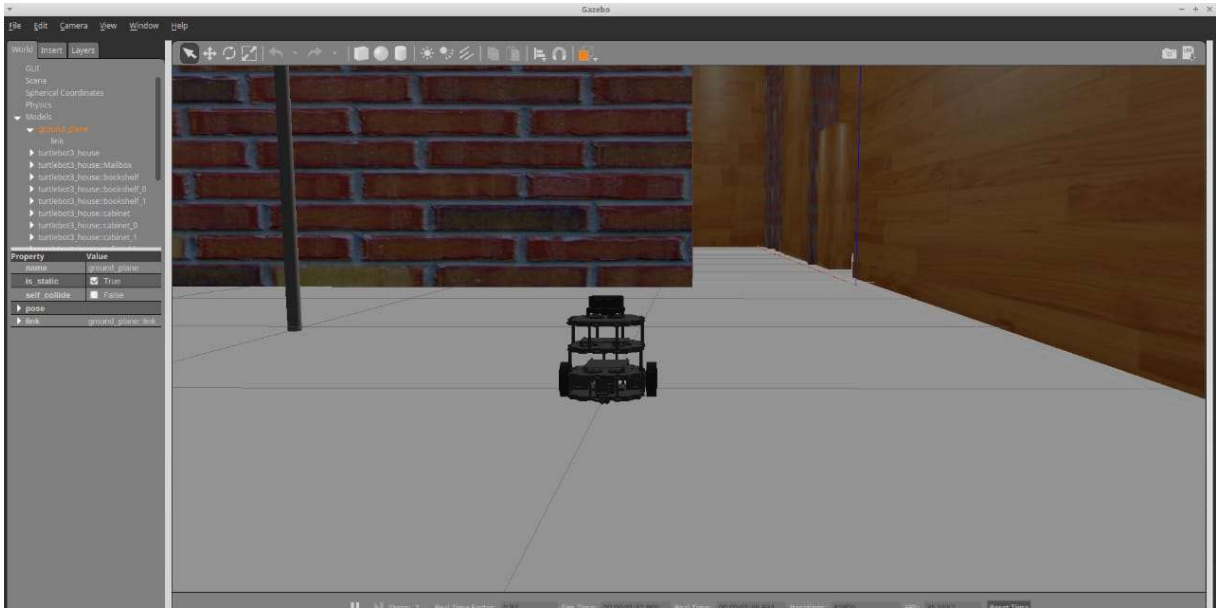



Figura 8 – Gazebo - Fonte: Autoria Própria

- .NET *Framework*

  O .NET *Framework* é uma plataforma de desenvolvimento projetada pela *Microsoft*, com sua primeira versão publicada em fevereiro de 2002. A principal motivação da plataforma é proporcionar um ambiente ideal e produtivo para desenvolvimento de aplicações altamente distribuídas, mas também suportando o desenvolvimento de aplicações *Desktop* (GALUPPO; MATHEUS; SANTOS, 2003).

  A plataforma disponibiliza inúmeras bibliotecas de auxílio e é capaz de suportar diferentes linguagens de programação como C/C++ e *Visual Basic*, dando mais liberdade para os programadores e garantindo a performance de seus sistemas, pois todas elas passam por um processo de compilação para uma linguagem única chamada *Common Language Runtime*(CLR), sendo está a fundação que garante o funcionamento da plataforma, juntamente com outras bibliotecas padronizadas. A estrutura da versão mais moderna do *Framework* pode ser vista na Figura 9, contendo também sua versão multiplataforma chamada .NET *Core* e sua distribuição para dispositivos Unix chamada *Xamarin*.

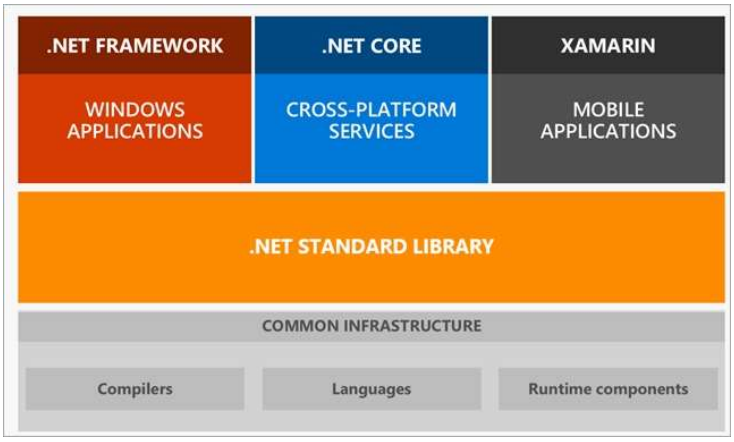



Figura 9 – Estrutura do .NET *Framework* - Fonte: (MICROSOFT, 2020b)

Esta estrutura isola as aplicações do sistema operacional e permite assim que os desenvolvedores foquem em produzir com foco na plataforma, proporcionando uma melhor portabilidade e distribuição de código (GALUPPO; MATHEUS; SANTOS, 2003).

O .NET *Framework* também traz em seu pacote a linguagem de programação C♯, a linguagem de programação escolhida para o desenvolvimento do *framework Ready*, que será explicada no item C♯, abaixo. Neste trabalho a distribuição .NET *Core* foi escolhida por ser compatível com o sistema operacional *Linux*.

– C♯

  O C♯ é uma linguagem de programação orientada a objetos, fortemente tipada desenvolvida pela *Microsoft*. Esta foi escolhida para ser a linguagem de programação de desenvolvimento do trabalho por conter bibliotecas e ferramentas de desenvolvimento de *WebServices* e por ser compatível com a comunicação ROS por meio da biblioteca *Ros-Sharp*, explicada na subseção 2.2.3 (MICROSOFT, 2019).

– Eventos

  O .NET *Framework* conta com um sistema de gerenciamento e criação de eventos. Esta estrutura se baseia no padrão de design do observador (MICROSOFT, 2020c), onde um cliente pode se registrar com um provedor de mensagens, e o mesmo ser notificado por ele. Eventos podem ser acionados de diversas maneiras, podendo ou não envolver o usuário (MICROSOFT, 2020a).

  Um delegado (também chamado de Representante) é o responsável por relacionar eventos às funções (ou clientes) que estão inscritas no evento. Segundo a documentação, "[...] um representante é um intermediário (ou mecanismo do tipo ponteiro) entre a origem do evento e o código que manipula o evento" (MICROSOFT, 2020a).

– Biblioteca Ros-Sharp

  *Ros-Sharp* é um conjunto de bibliotecas e ferramentas de *software* de código aberto desenvolvido em linguagem de programação C♯ que tem como objetivo de realizar

a comunicação entre o ROS e aplicativos .NET, em particular o *Unity* (SIEMENS, 2018).

A biblioteca foi desenvolvida pela Siemens e está disponível abertamente no *GitHub*. Há também uma página no estilo *Wiki* onde se encontram as definições em alto nível e alguns exemplos de como utilizar as funções da biblioteca (ROS SHARP WIKI, 2017).

Para realizar a comunicação com o ROS, a biblioteca possui modelos de objetos equivalentes às mensagens ROS e implementa funções que possibilitam a comunicação através do sistema de troca de mensagem do ROS.

– ASP.NET

O ASP.NET é uma extensão do .NET *Framework*, tratando-se de um conjunto de ferramentas e bibliotecas com foco em desenvolvimento *web*. Serviços como processamento de requisições HTTP e autenticação fazem parte do pacote, auxiliando a construção de ambientes *Backend* modernos (MICROSOFT, 2020d).

Uma das estruturas que sustentam o *framework Ready* se encontra no pacote ASP.NET, sua *Web*API RESTful presente no pacote ASP.NET MVC, implementando a arquitetura MVC (*Model View Controller*). As *Controllers* disponibilizadas pelo pacote são responsáveis por gerenciar as requisições HTTP geradas pelas aplicações que se conectam com o *framework*. Detalhes sobre o funcionamento podem ser encontrados na seção 3.2.3.

- Banco de Dados

O *Ready*, é um *Framework* que objetiva facilitar a inclusão de novas aplicações em robôs domésticos, podendo conter inúmeras aplicações instaladas, de acordo com o desejo do usuário e da capacidade do robô. Portando, para que o mesmo funcione corretamente, é necessário que o trabalho seja capaz de guardar algumas informações em memória, como o nome e objetivo das aplicações instalada, bem como as palavras aprendidas e utilizadas pelo robô durante a comunicação.

Para isso decidiu-se utilizar o *MySQL*, por ter licença livre e ser conhecido por todos os membros da equipe, mais detalhes sobre a ferramenta serão apresentados no item abaixo:

– *MySql*

O *MySQL* é um Servidor e Gerenciados de Banco de Dados, também conhecido pela sigla SGBD, *open source*, que possui características de gerenciamento e gestão de dados e multi acesso (MILANI, 2007).

Ele é conhecido por ser um banco de dados relacional, ou seja, que apresenta as informações em forma de tabelas, que utiliza a linguagem de programação SQL (*Structured Query Language*), implementada por meio de códigos e funções que

foram atomizadas pelos desenvolvedores e a tornam uma linguagem extremamente rápida (MILANI, 2007).

Além disso, o *MySql* conta com um recurso denominados *Sounds Like*, capaz de comparar duas palavras a partir da sonoridade da mesma (ORACLE, 2020). Fator que foi fundamental para a escolha da ferramenta, uma vez que auxilia na detecção de palavras durante a comunicação com o robô.

# 3 Desenvolvimento

Este trabalho propõe um *framework* denominado *"Ready"*, uma arquitetura de *software* organizada por módulos voltados a fornecer funcionalidades e tarefas especializadas às aplicações, bem como intermediar a comunicação destas com os respectivos serviços que são implementados utilizando os recursos do ROS. No escopo deste trabalho, serviços são programas responsáveis em formatar as informações obtidas do robô, permitindo que o *middleware* possa utilizá-las. Além disso os serviços comandam diretamente o robô de acordo com as requisições realizadas pelo *middleware*.

Os módulos citados no parágrafo anterior estão contidos em um *middleware*, denominado "*middleware Ready*", um *software* que se comunica e captura as informações provenientes dos serviços para prover funcionalidades mais elaboradas com um maior nível de abstração aos desenvolvedores de aplicações. Dentre estas funcionalidades pode-se citar padrões de movimentação, gerenciamento de recursos do robô, algumas ferramentas de comunicação em linguagem natural entre outros recursos.

Além do *middleware*, o *framework Ready* é composto por bibliotecas que permitem as aplicações acessarem os recursos fornecidos pelo *middleware*. Cada biblioteca se conecta a um respectivo submódulo que compõe o "*middleware Ready*", organizando o acesso às funcionalidades do *framework* para os desenvolvedores. Mais detalhes sobre o funcionamento dos módulos bem como da respectiva biblioteca que o acessa serão escritos no decorrer deste capítulo.

Outro elemento importante na arquitetura deste trabalho são os já citados serviços, pois fornecem dados e controle dos atuadores do robô ao *framework Ready*, por meio dos tópicos do ROS. Considerando que cada robô possui uma arquitetura mecânica única, os serviços são um meio de padronizar a entrada destes dados para que o *framework* possa funcionar em diferentes dispositivos, portanto idealmente, estes programas devem ser escritos pelos fabricantes e desenvolvedores dos robôs.

A Figura 10 apresenta os níveis de abstração dos componentes envolvidos no trabalho. A barra representada pelo número 5, denominada como *drivers/hardwares* é o menor e a barra de aplicações, representada pelo número 1 é o mais alto nível de abstração dos componentes envolvidos no trabalho. Desde modo, os termos "alto nível" e "baixo nível" podem aparecer no decorrer do trabalho complementando a palavra "abstração".

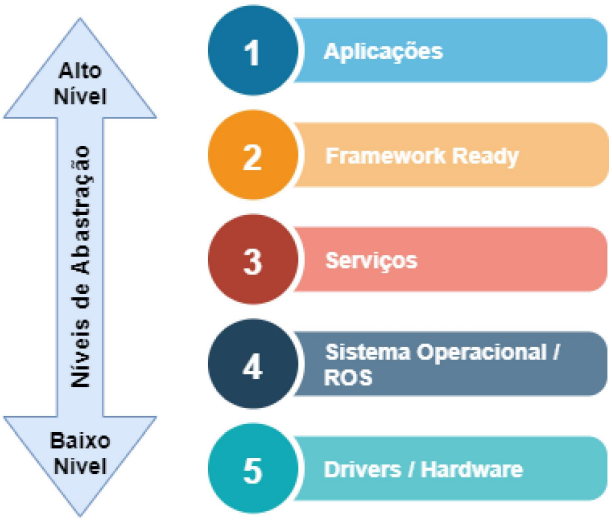



Figura 10 – Representação dos "Níveis de Abstração" dos componentes envolvidos no trabalho - Fonte: Autoria Própria.

Entre os níveis intermediários da Figura 10, logo acima dos *Drives/Hardwares*, há o sistema operacional / ROS e os serviços, exatamente nesta ordem de abstração, considerando do mais baixo ao mais alto nível.

Acima dos serviços encontra-se o *Framework Ready*, uma camada de abstração que resume o ROS, e todas as camadas abaixo dele, ao programador de aplicações para robôs. Desde modo, o desenvolvedor não terá a necessidade de conhecer as implementações e a estrutura do sistema que utilizam o ROS, pois as *Daemons* do *Ready*, (ver subseção 3.2.5), as encapsularão, para entregar funcionalidades mais acessíveis e diretas ao desenvolvedor de aplicações. Na Figura 11, há um novo diagrama do que ilustra a comunicação entre os componentes do trabalho.

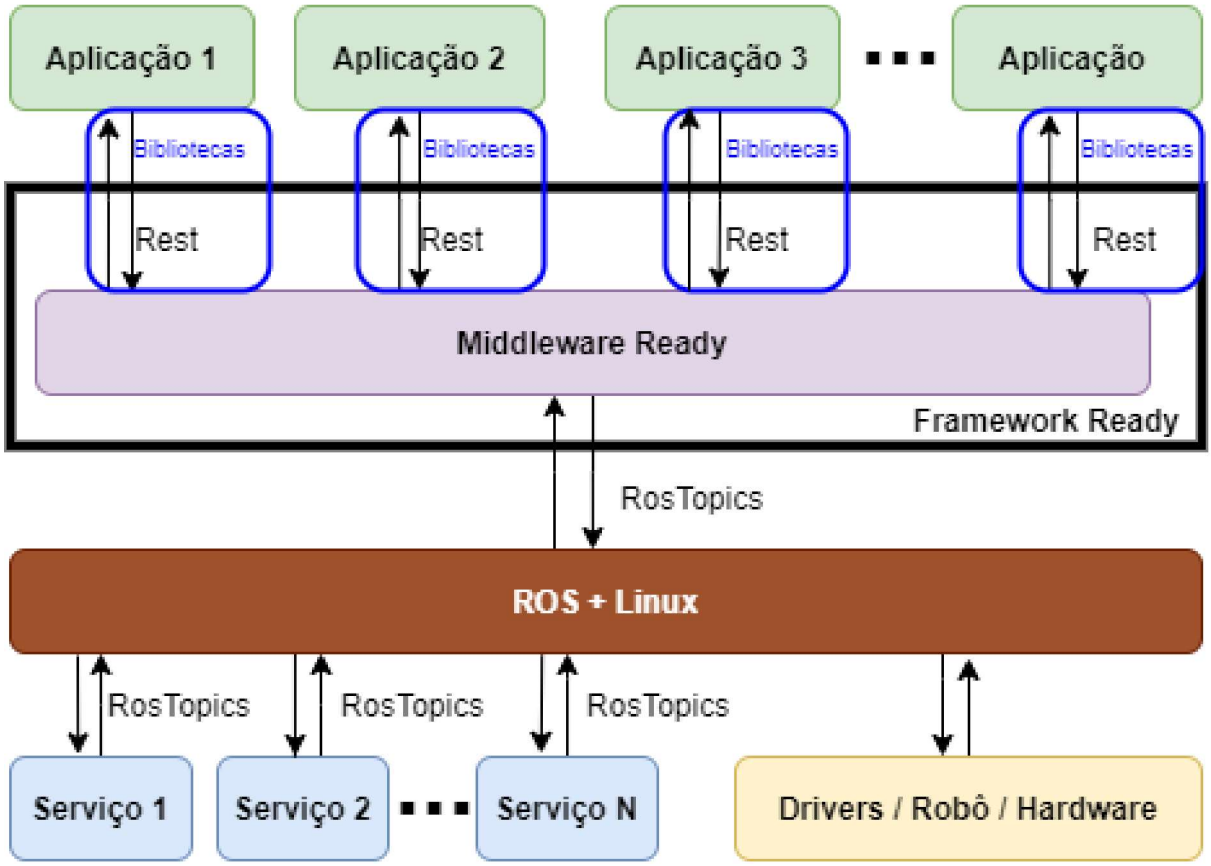



Figura 11 – Diagrama de blocos da comunicação entre os componentes do *framework Ready*- Fonte: Autoria Própria.

Na primeira etapa superior, em cor verde, encontram-se os blocos que ilustram as inúmeras aplicações que podem funcionar no *Framework Ready*. As flechas que conectam as aplicações com o *Middleware Ready* na Figura 11 representam os canais de comunicação em que as trocas de dados destes elementos ocorrem, baseado na arquitetura REST.

Logo abaixo, é mostrado o bloco do *Framework Ready*, que é composto pelo *Middleware Ready* e pelas bibliotecas implementadas para fornecer a comunicação entre o *Middleware* e as aplicações utilizando para isso a arquitetura REST. O *middleware* recebe as solicitações das aplicações, e encaminha para os serviços. Esta interação ocorre através dos tópicos do ROS.

O ROS, juntamente com o Sistema operacional *Linux*, está em contato direto com os *Drivers* e o *Hardware* do robô, assim como com os serviços, descrito na subseção 3.1.

Embora o *Framework Ready* tenha sido planejado para ser executado em um mesmo computador, é possível distribuir o funcionamento do sistema em diferentes dispositivos através de rede interna devido ao fato de que os diferentes elementos do projeto utilizarem tanto o protocolo de comunicação do ROS quando a tecnologia REST, capazes de fornecer esta funcionalidade.

## 3.1 Serviços

São definidos como serviços os programas que atuam no nível de operação do robô, capazes de acessar diretamente os componentes físicos do dispositivo, como por exemplo os motores, atuadores e sensores, além de fornecer comportamentos mais complexos, como a navegação e a interpretação de sinais de voz.

Para viabilizar este Registros, foi definido que estes programas sejam desenvolvidos pelos fabricantes ou projetistas dos robôs, e que sejam compatíveis com os tópicos *ROS* utilizados pela arquitetura.

Os serviços, em resumo, propiciam ao *Middleware* o acesso ao *hardware* do robô, provendo informações pertinentes ao funcionamento do sistema e a possibilidade de controle dos respectivos componentes físicos. Nesta camada, o *ROS* se faz presente, pois o mesmo possui inúmeros pacotes implementados pela comunidade capazes de fornecer diversas funções ao robô, que podem ser adaptados como serviços para o *framework Ready*.

O ROS permite a troca de informações entre diferentes programas por meio de estruturas denominadas “Tópicos”. Este canal de informações utiliza o protocolo TCP/IP como base, conforme explicado no capítulo 2. Os tópicos possuem uma estrutura padronizada, com uma documentação, definindo os tipos de dados e a disposição destes, tornando mais simples a adaptação de serviços *Ready* utilizando como base programas desenvolvidos para o ROS.

Utilizando o sistema de troca de mensagem do ROS, os serviços captam as informações do robô e as adequam ao formato de tópico esperado pelo *Middleware*. Como o ROS está envolvido tanto no gerenciamento das atividades do robô como na comunicação com a respectiva *Daemon*, estes serviços podem ser implementados com qualquer linguagem de programação suportada pelo ROS.

Para ilustrar este conceito foram implementados serviços tanto serviços relacionados a Iteração Humano-Computador quanto serviços relacionados a navegação e locomoção do robô. Ressalta-se que, em futuros trabalhos, ao se adicionar novas *Daemons* ao *middleware*, novos serviços que as alimentariam deverão ser criados.

Os serviços de Iteração Humano-Computador têm como objetivo fornecer ao sistema a capacidade de simular a comunicação humana, e é separada em dois serviços, um responsável pela escuta e outro responsável pela fala, que serão descritos com mais detalhes nas subseções 3.1.1 e 3.1.2 respectivamente. Já os detalhes sobre o serviço de navegação, podem ser encontrados na subseção 3.1.3.

### 3.1.1 Serviço de escuta (reconhecimento de voz)

O serviço de escuta é responsável por detectar a fala humana capturando amostras de voz obtidas por meio de microfones e as convertendo em texto. Para a implementação deste

serviço foi utilizada a linguagem de programação *Python* e a biblioteca *DeepSpeech*.

Ao detectar um determinado nível de amplitude do sinal obtido do microfone que existe no robô, o sistema inicia a aquisição do som. Quando este sinal volta a estar abaixo de uma amplitude mínima, o robô utiliza as ferramentas fornecidas pela biblioteca *DeepSpeech* para converter o sinal obtido em texto. Finalizando a análise, o serviço publica o texto obtido no tópico "*voiceInput*".

Nenhum processamento ou filtragem de palavras é realizado no serviço, apenas a aquisição da informação. A responsável por interpretar e validar esta informação é a *Daemon* de Iteração Humano-Computador , que será apresentada na subseção 3.2.5.3.

### 3.1.2 Serviço de Fala

Toda vez que uma aplicação solicita que um determinado texto seja pronunciado pelo robô, a *Daemon* de Iteração Humano-Computador irá receber esta requisição e encaminhar para o tópico ROS "*voiceOutput*". Este tópico é monitorado constantemente pelo serviço de fala, que também foi desenvolvido utilizando a linguagem de programação *Python*, e este, ao detectar um texto, irá sintetizá-lo por meio da biblioteca "*eSpeak*".

### 3.1.3 Serviço de navegação e locomoção

Este serviço tem como objetivo fornecer ao *middleware* controle sobre a locomoção do dispositivo. Este serviço deve conter não só o controle direto de locomoção, mas também as ferramentas de navegação.

Neste trabalho, os serviços são baseados nos programas fornecidos pelo fabricante do *TurtleBot*, e podem ser considerados um exemplo de como um pacote ROS já existente pode ser adaptado como um serviço para o framework Ready. Para este serviço, muitas das funcionalidades disponibilizadas pelo SLAM e pelo *TurtleBot* foram utilizadas para conectar o sistema aos componentes físicos do robô.

Para acessar a navegação direta do robô, ou seja, controlar a sua velocidade linear e angular, o *TurtleBot* disponibiliza o tópico "*/cmd_vel*", permitindo assim um controle do deslocamento de forma precisa de acordo com as necessidades da aplicação.

O SLAM disponibiliza navegação autônoma utilizando um mapa do ambiente construído na memória do robô. Utilizando o tópico */move_base_simple/goal*", é possível enviar uma coordenada deste mapa ao robô para que este planeje uma rota desviando de obstáculos que possivelmente estarão em seu caminho. Durante a execução do percurso, através dos sensores, o robô é capaz de capturar mudanças no ambiente e atualizar o mapa.

O *TurtleBot* também disponibiliza acesso à sensores de distância, que são capazes de detectar obstáculos ao redor do robô. Para acessá-los, utiliza-se o tópico "*/scan*", que contém 360 valores distintos disponíveis, cada um sendo equivalente a um ângulo em torno do robô.

Mais detalhes de como este serviço se comunica e é utilizado pelo sistema podem ser encontrados na seção 3.2.5.4

## 3.2 *Middleware*

Desenvolver aplicações que possam ser utilizadas em outros modelos de dispositivos, sem conhecer os detalhes de funcionamento do robô e evitando conflitos com o restante do sistema são alguns dos resultados que se espera alcançar com o desenvolvimento desde Registros. Para isso, o *Middleware Ready* assume a funções fundamentais, intermediando a troca de informação entre as aplicações e a estrutura de *software* responsável pelo controle das atividades do robô.

O *Middleware Ready* utiliza as informações dos robôs fornecidas pelos serviços, agrupando as funcionalidades do sistema e organizando-as para entregar funções mais complexas, como por exemplo, funcionalidades relativas a simulação da comunicação em linguagem natural, permitindo que o desenvolvedor concentre-se apenas nos objetivos das aplicações, sem precisar investir tempo nestas funções. Para isso, o *Middleware Ready* possui vários componentes, que serão apresentados na subseção 3.2.1.

### 3.2.1 Arquitetura Interna

A visão geral do sistema é representada pelo diagrama de blocos apresentado na Figura 12, que resume o diagrama de classes completo disponível no anexo A. Por ela, é possível perceber que o sistema é composto por objetos *Daemons*, classes *singletons* e *controllers*.

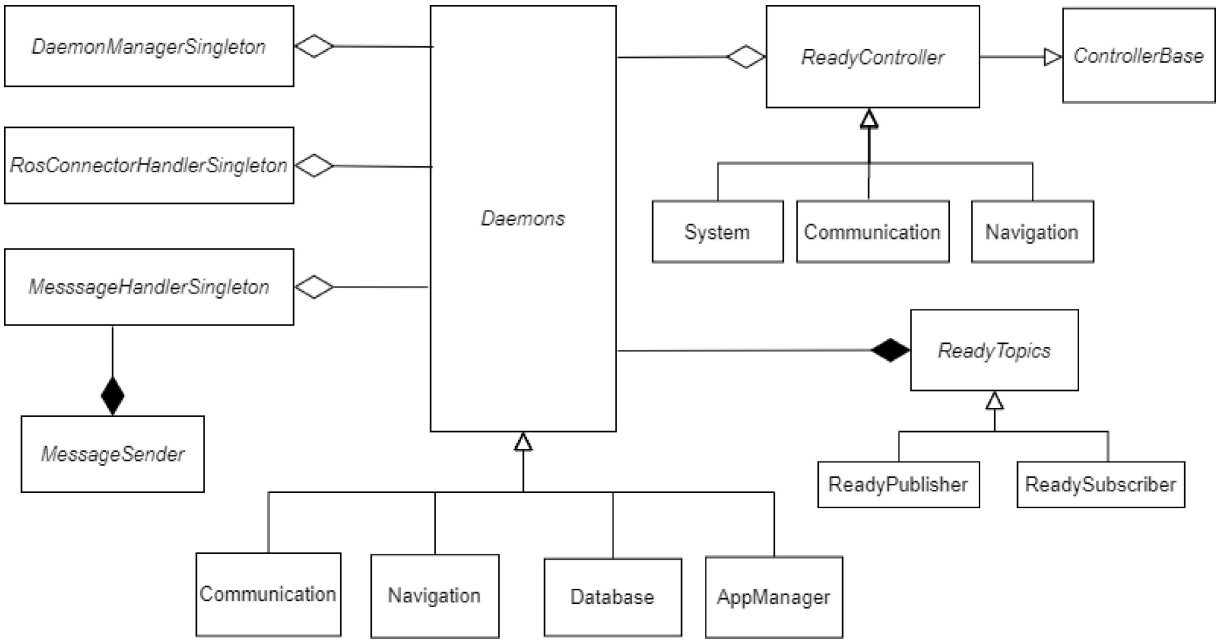



Figura 12 – Diagrama Resumido das Classes - Fonte: Autoria Própria.

As *Daemons*, são os objetos principais do Registros, são responsáveis por capturar as informações dos serviços, encapsular em conjuntos de funcionalidades e fornecê-las aos desenvolvedores de aplicativos, além de controlar os acessos e inicialização de aplicações e permitir a extensão das funcionalidades do *middleware*, de acordo com as necessidades e capacidades dos diversos tipos de robôs.

Na versão atual do *framework Ready* foram implementadas quatro *Daemons* sendo elas: A *daemon* Humano-Computador, a *daemon* Navegação, a *daemon* Registros e a *daemon App-Manager*, que serão explicadas com mais detalhes na seção 3.2.5.

As classes *singletons* são estruturas que possuem como objetivo gerenciar as funcionalidades das *Daemons*. Foram implementadas três classes com esta característica, sendo elas as *DaemonManagerSingleton*, *MessageHandlerSingleton* e *RosConnectorHandlerSingleton*. Os detalhes de funcionamento bem como da implementação serão explicados na seção 3.2.2.

As classes *Controllers* possuem uma função fundamental para o sistema. Elas são responsáveis por definir os *endpoints* pelos quais as aplicações entrarão em contato com o Ready, ou seja, são elas que fornecem o acesso às aplicações. Os detalhes do funcionamento das *controllers* serão explicados na subseção 3.2.3.

### 3.2.2 *Singletons* Utilizados

O sistema possui três classes *Singleton*, denominadas como *DaemonManagerSingleton*, *MessageHandlerSingleton* e *RosConnectorHandlerSingleton*. Estes têm como objetivo gerenciar o funcionamento de cada *Daemon* além da Iteração Humano-Computador entre elas e tópicos dos ROS. O *DaemonManagerSingleton* é responsável por registrar e listar todas as *Daemons* ativas no sistema, bem como validar a possibilidade de inicialização de uma *Daemon* através de um método chamado "*logIn*".

Quando o sistema é iniciado, cada *Daemon* deve passar por uma avaliação para verificar se ela possui um nome único no sistema, ou seja, o *DaemonManagerSingleton* irá confirmar se o nome da nova *Daemons* já está na lista de registro de *Daemons*, comprovando se o mesmo nome está ativo. Se o nome já existir o objeto não é inicializado e é destruído por estar em duplicidade e ser um possível risco ao sistema. Caso a nova *Daemon* seja aprovada, o nome dela é inserido na lista de referências de *Daemons* e ela é inicializada. O fluxo da inicialização de uma nova *Daemon* foi retratado na Figura 13.

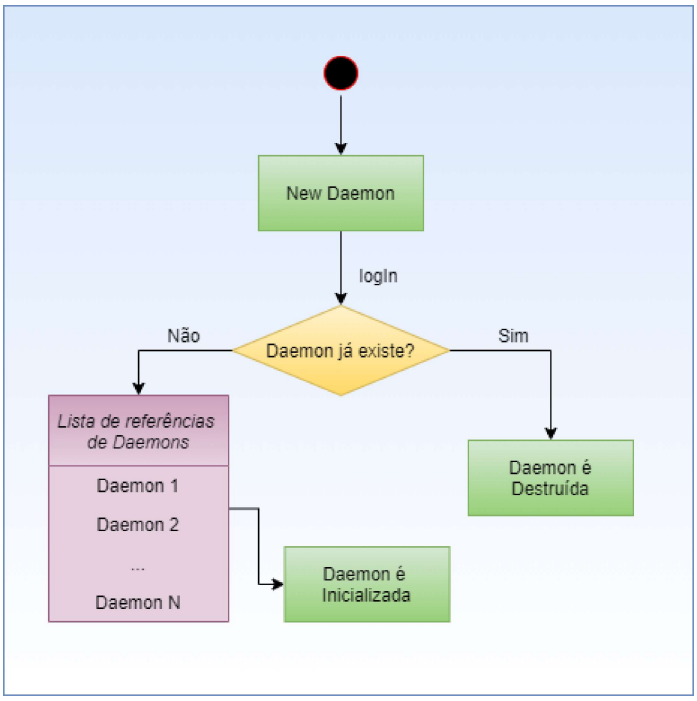



Figura 13 – Fluxo da inicialização de uma nova *Daemon* - Fonte: Autoria Própria.

O *MessageHandlerSingleton* é responsável por gerenciar o fluxo de troca mensagens entre as *Daemons*, pois o evento de disparo de mensagens pertence a ele. Detalhes sobre a comunicação interna das *Daemons* serão descritos na subseção 3.2.5.1.

Por fim, o *RosConnectorHandlerSingleton* tem a responsabilidade de gerenciar os tópicos suportados pelo sistema, mantendo a sessão de todos os objetos que possuem conexão com o *ROS*, além de ser responsável por impedir a entrada de mais de um tópico com o mesmo nome e tipo. Este *singleton* será abordado novamente na subseção 3.2.4.

### 3.2.3 *Ready Controllers*

As classes *controllers* possuem a tarefa de armazenar os *endpoints*, disponíveis nos Registros, que consistem em interfaces utilizadas pelas aplicações para realizar a comunicação com o *Framework Ready*. Para cada *endpoint*, existe uma função responsável por encaminhar a mensagem recebida da aplicação para a *Daemon* correta, onde a mensagem será tratada e consequentemente gerará uma resposta para a aplicação.

Deste modo, as *Controllers* da *WebApi* do *Framework Ready* são a porta de entrada para o sistema, pois elas são as responsáveis por receber e gerenciar as requisições geradas pelas aplicações, proporcionando acesso aos recursos geridos pela arquitetura. Elas são herdeiras da classe *ControllerBase*, disponibilizada pelo pacote MVC (*Model View Controller*) ASP.NET, ou seja, recebem as mesmas características e funções da classe *ControllerBase*, mas também possuem algumas particularidades, por isso foram denominadas *Ready Controllers*.

A *ReadyController* é a classe base capaz de armazenar a instância e acessar funções *Daemons* do sistema, obtidas através do *DaemonManagerSingleton*. Durante a inicialização das classes que herdam o padrão da *ReadyController*, elas devem conectar-se a *Daemon* desejada. A Figura 14 apresenta esta estrutura:

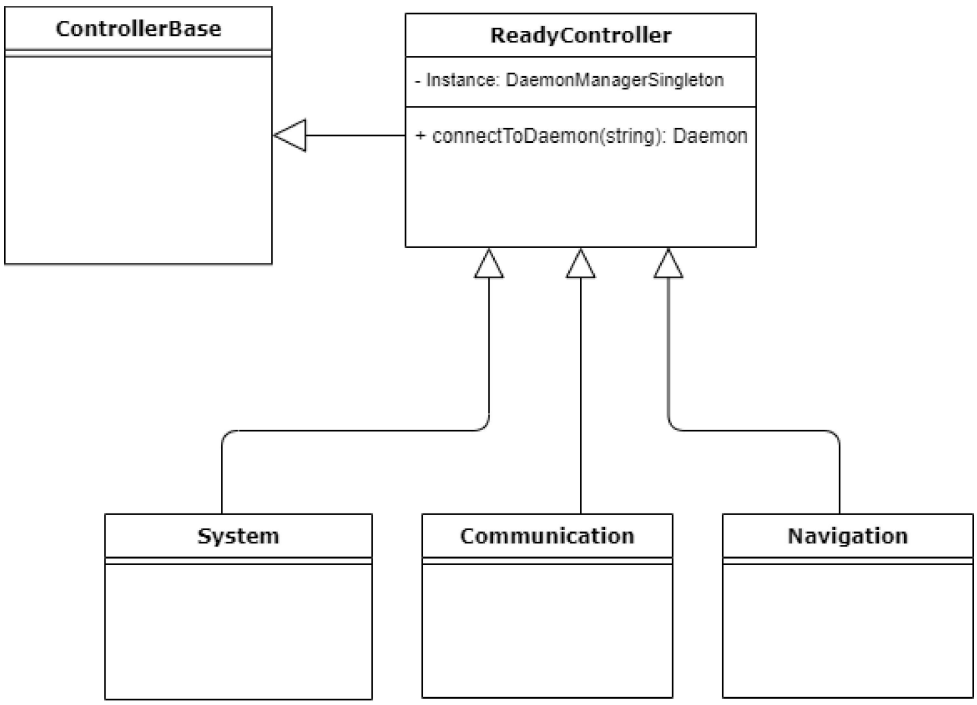



Figura 14 – Estrutura das *Ready Controllers* - Fonte: Autoria Própria.

A lista abaixo apresenta as *Ready Controllers* que foram desenvolvidas neste Registros e suas funcionalidades serão especificadas em conjunto com as *Daemons* com as quais as mesmas conectam.

- *SystemController* (detalhada na subseção 3.2.5.2)
- *CommunicationController* (detalhada na subseção 3.2.5.3)
- *NavigationController* (detalhada na subseção 3.2.5.4)

#### 3.2.3.1 Comunicação e parâmetros

Para padronizar a conexão com o *Middleware Ready*, decidiu-se que todas as chamadas HTTP aos *endpoints* são necessariamente do tipo POST, ou seja, é possível enviar informações de forma serializada para a *WebAPI* e estas serem consumidas pelo *Middleware*.

Como forma de garantir que todas as informações cheguem ao *Middleware* corretamente, foi elaborado um modelo de objeto chamado *ReadyDataPackage*, que tem como objetivo encapsular informações pertinentes ao *Ready* e validar as chamadas. O modelo do objeto está representado na Figura 15.

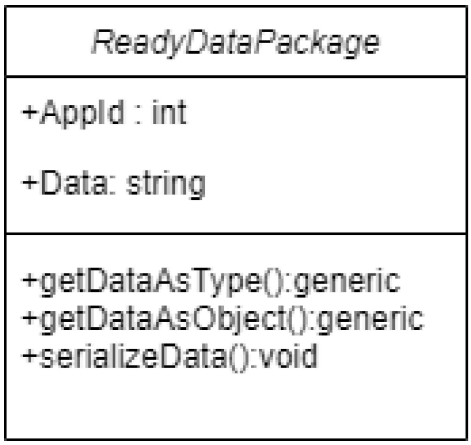



Figura 15 – Classe *ReadyDataPackage* - Fonte: Autoria Própria.

A ideia por trás deste modelo foi criar algo genérico o suficiente para ser recebido pelo *Middleware* e garantir uma comunicação para as aplicações, sendo ele facilmente replicável em outras linguagens e mantendo os parâmetros serializados na propriedade "*Data*".

As Bibliotecas *Ready*, abordadas na seção 3.3, são as responsáveis por serializar os parâmetros necessários para cada *endpoint*. No *middleware*, caso o parâmetro seja apenas um tipo primitivo, o *endpoint* utilizará a função *getDataAsType* para extrair as informações do pacote, informando o tipo de objeto esperado. Caso os dados estejam agrupados em um objeto, utiliza-se a função *getDataAsObject*, também informando a classe esperada.

### 3.2.4 *Ready Topics*

Utilizando as ferramentas fornecidas pela biblioteca *Ros-Sharp* 2.2.3, foi desenvolvida uma classe genérica chamada *ReadyTopic* com o intuito de melhorar o suporte as estruturas disponibilizadas pelo *ROS* e organizá-las de acordo com a proposta do Registros. Desta classe, são herdadas as estruturas *ReadyPublisher* e *ReadySubscriber*, que possuem a finalidade de publicar mensagens em tópicos *ROS* e recebê-las, respectivamente.

O gerenciamento dos tópicos suportados pelo sistema é realizado pelo *RosConnectorHandlerSingleton*, que mantém a sessão de todos os objetos que possuem conexão com o *ROS*, além de ser responsável por impedir a entrada de mais de um tópico com o mesmo nome e tipo.

Estes tópicos são utilizados pelas *Daemons* para realizar a comunicação com os serviços desenvolvidos pelo fabricante do robô, permitindo assim que o *middleware* seja capaz de acessar e alterar os recursos do robô conectados ao *ROS*.

### 3.2.5 *Daemons*

Os módulos são definidos no escopo deste trabalho como *Daemons*, denominação escolhida devido à similaridade de funcionamento destes subprocessos em relação ao conceito original de *Daemon* empregado no escopo dos Sistemas Operacionais Multitarefa, que descreve programas que funcionam em segundo plano, ou seja, atuam sem prioridade máxima e sem acesso a todos os recursos, aguardando alguma requisição de serviço sem depender de entradas de um usuário (ARAUJO; MAZIERO; NIEVOLA, 2011).

Deste modo, neste trabalho, o termo *Daemon* é empregado a todos os objetos herdeiros da classe abstrata "*Daemon*". Esta classe contém, entre outros métodos, um método abstrato principal chamado *Run()*, que é executado no momento da inicialização do objeto e deve ser implementado pelo programador ao desenvolver um submódulo, sobrecarregando o método da classe herdada.

Com a finalidade de garantir a comunicação entre as *Daemons*, foi desenvolvido um protocolo que as permite trocar informações sem sacrificar o funcionamento independente das mesmas. Deste modo, é possível desenvolver uma arquitetura escalonável, a qual admite que as *Daemons* possam ser adicionadas ou removidas afetando apenas as tarefas envolvidas na modificação. Esta comunicação interna das *Daemons* será explicada com mais detalhes na subseção 3.2.5.1.

A comunicação entre *Daemons* visa flexibilizar o desenvolvimento de tarefas internas ao sistema, que combinem diferentes funcionalidades. Esta comunicação é denominada "Interna", pois intermedia apenas as comunicações entre as próprias *Daemons*.

Os componentes envolvidos na comunicação interna das *Daemons* juntamente com outros elementos que permitem as demais funcionalidades do sistema, serão apresentadas na subseção 3.2.5.1.

#### 3.2.5.1 Comunicação Interna

Quando inicializada, a *Daemon* se comunica com o *MessageHandlerSingleton* para registrar seu método *receiveMessage* no Evento responsável pelo disparo de mensagens, recebendo sempre uma notificação quando o evento for acionado. Através do *MessageHanderSingleton*, as *Daemons* são capazes de encapsular e manipular o acionamento do Evento de disparo de mensagens, sendo assim capazes enviar mensagens através do método *SendMessage*.

As mensagens consistem em um objeto que possuem as informações "*Daemon* de Origem", que determina qual a *Daemon* que disparou a mensagem, a "*Daemon* Destinatário", que determina qual *Daemon* deve consumir a mensagem, o "Comando" que deve ser entendido como a entrada da *Daemon* destinatário e os "Parâmetros", uma lista com os dados utilizados pelo comando. Cada *Daemon* possui a própria lista de comandos, os quais podem disparar diferentes comportamentos, além de permitir que estas forneçam diferentes tipos de informações.

Todas as mensagens enviadas ficam armazenadas em uma fila aguardando serem acessadas. Embora todas as *Daemons* componentes do sistema sejam capazes de acessar esta fila, as mensagens só podem ser consumidas pelo respectivo destinatário. Há um caso especial de mensagem denominado "*Broadcast*" que só pode ser criado pela *Daemon* "*AppManager*" e é capaz de ser acessada por todas as demais *Daemons*.

As *Daemons* contam com dois métodos de envio, os métodos *SendMessage* e *SendMessageAndWait*. O primeiro envia uma determinada mensagem à outra *Daemon* sem esperar uma resposta, enquanto o segundo deixa a chamada em espera até obter uma resposta.

Visando comprovar o conceito deste sistema, algumas *Daemons* foram implementadas e serão explicadas nas subseções a seguir: *Daemon AppManager* (subseções 3.2.5.2), *Daemon de Iteração Humano-Computador* (subseções 3.2.5.3), *Daemon de Navegação* (subseções 3.2.5.4) e *Daemon de Registros* (subseções 3.2.5.5).

Ressalta-se que outras *Daemons* e funcionalidades podem ser incluídas no *Framework Ready*, pois o mesmo é expansivo horizontalmente. Porém, neste momento, foram criadas apenas quatro de modo a demonstrar as principais funcionalidades do *framework Ready*.

#### 3.2.5.2 *Daemon AppManager*

A *Daemon AppManager* é responsável por permitir que o sistema *Ready* seja capaz de gerenciar a execução das aplicações e distribuir alguns recursos do robô entre estas. Funções importantes como a execução, listagem e o cancelamento de uma aplicação foram consideradas como o cerne de funcionamento desta *Daemon*. Além disso, ela também realiza o controle de recursos protegidos do robô através de uma convenção de dois tipos distintos de execução para as aplicações: Primeiro Plano e Segundo Plano.

Define-se como "Primeiro Plano" o modo de execução que permite que o aplicativo seja capaz de controlar recursos privilegiados do robô que são considerados críticos, como por exemplo as funções de Movimentação. Outras funcionalidades que porventura possam ser implementados em versões futuras do sistema também podem ser incluídas nessa categoria, quando requisitarem recursos físicos de acesso exclusivo. Somente uma aplicação pode estar em Primeiro Plano por vez.

Quando submetido ao modo de execução "Segundo Plano", a aplicação possui acesso a todos os recursos do sistema, com exceção dos recursos citados no parágrafo anterior. Este controle se faz necessário para que apenas uma aplicação possua controle sobre funcionalidades críticas por vez, pois, como trata-se de um dispositivo mecânico, garantir o comportamento esperado pela aplicação e evitar movimentações e atuações erráticas que possam resultar em mal funcionamento ou possíveis acidentes.

A política de alocação de planos é realizada da seguinte maneira: Ao ser aberta, uma aplicação é colocada em segundo plano, entretanto, esta é capaz de solicitar ao sistema ser

alocada ao primeiro plano se necessitar dos recursos protegidos. Caso mais de uma aplicação realize a solicitação, esta é enfileirada e aguarda em sua posição na fila de espera, até que os recursos solicitados sejam liberados pelas aplicações que entraram antes na fila. Ressalta-se que o desenvolvedor das aplicações deve conhecer esta política a fim de desenvolver aplicações capazes de lidar com estas esperas de sistema.

Deste modo, a *Daemon* permite que a aplicação seja capaz de solicitar e manter a execução em primeiro plano por quanto tempo for necessário, caso este tenha sido reservado. Quando não for mais preciso utilizar os recursos críticos do robô, é possível liberá-los solicitando novamente a execução em segundo plano. A *daemon AppManager* possui uma lista de todas as aplicações abertas no sistema, além de ter acesso aos *status* das requisição de foco de cada uma delas. A partir deste *status*, a *AppManager* é capaz de verificar se o recurso critico esta ou não em uso. Em cada função que executa alguma atividade critica do sistema, é primeiramente realizada um verificação para validar se a aplicação requisitante é realmente a aplicação de primeiro plano. Abaixo encontra-se uma lista com todos os *status* possíveis para a requisição de foco da aplicação junto.

- *Unset*: Aplicação que não solicitou o primeiro plano;
- *Pending*: Status da aplicação que solicitou o primeiro plano e esta aguardando;
- *Executing*: Aplicação atuando em primeiro plano;
- *Finished*: Aplicação que finalizou a utilização dos recursos criticos;
- *Locked*: Aplicação em primeiro plano, que solicitou o bloqueio dos recursos criticos;

Para evitar que uma aplicação reserve o primeiro plano e não utilize nenhum recurso, privando as outras aplicações, foi implementado um mecanismo na *Daemon* denominado "controle de requisição", que verifica se os recursos críticos estão sendo utilizados ou não. Caso a aplicação que estiver em primeiro plano não utilize nenhum recurso crítico por cinco segundos, ou seja, permaneça no *status* "*Pending*" durante este tempo arbitrário que a equipe julgou como suficiente para a utilização da função, a *daemon* irá colocá-la em segundo plano e concederá o lugar à próxima da fila.

A Figura 16 apresenta um diagrama que ilustra o funcionamento desta *Daemon*.

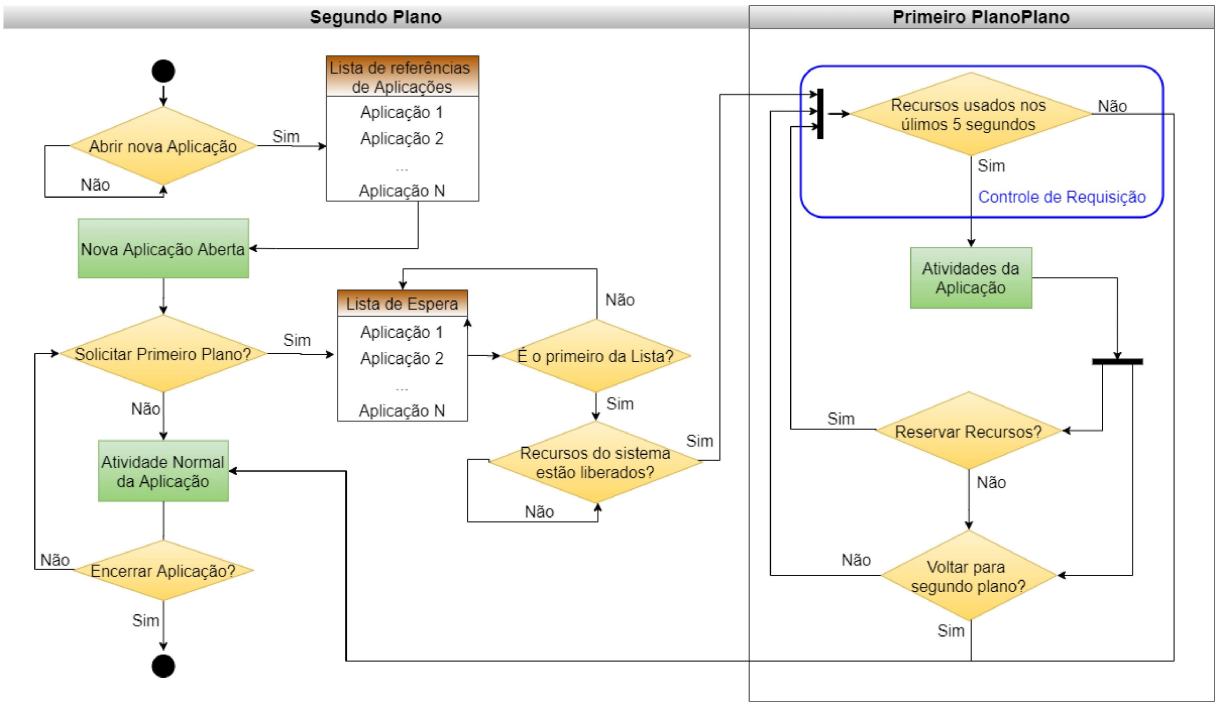



Figura 16 – Diagrama de funcionamento da *Daemon AppManager* - Fonte: Autoria Própria.

A classe *Daemon* contém um subprocesso denominado *processFocusRequest* responsável em receber, gerenciar e descartar requisições que envolvem o plano de execução, trabalhando em conjunto com o subprocesso responsável pelo gerenciamento de mensagens internas, também incluso na *Daemon*.

A *processFocusRequest* enfileira as requisições por ordem de chegada, dando o poder de execução exclusiva sempre para a primeira aplicação a entrar na fila. Aqui ocorre a aplicação do mecanismo de controle de requisição, citado anteriormente, revogando o acesso caso a aplicação não utilize este recurso nos próximos cinco segundos, e repassando para a próxima requisição na fila, se houver alguma disponível.

A *Daemon* foi implementada utilizando conceitos de *WebServices*, e trabalha em conjunto com a estrutura *SystemController*. Para acessar cada comando, basta utilizar o *endpoint* correspondente ao respectivo comando. Uma lista contendo todos os *endpoints* que esta *Daemon* possui pode ser visualizada na tabela 1 e uma lista com as funções desenvolvidas para esta *Daemon* está disponível no anexo B.

| System Controller | | | |
|---|---|---|---|
| **Endpoints** | **Função Relacionda** | **Parametros Necessarios** | **Tipo do Retorno** |
| api/System/CloseApplication | closeApplication | - | bool |
| api/System/IsCurrentApplication | isCurrentApplication | - | bool |
| api/System/LockFocus | lockFocus | - | string |
| api/System/RequestFocus | requestFocus | - | string |
| api/System/SetAsBackground | setAsBackground | - | string |
| api/System/UnlockFocus | unlockFocus | - | string |

Tabela 1 – Relação entre *Endpoints* e funções da biblioteca relacionados aos comandos do sistema - Fonte: Autoria Própria.

#### 3.2.5.3 *Daemon* de Iteração Humano-Computador

Na *Daemon* de Iteração Humano-Computador encontram-se as funcionalidades voltadas à comunicação do robô, através de linguagem natural. Ela está intrinsecamente atrelada a todo o funcionamento do *framework*, comportando-se como uma interface de interação Humano-Computador do sistema. Para a execução de tal tarefa, o sistema depende amplamente dos serviços de síntese e análise de voz implementados pelo fabricante do robô.

Assim como a linguagem natural humana, a Iteração Humano-Computador desta *Daemon* pode ser dividida em dois processos, a fala e a escuta. Baseando-se na audição humana, a operação de escuta do robô inicia-se com a análise das alterações dos sons ao redor do robô, com o intuito de identificar e reconhecer as palavras ditas, e assim entender o significado geral do que foi dito. Para isso, definiu-se que a parte inicial, de captura das palavras devem ser realizadas pelos serviços.

Como já citado anteriormente, a implementação dos serviços foi designada aos fabricantes de robôs, porém, para testes e validação da *Daemon*, foi desenvolvido um serviço de análise de voz, implementado utilizando a linguagem de programação *Python*. Este serviço monitora, através de um microfone, os sons em torno do robô e quando este sinal ultrapassa um determinado nível de volume, o serviço passa a capturá-lo e o transforma em uma amostra. A gravação da amostra é finalizada quando o som captado volta a possuir menor nível de intensidade por mais de dois segundos.

Após a captura, esta amostra de som é analisada pela biblioteca *Deepspeech*, responsável por realizar a conversão de voz em texto. A informação é enviada a *Daemon* de Iteração Humano-Computador através do tópico */voiceInput* (MOZILLA, 2017). Assim que a mensagem é capturada pela *Daemon*, esta limpa as informações existentes no tópico sobre a mensagem salva. Esta estrutura permite que os serviços sejam implementados em outras linguagens utilizando técnicas diferentes, desde que sejam compatíveis com as entradas da *Daemon*.

Quando a mensagem ouvida pelo robô, vinda do serviço de análise de voz, é recebida

pela *Daemon*, que pode inicia a análise da informação obtida. Para isso, no primeiro momento, todas as palavras da mensagem recebida são identificadas, separadas e salvas na memória. A identificação de cada palavra é realizada na ordem em que a mensagem foi recebida, possibilitando o futuro agrupamento e a possível identificação de frases.

Com a mensagem recebida salva e separada em palavras, é realizado o procedimento de análise e identificação de cada palavra. Para isso, a *Daemon* de Iteração Humano-Computador entra em contato com a *Daemon* de Registros 3.2.5.5, que gerencia tabelas de armazenamento onde encontram-se todas as palavras conhecidas pelo robô, seja comandos do sistema, comandos da aplicação, ou qualquer outro tipo de palavra já detectada pelo robô.

Os comandos são palavras definidas e salvas no sistema, que possuem alguma ação a ela atrelada, como "abrir" que irá abrir uma aplicação. Além dos comandos utilizados para controlar o sistema, desenvolvedores podem criar comandos para as próprias aplicações.

A *Daemon* de Iteração Humano-Computador verifica então todas as palavras existente na mensagem recebida, analisado e comparando com as informações contidas no banco de dados e as classifica pela similaridade encontrada a partir da sonoridade da palavra. Todas as palavras capturadas que obtiveram alguma correspondência no banco de dados são armazenadas utilizando o valor correspondente encontrado no banco de dados, em um novo local da memória.

Um dos objetivos de manter na memória um local com a palavras na forma em que foram recebidas e em outro com a maneira identificada é minimizar as possíveis inexatidões decorrentes de falsos cognados, ou palavras homófonas, como por exemplo, se em uma mensagem recebida, encontra-se a palavra "trás" indicando um local posterior, porém pelas tabelas do sistema, só foi possível identificar a palavra "traz" do verbo trazer, que possui o mesmo som, porém tem significados completamente diferentes. A *Daemon* de Iteração Humano-Computador irá armazenar a palavra "traz" no local de mensagens já identificadas, com uma referência da palavra original, e manterá a palavra "trás" registrada junto com as palavras recebidas, com uma marcação de que a mesma já foi avaliada, para o caso de a aplicação escolher utilizá-la ou aprendê-la. Para o caso de o sistema ter em sua base de dados, tanto a palavra "trás" quanto "traz", as duas serão armazenadas junto com as palavras já identificadas, e terão uma referência da palavra original.

Ter as palavras separadas e identificadas de acordo com o conhecimento dos robô, pode ajudar o desenvolvedor de aplicações para robôs à desenvolver aplicativos mais robustos, pois ele terá a opção, por exemplo, de deixar palavras pré-programadas, que ativem a aplicação, ou alguma atividade específica da aplicação, comandar para ser avisado assim que a palavra escolhida for capturada pela *Daemon*, ou mesmo, ficar verificando se o comando já foi dito.

Visando diminuir o consumo de memória do sistema, as palavras salvas, tanto as que foram agrupadas como recebidas quanto as processadas, são apagadas assim que alguma aplicação as consumir, ou após um minuto sem consumo, pois é um tempo que a equipe considerou

suficiente para o consumo de palavras para os testes nesta versão do sistema.

Além da função de escuta, a *Daemon* de Iteração Humano-Computador realiza a síntese de voz, simulando assim a fala humana. A síntese de voz do robô é acionada sempre que uma aplicação necessitar. Para isso, a aplicação define o momento em que deseja que o robô "fale", e realiza a chamada para a *Daemon* que publica a mensagem no tópico de ROS, pelo serviço específico, desenvolvido pelo fabricante de robô.

Para ser capaz de simular e validar o funcionamento desta função da *Daemon*, foi desenvolvido um serviço de síntese de voz utilizando novamente, a linguagem de programação *Python*. Este serviço tem a responsabilidade de trocar informações com a *Daemon* de Iteração Humano-Computador através do tópico ROS *voiceOutput*. O serviço captura a mensagem que contém a informação a ser sintetizada em voz deste tópico e a envia, por meio de linha de comando ao programa *Espeak* (apresentado na subseção 2.2.1).

A Figura 17 sumariza o funcionamento da *Daemon* de Iteração Humano-Computador .

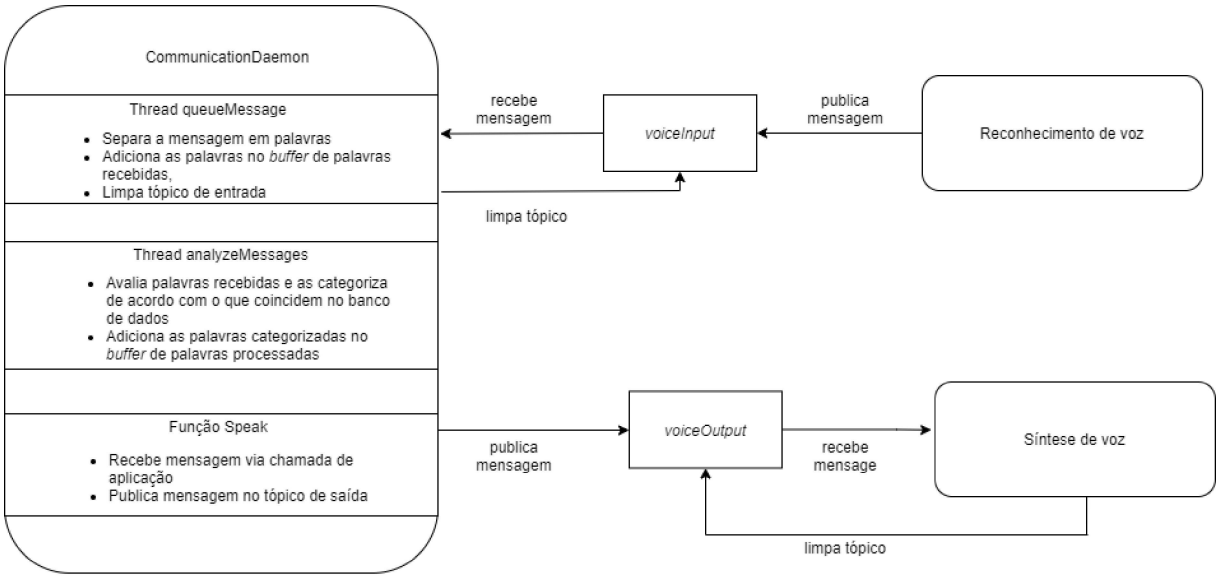



Figura 17 – Diagrama de funcionamento da *Daemon* de Iteração Humano-Computador - Fonte: Autoria Própria.

A *Ready Controller*, responsável por permitir que as aplicações acessem a *Daemon* de Iteração Humano-Computador , é denominada *CommunicationController*, estando fortemente relacionados com as responsabilidades da *Daemon* e suas funções. Os *endpoints* da *Daemon* de Iteração Humano-Computador estão listados na tabela 2 e uma lista com as funções desenvolvidas para esta *Daemon* está disponível no anexo C.

| Communication Controller | | | |
|---|---|---|---|
| **Endpoints** | **Função Relacionda** | **Parametros Necessarios** | **Tipo do Retorno** |
| api/Communication/CommandWasSaid | commandWasSaid | string command | bool |
| api/Communication/GetLatestCommands | getLatestCommands | int numberOfCommands | string [] |
| api/Communication/GetLatestWords | getLatestWords | int numberOfWords | string [] |
| api/Communication/ListenCommandsForSeconds | listenCommandsForSeconds | int seconds | string [] |
| api/Communication/ListenWordsForSeconds | listenWordsForSeconds | int timeCount | string [] |
| api/Communication/Speak | speak | string word | string |
| api/Communication/WaitForCommand | waitForCommandBoolean | string command | bool |
| api/Communication/WaitForWord | waitForWord | string word | List<VoiceWord> |
| api/Communication/WordWasSaid | wordWasSaid | string word | bool |

Tabela 2 – Relação entre *Endpoints* e funções da biblioteca relacionados a Iteração Humano-Computador - Fonte: Autoria Própria.

#### 3.2.5.4 *Daemon* de Navegação

A *Daemon* de navegação possui dois principais objetivos: o primeiro, que a nomeia, é utilização dos Serviços de SLAM para gerenciar a navegação do robô; e o segundo, consiste em controlar a locomoção do dispositivo.

Pensar em locomoção de robôs, significa considerar as mais variadas formas destes dispositivos se movimentarem, pois, mesmo dentro do grupo dos robôs terrestres, alvo do escopo deste trabalho, existem diferentes representantes capazes de se locomoverem de formas distintas, como por exemplo, os robôs omnidirecionais, capazes de se movimentar em todas as direções sem realizar manobras ou os robôs diferenciais, que para alterarem a própria direção devem realizar curvas ou giros.

Sabendo disso, houve a preocupação de se desenvolver uma *Daemon* de navegação capaz de abstrair as mudanças de tipos de movimentação, adequando-se às funcionalidades possíveis de acordo com o que o modelo é capaz de fornecer. Entretanto, para manter o escopo da proposta, a *Daemon* de navegação, na a versão atual, decidiu-se utilizar apenas a movimentação de robôs diferenciais.

A *Daemon* de navegação engloba todas as tarefas relacionadas à locomoção e navegação de robôs diferenciais, incluindo ferramentas que disponibilizam às aplicações a capacidade de controlar o robô diretamente, por meio de comandos manuais de movimentação, ou indiretamente, utilizando a navegação fornecida pelo serviço de SLAM (apresentado na subseção 3.1.3), para guiar o robô para algum lugar.

O SLAM (*Simultaneous Localization and Mapping*) é à uma técnica de navegação que permite o robô, ou outro dispositivo móvel, planejar rotas em um determinado ambiente utilizando um mapa armazenado em memória e paralelamente ser capaz de atualizá-lo utilizando informações dos sensores.

Dispondo de suporte aos serviços de SLAM, a *Daemon* é capaz de processar dados de

navegação e posição do dispositivo, fornecendo aos desenvolvedores funcionalidades como a navegação até um determinado ponto de interesse. Esse ponto, pode ser definido de diversas maneiras, passando por exemplo para *Daemon* os valores das coordenadas em relação ao mapa utilizado pelo robô do local de destino, ou a distância e ângulo total pelo qual o robô deve se movimentar, ou mesmo, informando nomes de registros já salvos na *Daemon* de Registros, denominados *Waypoints*. Os *Waypoints* que consistem em pontos de referência, que estão armazenados na base de dados do robô, definem o nome e posição de locais previamente salvos.

Além de ser capaz de movimentar-se até um determinado ponto de interesse, esta *Daemon* possui funções como a de retornar os valores das coordenadas do local onde o robô encontra-se, calcular a distância até um determinado ponto, ou encontrar o valor da distância em uma determinada direção, ou seja, ela é capaz de retornar à aplicação o valor numérico da quantidade de metros em que encontra-se o obstáculo mais próximo na direção escolhida. Para isso, utilizam-se informações coletadas a partir dos sensores de distância que o robô possui, estes valores podem ser úteis aos desenvolvedores para traçar uma rota sem atingir nenhum objeto, quando a navegação manual direta for a escolhida, uma vez que, utilizando as ferramentas de percursos do SLAM, tem-se como benefício possuir um mapa geral do ambiente próximo aonde o robô já esteve.

Assim como o acesso aos valores fornecidos pelos sensores de distância, a *Daemon* de navegação também possui acesso aos recursos críticos do robô, como o acionamento dos motores, que ativam a movimentação física do mesmo. Desde modo, ela está fortemente ligada ao gerenciamento de planos das aplicações realizado pela *Daemon AppManager* 3.2.5.2, pois há funcionalidades que só devem ser acessadas por uma única aplicação por vez, ou seja, só pelas aplicações que estejam em primeiro plano, a fim de evitar que mais de uma aplicação diferente possa acessar estes recursos críticos ao mesmo tempo e causem possíveis acidentes ou inconsistências no funcionamento do robô.

A *Ready Controller* responsável por permitir que as aplicações acessem a *Daemon* de navegação é denominada de *NavigationController* e possui os *endpoints* listados na tabela 3, que estão fortemente relacionados com as funções de responsabilidade da *Daemon*. Uma lista com as funções desenvolvidas para esta *Daemon* está disponível no anexo D.

| Navigation Controller | | | |
|---|---|---|---|
| **Endpoints** | **Função Relacionda** | **Parametros Necessarios** | **Tipo do Retorno** |
| api/Navigation/DeleteWaypoint | deleteWaypoint | string waypointName | string |
| api/Navigation/GetCurrentCoordinate | getCurrentCoordinate | - | Coordinate |
| api/Navigation/GetDistance | getDistance | double angle | double |
| api/Navigation/GetDistanceFromCoordinate | getDistanceFromCoordinate | Position position | double |
| api/Navigation/GetDistanceFromWaypoint | getDistanceFromWaypoint | string waypointName | double |
| api/Navigation/GetNearestWaypoints | getNearestWaypoints | int quantity | List<WaypointDistance> |
| api/Navigation/GetWaypoint | getWaypoint | string waypointName | Waypoint |
| api/Navigation/GoToCoordinate | goToCoordinate | Position position,<br>double angle = 0.0f | string |
| api/Navigation/GoToWaypoint | goToWaypoint | string waypointName | string |
| api/Navigation/SaveWaypoint | saveWaypoint | string waypointName | string |
| api/Navigation/WalkCurved | walkCurved | walkStraight | string |
| api/Navigation/WalkCurved | walkCurved | double linearSpeed,<br>double radius,<br>WalkCurvedModel.Orientation orientation | string |
| api/Navigation/WalkStraight | walkStraight | double speed, double angle = 0.0f,<br>double maneuverTime = 2.0f | |
| api/Navigation/WalkStraight | walkStraight | double speed,<br>double distance,<br>double angle = 0.0f,<br>double maneuverTime = 2.0f | string |

Tabela 3 – Relação entre *Endpoints* e funções da biblioteca relacionados a navegação - Fonte: Autoria Própria.

#### 3.2.5.5 *Daemon* de Registros

As *Daemons* podem ser descritas resumidamente como subprocessos que fornecem funcionalidades que serão utilizadas por aplicações ou pelo próprio sistema. Considerando isso, cada *Daemon* criada, possui como foco, alguma finalidade específica, como a navegação, a Iteração Humano-Computador ou o gerenciamento interno. Ao desenvolvê-las, percebeu-se a necessidade de padronizar e armazenar informações.

Portanto, um dos conceitos essenciais para o funcionamento do sistema foi designar um módulo capaz de gerenciar funcionalidades relacionadas ao Registros de dados. Esta é a responsabilidade designada para a *Daemon* de Registros, que, seguindo o padrão descrito, foi criada com o objetivo de dar suporte as outras *Daemons* e às aplicações, no que se refere ao Registros de informações.

A *Daemon* de Registros é responsável por coletar, tratar, adicionar e remover as informações do banco de dados *MySQL*, no qual de fato os dados são armazenados, além de fornecer a informações para as demais *Daemons* por meio do protocolo de comunicação interno 3.2.5.1.

Uma funcionalidade interessante do banco de dados *MySQL*, é o comando *Sounds Like*. Este comando retorna o valor armazenado no banco de dados com a fonética mais similar à palavra fornecida para comparação. Neste trabalho é uma funcionalidade utilizada com frequência pela *Daemon* de Iteração Humano-Computador para melhorar a detecção de comandos (W3RESOURCE, 2020).

Os dados armazenados na base *MySQL*, estão divididos em tabelas, de acordo com a

necessidade do Registros. O modelo armazenado no banco de dados é composto por seis tabelas apresentados na Figura 18.

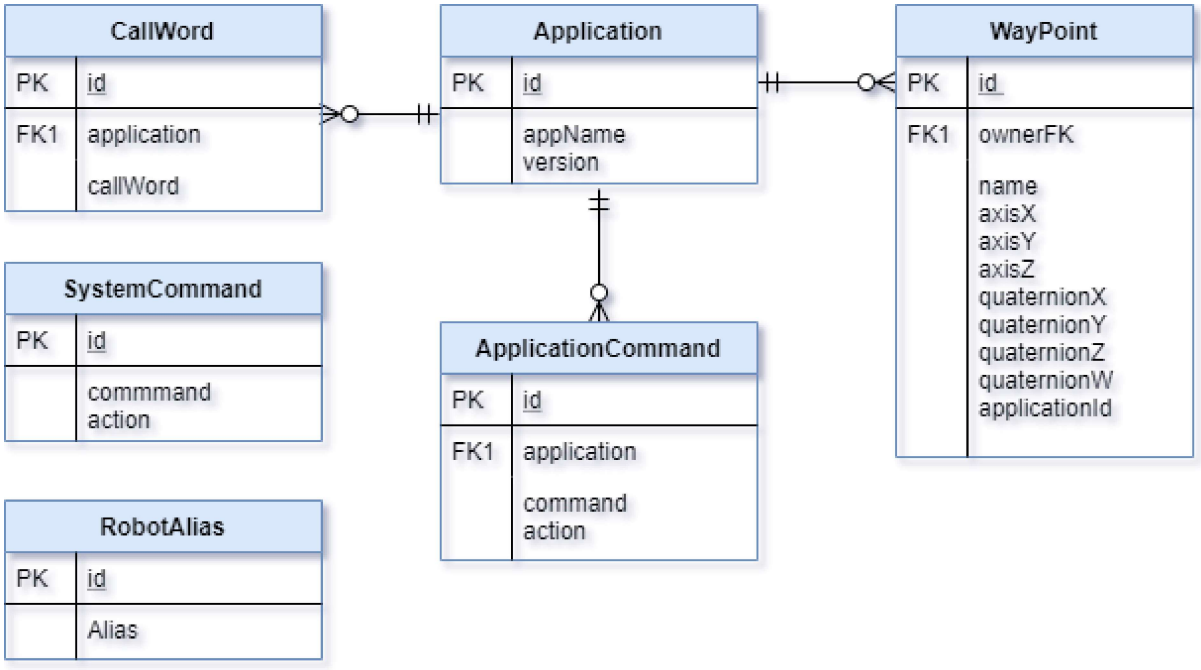



Figura 18 – Diagrama Entidade-Relacionamento - Fonte: Autoria Própria.

A Figura 18 contem uma tabela nomeada como *Application* é responsável por armazenar as informações básicas que toda a aplicação instalada no sistema proposto deve ter, que são o nome, um valor numérico único de identificação e a versão.

A identificação da aplicação é utilizada por mais três tabelas no banco de dados, chamadas de *CallWord*, *ApplicationCommand* e *Waypoint*. A primeira armazena as palavras que são utilizadas para iniciar a execução de uma determinada aplicação, definidas pelas próprias aplicações, como por exemplo, uma aplicação que joga xadrez, pode escolher definir que toda vez que o usuário diga a palavra xadrez, a aplicação seja aberta. Para isso, palavra xadrez deve ser adicionada na tabela *CallWord*, assim como o número único da identificação da aplicação, e mais um número de identificação único para a palavra escolhida, esse último é adicionado automaticamente pelo sistema. A tabela *CallWord*, com a tabela *Application* possuem um relacionamento de vários para um, ou seja, é possível ter mais de uma palavra que ative uma mesma aplicação, desde que essas não se repitam.

A segunda tabela que utiliza a informação do ID (número de identificação único) da aplicação, é a tabela *ApplicationCommand*, que contém os comandos que cada aplicação pode utilizar durante o funcionamento e a ação a que o comando se refere, que pode ou não ser o mesmo nome do comando. Para exemplificar, podemos voltar ao exemplo da aplicação capaz de jogar xadrez, esta pode definir que a palavra "*play*" seja o comando responsável por acionar o jogo, após a aplicação já estar aberta, então a palavra "*play*" será salva como comando, e a

ação será "iniciar o jogo". Os comandos da aplicação, também possuem um relacionamento de vários para um, sendo possível adicionar a quantidades de comandos desejada para uma única aplicação, tendo como única restrição, que a comando não esteja inserido na tabela *System-Command*, à ser descrita a seguir. A tabela *ApplicationCommand* tem como objetivo melhorar a precisão da detecção dos comandos utilizados pelo aplicativo em execução.

E por fim, a tabela *Waypoint* é a última a possuir um relacionamento com a tabela de aplicações. Ela é responsável por armazenar as posições utilizando os dados gerados pela navegação e as nomear de acordo com o interesse das aplicações. Por exemplo, se em uma aplicação em que o robô pode se locomover entre lugares, escolheu definir que a posição atual do robô é a cozinha, então a tabela *Waypoint* irá coletar as coordenadas atuais do robô (coordenadas x,y e z e também em formato de quatérnio), e salvar na tabela, com um número de identificação único e a referência para a aplicação que a definiu. Cada conjunto desses das informações salvas na tabela *Waypoint* é definido como um *Waypoint* e será usado pela própria navegação, assim, quando for solicitado para o robô ir até a cozinha, a aplicação pode buscar as informações nessa tabela e utiliza alguma função da *Daemon* de navegação para se movimentar até lá. Novamente, cada aplicação tem a liberdade de definir e salvar quantos *Waypoint* desejar.

Além destas, a base de dados conta com uma tabela nomeada de *SystemCommand*, que não possui relação direta com as citadas anteriormente. Ela é responsável por guardar os comandos do sistema e a ação que será desenvolvida. Esses comandos servem para referenciar qualquer aplicação e possui um nível de prioridade maior que comandos definidos na tabela *ApplicationCommand*, ou seja, uma palavra definida como comando no *SystemCommand* não poderá ser utilizada no *ApplicationCommand*. Os comandos do *SystemCommand* estão em níveis de sistema, e ajudam no gerenciamento das aplicações. Por exemplo, se for definido que a palavra "jogar" é um comando para abrir aplicações, voltando ao exemplo da aplicação de xadrez, quando o usuário falar "jogar" e, logo em seguida a *Daemon* de Iteração Humano-Computador identificar que a palavra "xadrez" foi dita, a aplicação de xadrez será aberta. O mesmo ocorre no caso da próxima palavra ser "dominó" e no sistema existir uma aplicação que joga dominó devidamente instalada, e configurada com a palavra "dominó" na tabela de *CallWord*.

A última tabela definida na base de dados para a versão atual do *Framework Ready* é denominada como *RobotAlias*. Esta armazena os diversos nomes que o usuário pode escolher nomear o robô e um número de identificação único para cada nome incluído. Assim, se o usuário preferir dar um nome mais amistoso ao seu robô doméstico, como "Fulaninho", "R2D2" ou "HAL", basta inserir esses nomes na tabela *RobotAlias* por intermédio dos comandos do sistema.

A *Daemon* de Registros tem a responsabilidade de realizar o gerenciamento de todas as tabelas e a estrutura da base de dados existentes no *Ready*, utilizando comandos internos atrelados a funções, que podem ser acessados por outras *Daemons*, como por exemplo funções

responsáveis por salvar novos *Waypoints*, ou buscar ou comando do sistema, ou mesmo, apagar um comando da aplicação.

Para citar um exemplo de caso de uso, uma aplicação poderia enviar a *Daemon* de navegação o comando "Ir para a cozinha" por meio da comunicação interna. A *Daemon* de navegação irá consultar as coordenadas do ponto de interesse "cozinha" e comandaria a navegação do robô para esta região. Deste modo, no caso dos *Waypoints*, o caminho natural é uma aplicação acessar o *endpoint* da navegação, e esta irá se comunicar com a *Daemon* de Registros, para realizar o devido procedimento na base de dados.

Esta *Daemon* define também a ordem de prioridade da busca de informações na base de dados. Para busca de comandos por exemplo, se uma aplicação está aberta, o sistema irá buscar primeiramente na tabela da comandos da aplicações, e se não conseguir encontrar nenhuma palavra correspondente com a palavra que foi dita pela usuário, captada pela *Daemon* de navegação, ele irá buscar na tabela de comandos do sistema. Essa ordem de prioridade, foi definida para impedir que o sistema se feche inesperadamente, por exemplo, ao decidir fechar apenas a aplicação corrente.

Uma lista com as funções desenvolvidas para esta *Daemon* está disponível no anexo E.

## 3.3 Bibliotecas *Ready*

As bibliotecas fornecidas pelo *Framework Ready* tem como objetivo permitir que os recursos disponibilizados pelo *Middleware Ready* possam ser usados pelas aplicações, utilizando a tecnologia de *WebAPI*, de modo que os desenvolvedores destas não precisem conhecer os detalhes da biblioteca *Ready* nem as estruturas do *Middleware* que recebem estas requisições, focando apenas no desenvolvimento da aplicação com as funções disponibilizadas pela biblioteca. Por mais que diferentes bibliotecas possam ser implementadas, nada impede que o desenvolvedor acesse os *endpoints* do *middleware* diretamente na atual versão do *framework*, deixando isso à critério do mesmo, permitindo a criação de bibliotecas próprias. Entretanto, recomenda-se utilizar as bibliotecas padrão para garantir o funcionamento adequado do *framework*.

Escolher a tecnologia de *WebAPI* como meio de comunicação entre o *Middleware* e as aplicações têm como vantagem implementar bibliotecas em diferentes linguagens de programação, desde que estas sejam compatíveis com esta tecnologia. Nesta versão do *framework Ready* foi desenvolvida apenas uma biblioteca, focada em aplicações implementadas na linguagem C♯, responsável em intermediar a comunicação com o *Middlware* e entregar aos programadores funções capazes de acessar os *endpoints* disponibilizados, além de possuir os modelos de objetos utilizados pelo *framework*. Durante o desenvolvimento desta biblioteca, foi definido que as funções que a compõe seriam de execução síncrona, ou seja, a execução da aplicação que realiza uma chamada à *WebAPI* só irá prosseguir após receber uma resposta da mesma, podendo

ser tanto uma mensagem de sucesso quanto uma falha.

Utilizar como base funções síncronas possuí a vantagem de manter as funções mais estruturadas, deixando mais simples para desenvolvedores com pouca experiencia em programação. Além disso, o sistema permite que novas funções assíncronas possam ser implementadas, e disponibilizar funções síncronas em outras versões do sistema.

A tabela 4, apresenta a relação entre todos os *endpoints* existentes na versão atual do *Middleware Ready* e as funções da biblioteca relacionada, assim como o parâmetros de entrada e o tipo de retorno da função. Uma explicação mais detalhada da finalidade de cada funções, pode ser encontradas nos anexos B, C e E.

| Endpoints e Funções | | | |
|---|---|---|---|
| **Endpoints** | **Função Relacionda** | **Parametros Necessarios** | **Tipo do Retorno** |
| **System Controller** | | | |
| api/System/CloseApplication | closeApplication | - | bool |
| api/System/IsCurrentApplication | isCurrentApplication | - | bool |
| api/System/LockFocus | lockFocus | - | string |
| api/System/RequestFocus | requestFocus | - | string |
| api/System/SetAsBackground | setAsBackground | - | string |
| api/System/UnlockFocus | unlockFocus | - | string |
| **Communication Controller** | | | |
| api/Communication/CommandWasSaid | commandWasSaid | string command | bool |
| api/Communication/GetLatestCommands | getLatestCommands | int numberOfCommands | string [] |
| api/Communication/GetLatestWords | getLatestWords | int numberOfWords | string [] |
| api/Communication/ListenCommandsForSeconds | listenCommandsForSeconds | int seconds | string [] |
| api/Communication/ListenWordsForSeconds | listenWordsForSeconds | int timeCount | string [] |
| api/Communication/Speak | speak | string word | string |
| api/Communication/WaitForCommand | waitForCommandBoolean | string command | bool |
| api/Communication/WaitForWord | waitForWord | string word | List<VoiceWord> |
| api/Communication/WordWasSaid | wordWasSaid | string word | bool |
| **Navigation Controller** | | | |
| api/Navigation/DeleteWaypoint | deleteWaypoint | string waypointName | string |
| api/Navigation/GetCurrentCoordinate | getCurrentCoordinate | - | Coordinate |
| api/Navigation/GetDistance | getDistance | double angle | double |
| api/Navigation/GetDistanceFromCoordinate | getDistanceFromCoordinate | Position position | double |
| api/Navigation/GetDistanceFromWaypoint | getDistanceFromWaypoint | string waypointName | double |
| api/Navigation/GetNearestWaypoints | getNearestWaypoints | int quantity | List<WaypointDistance> |
| api/Navigation/GetWaypoint | getWaypoint | string waypointName | Waypoint |
| api/Navigation/GoToCoordinate | goToCoordinate | Position position,<br>double angle = 0.0f | string |
| api/Navigation/GoToWaypoint | goToWaypoint | string waypointName | string |
| api/Navigation/SaveWaypoint | saveWaypoint | string waypointName | string |
| api/Navigation/WalkCurved | walkCurved | walkStraight | string |
| api/Navigation/WalkCurved | walkCurved | double linearSpeed,<br>double radius,<br>WalkCurvedModel.Orientation orientation | string |
| api/Navigation/WalkStraight | walkStraight | double speed,<br>double angle = 0.0f,<br>double maneuverTime = 2.0f | |
| api/Navigation/WalkStraight | walkStraight | double speed,<br>double distance,<br>double angle = 0.0f,<br>double maneuverTime = 2.0f | |

Tabela 4 – Relação entre *Endpoints* e funções da biblioteca - Fonte: Autoria Própria.

Mais detalhes de como esta comunicação ocorre são descritos na seção 3.4.

## 3.4 Aplicações

Aplicações são definidas como os programas que, utilizando as bibliotecas disponibilizadas do *Ready*, sendo capazes de se comunicar e atuar no robô através do *middleware*. Permitir a existência e a execução das aplicações é um dos objetivos do *framework Ready*, pois são elas que irão ditar o foco e o comportamento do robô, sendo o *framework* apenas o meio que possibilita essa variedade de funcionalidades.

Todas as aplicações, antes de serem executadas, devem ser previamente registradas no sistema, salvando informações pertinentes a ela no banco de dados, como Nome, Versão, Comandos e tipo de execução. Assim que uma aplicação é registrada, ela recebe um Id único que será utilizado como sua identificação dentro do sistema.

Cada aplicação em execução no sistema funciona de forma independente e paralela às demais, ou seja, o escopo de cada aplicação é isolado e não há nenhuma interação entre elas. Como já explicado anteriormente, somente aplicações que estão em Primeiro Plano podem acessar os recursos protegidos do sistema, garantindo assim que apenas uma aplicação por vez tome controle dos componentes físicos dos robôs, fortalecendo o conceito do isolamento de escopo de cada aplicação.

# 4 Experimentos

## 4.1 Aplicações

Para validar o funcionamento do *Framework Ready*, foram implementadas três aplicações capazes de utilizar as funções fornecidas pelo *framework*. As aplicações foram criadas utilizando a biblioteca *Ready* disponibilizada para desenvolvimento em linguagem C♯, como explicado na seção 3.3.

A primeira aplicação a ser apresentada recebeu o nome de *WatchBot* 4.1.1, pois ela realiza uma patrulha de um ponto escolhido, utilizando recursos da navegação e fala fornecidos pelo *Ready*. A segunda aplicação tem o objetivo de simular a função de uma agenda, capaz de armazenar e gerenciar e-mail e telefones, utilizando recursos da comunicação, como fala e reconhecimento de voz. Esta recebeu o nome de *Contacts* e é descrita com mais detalhes na subseção 4.1.2. A última aplicação implementada utiliza recursos da comunicação e movimentação e foi denominada *Waiter*. Recebeu este nome pois foi desenvolvida para atuar como um garçom de festas, conforme descrito na subseção 4.1.3.

Para um desenvolvedor de aplicações para robôs implementar aplicações a partir do Framework Ready, como as citadas nesta seção, que utilizem recursos de comunicação e movimentação, ou outros, basta utilizar as funções fornecidas pela biblioteca *Ready*. Caso o desenvolvedor escolha utilizar uma linguagem de programação para a qual não foi fornecida uma biblioteca *Ready*, mas que seja compatível com os recursos de *WebServices*, existe a opção de desenvolver uma biblioteca própria, utilizando os *endpoints* fornecidos pelo *middleware Ready*, ou utilizar diretamente os *endpoints*. Apesar desta ser uma solução viável, aconselha-se a sempre que possível utilizar as bibliotecas fornecidas pelo *Ready*, pois as mesmas já foram testadas, e possuem funções que auxiliam no desenvolvimento de aplicações.

Caso a aplicação à ser desenvolvida necessitar fazer uso de recursos críticos, restritos à aplicação corrente em primeiro plano, basta acionar a *Daemon AppManager*, realizar a demanda e aguardar a confirmação.

Se uma aplicação com o mesmo intuito fosse desenvolvida diretamente sob o ROS, seria necessário que o desenvolvedor conhecesse bem os componentes do robô e os pacotes fornecidos pelo ROS para a demanda requerida, além de ter que construir meios de gerenciar diretamente os recursos críticos do robô.

Deste modo, entre as vantagens de utilizar o *Framework Ready* para o desenvolvimento de aplicações robóticas, pode-se citar a redução na quantidade de informações que o desenvolvedor precisa conhecer, além da diminuição das ferramentas auxiliares a serem planejadas e implementadas, como controle de recursos, ou gerenciamento de dados. Essas abstrações que o

*Ready* é capaz de encapsular, permitem que o desenvolvedor foque apenas nas necessidades de construção da própria aplicação.

### 4.1.1 *WatchBot*

Ao ser iniciada, a aplicação aguardará que o usuário determine um *Waypoint* a ser vigiado, ou seja, que o usuário escolha um ponto de interesse que está previamente salvo na memória do dispositivo. Além do *Waypoint*, a aplicação espera o recebimento da distância que o dispositivo deve manter do ponto e uma orientação para a trajetória (horário ou anti-horário). Mais detalhes sobre o que é o *waypoint*, podem ser encontrados nas seções 3.2.5.5 e 3.2.5.4, consiste em uma coordenada de uma área de interesse já salva na memória do dispositivo.

Com estas informações, a aplicação calcula em torno do *Waypoint* selecionado oito diferentes coordenadas equidistantes e compreendidas em uma circunferência com o raio sendo a distância escolhida pelo usuário, conforme exemplificado na Figura 19. Foram definidos apenas 8 pontos por se mostrarem suficientes para o robô descrever uma órbita circular.

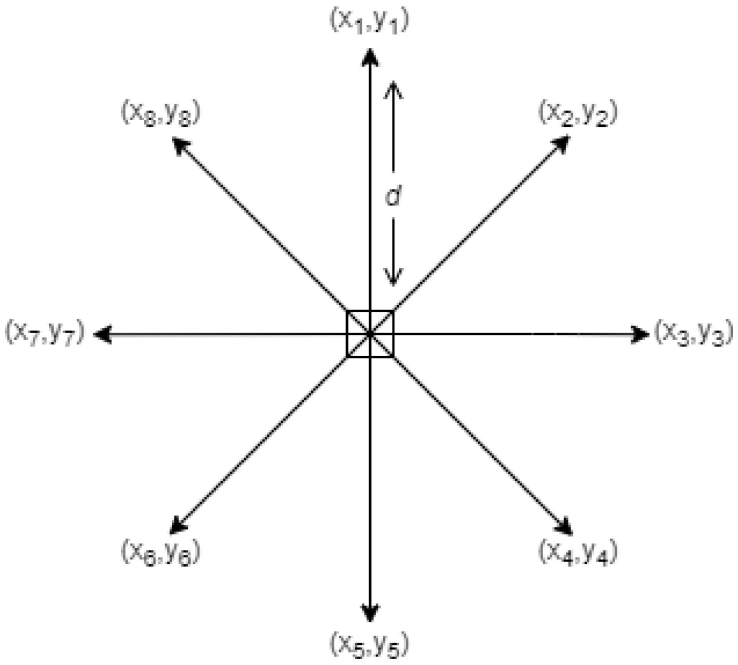



Figura 19 – Cálculo das órbitas - Fonte: Autoria Própria.

Utilizando os recursos de navegação e comunicação fornecidos pelo *middleware*, o robô se desloca até o primeiro dos pontos calculados e inicia uma ronda, navegando pelos demais pontos, de acordo com o sentido escolhido pelo usuário. Toda vez que o robô atinge um destes pontos, ele informa o usuário através da fala, usando a frase: "*going to the next position*" antes de navegar para o próximo.

Este comportamento se repete até o dispositivo ser ordenado a parar ou receber um novo *waypoint* para ser vigiado.

Os testes foram realizados no ambiente de simulação *Gazebo* onde arquivos necessários, como o modelo do robô e o ambiente foram fornecidos pela comunidade ROS (ROBOTIS, 2020a). A Figura 20 demonstra o robô neste ambiente virtual, em sua posição inicial de testes da aplicação.

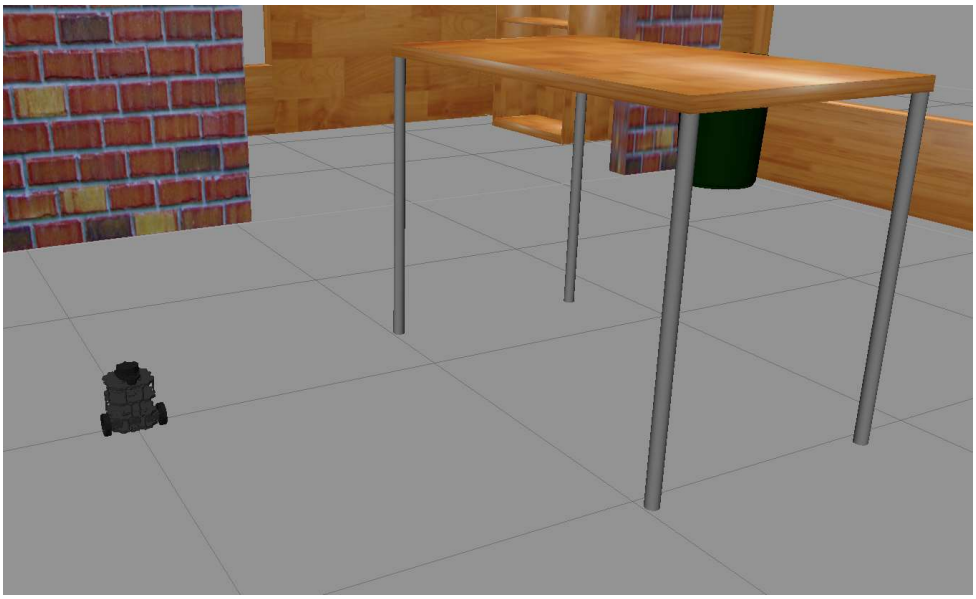
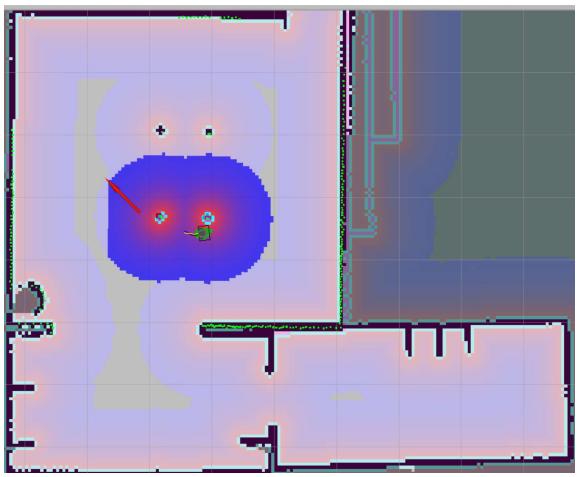
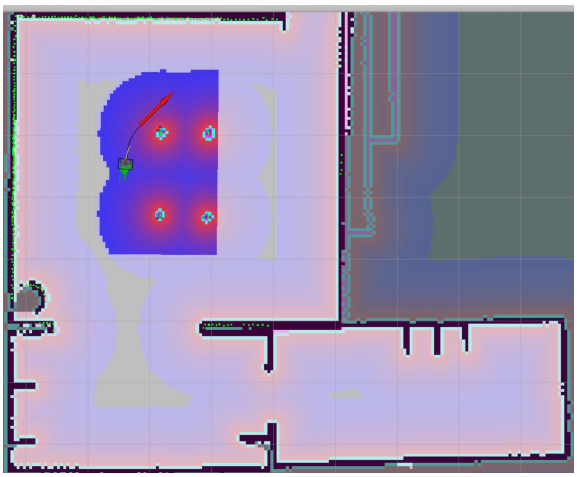


Figura 20 – Posição inicial do robô no ambiente virtual - Fonte: Autoria Própria.

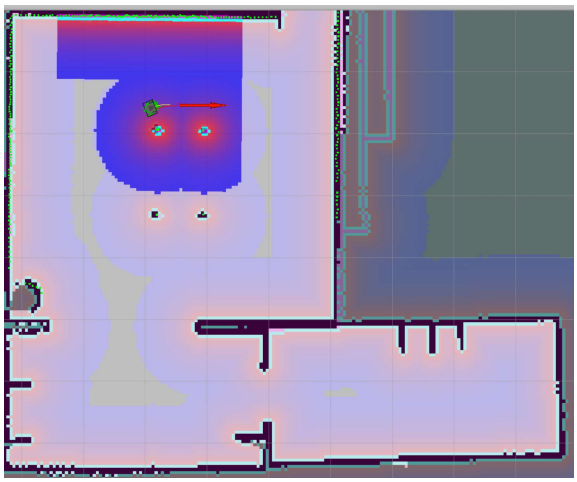
A Figura 21 contém imagens geradas durante um teste no qual o robô vigia um *waypoint* denominado "*table*", ilustrando a movimentação do robô em torno deste ponto bem como a distinção de obstáculos e rota projetada pelo robô.



(a) Início do ciclo.



(b) Sequência do ciclo.



(c) Continuação da sequência.

Figura 21 – Ciclo de funcionamento do *WatchBot* em torno de um ponto. Fonte: Autoria Própria.

Os primeiros ensaios realizados consistiram em testar a aplicação em uma área onde a movimentação do robô era livre e com poucos obstáculos. O *waypoint* escolhido se localizava no centro da mesa que compõe o ambiente e era denominado "*table*". Foi estipulada uma distância que colocaria a trajetória do robô interceptando os pés da mesa, para verificar o comportamento do dispositivo ao encontrar obstáculos durante a execução desta tarefa.

Neste ensaio o robô foi capaz de realizar a ronda e adequar a trajetória de acordo com os obstáculos que interferiam a movimentação. Usar a navegação para realizar esta tarefa teve a vantagem de tornar o robô capaz de adequar a trajetória frente a obstáculos.

Também foram realizados testes em regiões onde a quantidade de obstáculos era maior, visando avaliar o comportamento do serviço de navegação. O robô se mostrou competente em desviar dos obstáculos e chegar até o ponto de destino, porém, problemas foram encontrados na aplicação quando *waypoints* próximos a paredes e estruturas maiores foram utilizados, pois o cálculo da órbita não considerava estes objetos.

Mesmo com este resultado, sabe-se que o *framework* possui ferramentas suficientes para

contornar este problema se mais iterações de desenvolvimento forem feitas na aplicação.

No anexo J encontra-se o código da órbita que tem como objetivo armazenar informações pertinentes ao posicionamento do robô, além de calcular a rota da órbita em que o robô irá fazer a ronda, e o anexo I apresenta o código da aplicação *Watchbot*. Todas as linhas de código que provem do *Ready* são apresentadas em destaque no código.

### 4.1.2 *Contacts*

Atualmente existem inúmeros dispositivos voltados a serem assistentes pessoais, embora não sejam robôs propriamente ditos, o funcionamento deste tipo de dispositivo pode ser incluído em um robô doméstico.

Desta maneira, a equipe imaginou um aplicativo simples que atua como uma lista de contatos para mostrar que o sistema proposto pode ser utilizado para acrescentar funções de assistentes pessoais a um robô doméstico, utilizando a *Daemon* de Iteração Humano-Computador para escutar o desejo do usuário e transmitir os devidos resultados.

A aplicação *Contacts* simula uma agenda telefônica, pela qual o cliente pode solicitar ao robô adicionar, remover ou buscar um contato. Para isso, a aplicação é constituída por uma tabela na base de dados, onde fica salvo o nome do contato, telefone e *e-mail*.

Esta tabela foi nomeada como *Contacts* e foi desenvolvida em *MySql*, por utilizar comandos como o *Sounds Like*, que busca a palavra pela sonoridade da mesma. O código completo onde encontra-se a manipulação da base de dados da aplicação, pode ser visualizado no anexo H. Todas as linhas de código que provem do *Ready* são apresentadas em destaque no código.

A comunicação entre o usuário e o robô é feita pela fala. Uma vez que a aplicação está aberta, o robô utiliza a síntese de voz gerenciada pela *Daemon* de Iteração Humano-Computador para realizar a seguinte pergunta ao usuário: “O que você deseja realizar? Adicionar, Remover ou pesquisar alguém na agenda.” Para o caso da resposta recebida conter a palavra “remover” a aplicação realizará mais uma pergunta: “Quem você deseja remover?”, e aguardará pela resposta.

Após receber da *Daemon* de Iteração Humano-Computador a lista com as últimas palavras ditas pelo usuário, a aplicação comparará as informações desta com os nomes armazenados no próprio banco de dados. Se a lista não contiver dentre as palavras, um nome já salvo, o sistema retornará a mensagem “Nenhum contato foi encontrado com este nome, por favor, tente novamente.’”, mas se houver o sistema retornará a mensagem “Você tem certeza de que quer remover Fulano de sua agenda?” e aguardará a confirmação do usuário. Se usuário confirmar a remoção e a operação ocorrer om sucesso o sistema confirmará o sucesso, e em caso de falha o sistema retornará uma mensagem de erro e solicitará uma nova tentativa.

Basicamente a operação deste aplicativo pode ser resumida em o programa realizar a pergunta e fornecer as possíveis respostas ao usuário, por meio da síntese de voz, e aguardar

um determinado tempo para que o usuário responda a operação desejada. Uma vez detectada a resposta, a aplicação realiza a ação correspondente à operação desejada pelo usuário.

Após a tarefa ser realizada, a aplicação pergunta se o usuário deseja realizar outra operação, em caso afirmativo, a aplicação realiza novamente as perguntas iniciais, caso o contrário a aplicação irá se encerrar, finalizando assim a execução.

Ressalta-se que esta comunicação é realizada utilizando o idioma inglês por conta da ferramenta "*Mozilla DeepSpeech*" 2.2.3, utilizada no serviço de reconhecimento de voz, ser capaz de reconhecer apenas este idioma.

Para exercer as funcionalidades, a aplicação utiliza várias funções fornecidas pela *Ready*, como o "*speak*", "*listenWordsForSeconds*", "*closeApplication*". No anexo G encontra-se o código completo desenvolvido para esta aplicação e no anexo F um diagrama que apresenta a sequência de passos e processos que ocorrem durante o funcionamento da aplicação. Todas as linhas de código que provem do *Ready* são apresentadas em destaque no código.

Para a realização dos testes desta aplicação, foram utilizadas entradas definidas, ou seja, as entradas foram fornecidas na forma de texto com o objetivo de avaliar funcionamento e o comportamento exclusivo da aplicação, verificando e corrigindo possíveis falhas, se encontradas e servindo de auxílio para a validação da versão final da aplicação.

Durante a realização dos testes, percebeu-se que a aplicação não está preparada para aceitar nomes compostos, pois as palavras escutadas são separadas em listas e lidas como nomes únicos. Considerando que o objetivo desta aplicação é apenas comprovar que o *framework* pode funcionar como previsto, decidiu-se que o modo como encontra-se atualmente já é suficiente e em todos os testes realizados, as funções demonstraram ser eficientes, desempenhando o papel a que foi proposto a desempenhar.

### 4.1.3 *Waiter*

Alguns robôs disponíveis no mercado para o uso doméstico não são projetados para o entretenimento, mas para a execução de tarefas caseiras, como por exemplo, limpeza de chão. Neste contexto, foi imaginada uma aplicação que torna o robô capaz de auxiliar servindo os convidados em uma festa, por exemplo.

Para a execução desta tarefa, o robô utiliza a comunicação para escutar e a navegação para vagar por uma determinada área. A navegação tem como objetivo permitir o robô desviar de obstáculos, incluindo humanos que podem interceptar a trajetória.

Por fazer uso de recursos críticos do sistema, como os comandos de locomoção, a aplicação precisa inicialmente solicitar à *Daemon AppManager* para tornar-se a aplicações de primeiro plano. O código 4.1.3 contém a implementação desta aplicação. As linhas 21 a 25 descrevem como a aplicação requisita o primeiro plano para a *Daemon AppManager*, utilizando a função *sysHandler.requestFocus()*, e a linha 27 descreve como o foco do primeiro plano é

travado para esta aplicação. Todas as linhas de código que provem do *Ready* são apresentadas em destaque no código.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6  using Ready;
7  using Ready.Models;
8  using System.Threading;
9
10 namespace ReadyAppWaiterDotNetCore{
11     class Program{
12         static void Main(string[] args){
13             int id = 3;
14             var connection = new ReadyConnection(id, "http://192.168.100.142:5123/");
15             var commHandler = new ReadyCommunication(connection);
16             var navHandler = new ReadyNavigation(connection);
17             var sysHandler = new ReadySystem(connection);
18             var currentCoordinate = new Coordinate();
19             var targetCoordinate = new Coordinate();
20             var responseFocus = sysHandler.requestFocus();
21             Console.WriteLine(responseFocus);
22
23             while (!sysHandler.isCurrentApplication()) {
24                 Console.WriteLine("Nao sou a aplicacao em Foco");
25             };
26             sysHandler.lockFocus();
27             Console.WriteLine("Sou a aplicacao em Foco");
28             Random rnd = new Random();
29             bool called = false;
30             bool go = false;
31             double distance = 0f;
32             float angle = 0f;
33             currentCoordinate = navHandler.getCurrentCoordinate();
34             targetCoordinate = currentCoordinate;
35
36             while(true){
37                 currentCoordinate = navHandler.getCurrentCoordinate();
38                 if(GetDistance(currentCoordinate,targetCoordinate) < 0.3f){
39                     angle = rnd.Next(360);
40                     distance = navHandler.getDistance(angle);
41                     targetCoordinate.Position.x = Math.Cos(AngleToRadians(angle)) *
                       ↪  (distance -distance*0.2f) + currentCoordinate.Position.x;
42                     targetCoordinate.Position.y = Math.Sin(AngleToRadians(angle)) *
                       ↪  (distance -distance*0.2f) + currentCoordinate.Position.y;
```

```
43                      targetCoordinate.Position.z = currentCoordinate.Position.z;
44                      targetCoordinate.Orientation = angle;
45                      navHandler.goToCoordinate (targetCoordinate.Position,
                        ↪  targetCoordinate.Orientation );
46                  }
47                  if(go == true) go = false;
48                  called = commHandler.wordWasSaid ("waiter");
49                  if(called == true){
50                      navHandler.goToCoordinate (currentCoordinate.Position,
                        ↪  currentCoordinate.Orientation );
51                      DateTime time = DateTime.Now.AddSeconds(10);
52                      while(time>DateTime.Now && go!=true){
53                          go = commHandler.wordWasSaid ("go");
54                      }
55                      called=false;
56                  }
57              }
58          }
59          private static double GetDistance(Coordinate current,Coordinate target){
60              return Math.Sqrt(Math.Pow((target.Position.x - current.Position.x ), 2) +
                ↪  Math.Pow((target.Position.y - current.Position.y ), 2));
61          }
62           private static double AngleToRadians(double angle){
63              return (Math.PI / 180) * angle;
64          }
65      }
66  }
```

O objetivo desta aplicação é simular os serviços de um garçom, andando por uma região "carregando petiscos" de maneira aleatória até ser solicitado por alguém. Quando chamado, o robô permanece imóvel até completar 10 segundos ou, antes disso, se for dispensado, (linhas 50 à 58 do código). Decidiu-se por este tempo de espera por considerar ser um tempo suficiente para uma pessoa escolher o petisco e ir embora, antes do robô voltar as atividades, caso o usuário esqueça de dispensá-lo.

Enquanto espera uma solicitação, o robô permanecerá vagando pelo ambiente, tendo a movimentação definida pelo procedimento descrito a seguir:

- No instante inicial, é sorteado um valor de ângulo do giro a ser realizado pelo robô, e através dos sensores de distância do dispositivo, utilizando as funções fornecidas pela *Daemon* de navegação, encontra-se a distância livre até o primeiro obstáculo na direção definida pelo ângulo.
- Em posse deste valor, são utilizados cálculos trigonométricos para encontrar as coordenadas da próxima posição que deve ser alcançada pelo dispositivo. Com o objetivo de evitar

colisões, o cálculo da nova coordenada é realizado utilizando 80% do valor da distância medido, impedindo que o robô atinja o obstáculo detectado, pois estimou-se que 20% é a menos em uma distância é o suficiente para evitar colisões.

Utilizando o ângulo sorteado e a nova coordenada calculada, a aplicação entra novamente em contato com a *Daemon* de navegação, para realizar o deslocamento do robô utilizando a função "*goToCoordinate*". Ao atingir a nova posição, o procedimento é realizado novamente.

Em paralelo, o robô estará sempre esperando ser solicitado através da palavra *"waiter"* utilizando as funcionalidades da *Daemon* de Iteração Humano-Computador . Uma vez que esta ordem seja detectada, o robô suspende a movimentação e permanece imóvel por 10 segundos, ou até a *Daemon* de Iteração Humano-Computador retornar que a palavra *go*, foi detectada, retornando ao movimento.

O fluxograma representado pela Figura 22 resume o funcionamento da aplicação.

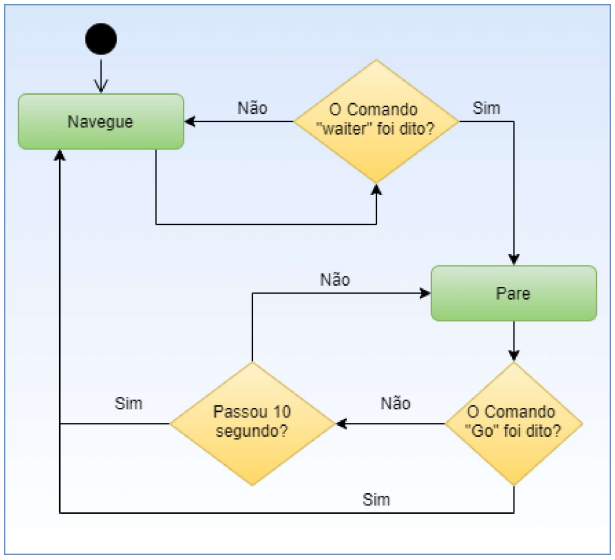



Figura 22 – Fluxograma de atividade da aplicação *Waiter* - Fonte: Autoria Própria.

O *Framework Ready* evitou que durante o desenvolvimento da aplicação fosse necessário desenvolver meios de prevenir, por exemplo, o conflito de duas aplicações acessando a navegação, uma vez que a *Daemon AppManager*, auxilia nesta tarefa, concedendo permissão de locomoção do robô apenas à aplicação em execução em primeiro plano, ou seja, apenas uma aplicação por vez. Além desta funcionalidade, não foi necessário desenvolver uma estrutura de código para tratar a função de "escuta" do dispositivo, pois tanto a captação, como a detecção e confirmação foram realizadas pelo *middleware*.

Caso esta mesma aplicação fosse desenvolvida utilizando apenas as funcionalidades fornecidas pelo ROS, seria necessário que o programador conhecesse os pacotes de ROS para

a navegação deste robô em específico, além de conhecer um pacote de ROS que fornece a capacidade de detecção de voz, ou então realizar a implementação de um pacote próprio.

Se a aplicação fosse desenvolvida apenas utilizando os recursos do ROS, o desenvolvedor deveria conhecer a arquitetura do sistema, ou seja, a maneira como os pacotes escolhidos para compor o sistema se relacionariam entre si. Esta compreensão é necessária pois como o ROS apenas intermedia a troca de informações, mais aplicações tentando consumir um determinado recurso pode causar um mau funcionamento no sistema, fato que pode gerar conflito em sistemas críticos, como no caso desta aplicação, a navegação e a própria "escuta".

Além da visão sistêmica, o desenvolvedor também deveria implementar algumas ferramentas, como por exemplo meios de calcular distância entre dois pontos, e de entender como os tópicos gerados pelos pacotes escolhidos organizam as informações sobre a fala captada e sobre o dispositivo, tal como a posição atual e o controle de locomoção do robô.

Após a implementação desta aplicação e instalação no *framework Ready*, vários testes foram realizados, para verificar o real funcionamento dos mesmo, deixando o robô navegando pela ambiente, para descobrir se iria colidir em algo, depois chamando-o para analisar sua pausa, analisando se após 10 segundos ele voltaria a navegar caso não fosse dispensado, ou verificando a volta à navegação instantânea quando o mesmo é dispensando, e em todos os casos, o funcionamento ocorreu como previsto.

### 4.1.4 Resultados gerais dos testes

Os testes individuais foram usados para auxiliar no desenvolvimento, revelando possíveis falhas e servindo para a validação da versão final da aplicação. Uma vez que os testes realizados individualmente nas aplicações atingiram os objetivos previstos, foram organizados novos testes envolvendo a execução de mais de uma aplicação ao mesmo tempo.

Os testes simultâneos envolveram as aplicações *WatchBot* e *Contacts*, e ambas conseguiram desempenhar as próprias funções corretamente, obtendo resultados semelhantes aos descritos nas seções anteriores.

Uma das características do sistema submetida a este tipo de teste foram as funcionalidades referentes a síntese de voz, as quais foram utilizadas por ambas aplicações ao mesmo tempo neste ensaio. Em algumas ocasiões, a requisição do serviço de voz foi acessada no mesmo instante pelas duas aplicações, entretanto, como o escopo de cada uma é isolado, o serviço processava os pedidos individualmente e por ordem de chegada, ou seja, todas as frases geradas pelas duas aplicações foram expressadas pelo robô e não houve nenhum problema de execução nas aplicações.

Outro procedimento realizado neste teste foi o desligamento do serviço de síntese de voz, visando avaliar o comportamento geral tanto do sistema quanto das aplicações em situações de falha ou indisponibilidades de algum serviço. Como esperado, ambas as aplicações

continuaram a execução sem nenhum erro ou comportamento imprevisto, somente com a funcionalidade de fala prejudicada. Além das aplicações, o *middleware* também foi capaz de continuar o funcionamento, mantendo a capacidade de operar e lidar com as requisições realizadas pelas aplicações, mesmo sem a disponibilidade do serviço de síntese de voz.

Um teste envolvendo as aplicações *WatchBot* e *Waiter* também foi cogitado, entretanto analisando o comportamento de ambas as aplicações e de como o sistema de requisição de travamento do foco de Primeiro Plano foi projetado, percebeu-se a necessidade de uma melhoria no modo de como o *middleware* lida com aplicações que travam o foco por muito tempo e não o liberam, assim raptando os recursos restritos apenas para ela.

De maneira geral, os testes realizados envolvendo mais de uma aplicação em execução ao mesmo tempo tiveram como o objetivo verificar o comportamento destas frente as políticas de gerenciamento de recursos do *middleware* e possíveis falhas em relação ao acesso simultâneo dos mesmos recursos. Desta maneira, durante os testes foi possível melhorar e validar as estruturas do *middleware* envolvidas nestes processos e o funcionamento do sistema como um todo.

# 5 Conclusão

O *Ready*, é o nome dado ao *Framework* que objetiva fornecer recursos que auxiliam desenvolvedores a construir aplicações para robôs domésticos, permitindo a expansão e o gerenciamento das funcionalidades existente no dispositivo e capaz de atender diversas linguagens de programação.

Para facilitar a construção e a validação do trabalho, o *Ready* foi dividido em partes menores: o *middleware Ready*, responsável por gerenciar a comunicação entre os dispositivos do robô e o próprio *framework*; e as bibliotecas que abstraem as funções fornecidas pelo *Ready* para as aplicações. O sistema também é composto por aplicações e serviços, que embora tenham sido planejados para serem implementados por terceiros, ou seja, desenvolvedores alheios ao *framework*, foram implementados para demonstrar o conceito.

Durante o desenvolvimento do trabalho, cada etapa gerou obstáculos que revelaram inúmeros desafios. No planejamento houve a difícil e impactante decisão de escolher a versão do ROS, baseando-se na disponibilidade de pacotes existentes, porém, a versão escolhida, embora capaz de atender outros requisitos, não possui pacotes voltados ao reconhecimento de voz, ampliando a dificuldade e escopo do trabalho, pela necessidade de implementação de um pacote próprio para esta função.

A configuração do ambiente de desenvolvimento também causou contratempos, tais quais, o corrompimento da instalação do ROS que ocorreu em diferentes situações, causados por atualizações de pacotes, ou pela instalação de programas com problemas de compatibilidade e até mesmo por falhas de sistema. Como medidas para contornar estes problemas a equipe desenvolveu *scripts* de instalação, acelerando a configuração do sistema e passou a utilizar máquinas virtuais, que permitiram a manutenção de cópias do sistema inteiro, tornando possível a recuperação do sistema com maior velocidade.

Um dos passos de desenvolvimentos mais importantes, foi definir em que linguagem de programação o *middleware Ready* seria desenvolvida. Em um primeiro momento, cogitou-se utilizar a linguagem Java para implementação de uma *webAPI*, pois além de possuir este recurso, ela também é compatível com o ROS.

Entretanto, após testes iniciais, a linguagem Java não apresentou estabilidade desejada para realizar uma implementação eficiente e de acordo com os planos da equipe. Após algumas pesquisas, a biblioteca *Ros-Sharp* para a linguagem C♯ foi encontrada, e depois de realizar estudos mais aprofundados, verificou-se que esta biblioteca se comportava da maneira desejada em termos de integração de *software*.

Algumas pequenas adaptações tiveram que ser realizadas para a biblioteca funcionar de acordo com o esperado, mas a partir do momento que foram finalizadas, a real implementação

do *middleware* começou a tomar forma, motivando ainda mais a equipe a seguir com o trabalho utilizando o C♯.

Desenvolver uma arquitetura genérica o suficiente, capaz de aceitar cada vez mais diferentes funcionalidades forçou a equipe a entender vários conceitos de engenharia de *software* e até mesmo alguns utilizados por sistemas operacionais, como os conceitos que inspiraram a criação das *Daemons* e dos serviços.

O trabalho em equipe por si só possui dificuldades inerentes e exige boa organização, planejamento e comunicação, que intrinsecamente são tarefas desafiadoras. Entretanto, a pandemia de Coronavírus adicionou outro obstáculo que forçou uma readaptação na forma de como o trabalho de cada integrante da equipe era realizado e de como a comunicação entre os envolvidos aconteceu.

Durante o desenvolvimento, a equipe aprendeu e se surpreendeu com o funcionamento das *Daemons* e dos serviços, que originalmente foram desenvolvidos para encapsular funcionalidades do robô, mas também são capazes de encapsular o funcionamento de outras ferramentas, como bibliotecas e *frameworks*, permitindo o sistema explorá-las apenas com as estruturas contidas no *Ready*.

Além disso, os serviços *Ready* podem ser utilizados sem o *middleware*, contribuindo com novas funcionalidades à plataforma ROS, e pelo fato de terem acesso a tópicos padronizados, podem contribuir com o fornecimento de pacotes mais modulares.

Após o desenvolvimento estar finalizado, foram realizados testes para comprovar o objetivo principal do trabalho, focando no desenvolvimento de aplicações diversas para robôs domésticos. Pensando nisso, foram construídas três aplicações que se destinam à validar o *framework* de forma completa, utilizando extensivamente as funções disponibilizadas pelo mesmo.

A primeira aplicação, denominada *WatchBot*, foi planejada para ilustrar a vantagem do dispositivo ser capaz de navegar em torno de pontos e referências conhecidas. A aplicação conseguiu utilizar as funcionalidades do *middleware*, dentre elas a navegação e a comunicação, deixando a cargo do sistema o gerenciamento destes recursos garantido que o desenvolvedor pudesse focar apenas na complexidade da própria aplicação.

A segunda aplicação desenvolvida tem o intuito de simular uma agenda, de forma que os comandos de adicionar, remover ou editar contatos são acionados por voz, validando o suporte à comunicação fornecido pelo *Ready*. Por fim, uma aplicação que replica as ações de um garçom de festas foi desenvolvida para validar as funções de comunicação e navegação disponibilizadas pela *Ready*.

Os ensaios realizados envolveram o teste das funções de cada aplicação de forma unitária, ou seja, executando uma aplicação de cada vez, e em grupo, com mais de uma aplicação funcionando em paralelo no sistema, comprovando o controle de recursos do *Framework* desenvolvido.

Os testes realizados mostraram-se satisfatórios, produzindo resultados que confirmaram a viabilidade do trabalho e validaram os objetivos definidos.

A construção desde trabalho agregou muito a todos os membros da equipe. As dificuldades alcançadas ajudaram a aumentar a maturidade e resiliência, e as pesquisas contribuíram a elevar o nível de conhecimento técnico de cada integrante, com informações que com certeza serão úteis no decorrer de nossas vidas profissionais.

A ideia de construir um sistema que pudesse de alguma forma agregar valor à sociedade foram fatores que sempre motivaram a equipe, mas ao iniciar, não tinha-se ideia da proporção que o trabalho iria tomar, e das possíveis dificuldades a serem superadas, sendo assim, mesmo com o sistema funcionado como esperado, e atingido todos os objetivos previsto, ainda existem alguns detalhes que poderiam ser melhorados, os quais foram incluídos na lista de implementações futuras da subseção 5.1.

O código completo da implementação desde *framework* encontra-se, para consultas mais aprofundadas, disponível no seguinte repositório:

- https://github.com/LuizHenriqueP/ReadyFramework

## 5.1 Implementação para futuras versões

Durante o desenvolvimento do trabalho, alguns recursos inicialmente planejados para o sistema não foram implementados, uma vez que consumiriam tempo do desenvolvimento e não seriam devidamente aproveitados nesta demonstração.

Também, durante o desenvolvimento, novas funcionalidades foram imaginadas, entretanto não foram desenvolvidas, por serem trabalho adicional ao planejado pela equipe, mas que podem expandir a capacidade do sistema.

Esta seção irá descrever algumas dessas ideias que poderão ser adicionar em versões futuras do sistema em futuros trabalhos.

- **Gerenciador de Instalação de Aplicações**

  Por se tratar de um *framework* voltado a robôs domésticos, a equipe imaginou desenvolver uma *Daemon* especializada em instalar as novas aplicações ao sistema, sem a necessidade de o usuário configurar o ambiente antes de as utilizar. Atualmente o sistema conta com a estrutura de arquivos necessária para que este componente possa funcionar, mas a adição deste no trabalho não contribuiria para a demonstração do conceito proposto, uma vez que esta parte do sistema funcionaria apenas durante o procedimento de instalação e portanto nem as aplicações e tampouco as *Daemons* se comunicariam ou requisitariam funções para este componente durante a maior parte do funcionamento do sistema.

- **Adicionar um "Gerenciador de tarefas" ao *Ready***

  Normalmente em sistemas capazes de executar aplicações em paralelo é comum existir um gerenciador de tarefas que permite ao usuário forçar o encerramento de um determinado programa. Assim, desenvolver uma aplicação capaz de realizar esta tarefa seria uma adição valiosa aos sistemas que utilizarão o *framework Ready*, entretanto, considerando que estes são recursos críticos, para garantir a segurança da operação seriam necessários algumas alterações no *middleware* que tomariam tempo do desenvolvimento e não contribuiriam com a demonstração do conceito.

- **Implementar bibliotecas em outras linguagens de programação**

  Utilizar *webservices* como meio de troca de informações entre as aplicações e o *middleware* permite que diversas linguagens de programação possam ser utilizadas para desenvolver aplicação de robôs, inclusive as não compatíveis com o ROS.

  Entretanto, para que este suporte seja possível, é necessário ter uma biblioteca que traduza os *endpoints* fornecidos pela *middleware* para dados e funcionalidades que possam ser usados na linguagem desejada. Acessar diretamente os *endpoints*, embora possível, tornaria o uso do *framework* mais complicado, já que seria necessário o conhecimento dessas estruturas e da tecnologia de *webservices*, prejudicando deste modo o objetivo original de simplificar o desenvolvimento deste tipo de aplicação.

  Considerando o fato que, todas as aplicações implementas durante o desenvolvimento usaram a linguagem de C#, não houve a necessidade de desenvolver bibliotecas para outras linguagens, não justificando investir tempo nesta implementação. Neste caso, a criação de novas bibliotecas foi adiada para próximas versões

- **Sistema de Reconhecimento de imagem (*Daemon*, Serviços)**

  Uma vez desenvolvido a *Daemon* de Iteração Humano-Computador e de Navegação, notou-se que uma *Daemon* capaz de tratar de reconhecimento de imagens seria uma adição interessante ao trabalho. Combinar as *Daemons* existentes a uma capaz de gerenciar as imagens captadas poderia adicionar inúmeras funcionalidades ao sistema, como por exemplo, a possibilidade de que alguns objetos pudessem ser utilizados para gerarem *waypoints* quando reconhecidos, tornando o robô capaz por exemplo de mostrar ao usuário onde ele detectou este objeto.

  Entretanto, além da implementação da nova *Daemon*, seria necessário levantar os requisitos funcionais para a implementação das funcionalidades que esta usaria, dos dados utilizados para alimentá-la e consequentemente, a definição das tarefas que devem ser executadas pelos serviços.

- **Melhorar o controle de recursos da *Daemon AppManager* para funcionar com aplicações abertas**

A *Daemon AppManager* conta com um mecanismo denominado controle de recursos, responsável por verificar se as aplicações que solicitaram a execução em primeiro plano, realmente iniciaram o uso. Caso o tempo limite determinado para o início da utilização de primeiro plano seja superado e a aplicação solicitante não comece à atuação de acordo, o controle de recursos irá removê-la da lista de espera para atuação como primeiro plano, e garantir o direito à próxima da fila, conforme explicado na seção 3.2.5.2.

Este mecanismo se mostrou eficaz durante os testes, atuando conforme definido. Porém, ao utilizar a aplicação *Waiter* 4.1.3, que necessita manter-se em primeiro plano a todo o momento, pois está em constante movimento, percebeu-se a necessidade de ampliar a funcionalidade do mecanismo para controlar também as aplicações que já possuem o direito de atuar em primeiro plano, mas que não liberaram este direito para outras aplicações que também desejam fazer uso dos recursos críticos do sistema.

Esta necessidade só foi percebida durante a realização dos testes, já na fase final do trabalho, pensando nisso, decidiu-se adiar a implementação desta nova funcionalidade para as futuras versões do código.

- **Sistema de manipuladores (*Daemon*, Serviços)**

  Adicionar manipuladores mecânicos, como por exemplo braços e garras, a robôs domésticos aumentaria as possibilidades a serem exploradas pelos desenvolvedores, pois dispositivos com esta característica poderiam ser utilizados para realizar tarefas que envolvem manipulação de objetos, como por exemplo simplesmente apanhá-los ou então utilizá-los de alguma maneira.

  Entretanto, esta funcionalidade não foi implementada na versão atual do sistema por demandar um tempo além do viável para escolher um robô com esta capacidade e as consequentes tarefas como selecionar quais tipos de manipuladores seriam suportados e quais funcionalidades estes poderiam prover.

  Além das escolhas que envolvem quais os componentes físicos suportados pelo sistema, seria necessário dedicar tempo também para a implementação de uma nova *Daemon* e dos respectivos serviços necessários para a realização desta tarefa, além do tempo necessário para o planejamento destas atividades.

  Pensando nisso, decidiu-se incluir o suporte para dispositivos “Manipuladores de Objetos” na lista de implementações para versões futuras.

# Referências

ACKERMAN. *Evan. Fribo: A Robot for People Who Live Alone*. 2018. Disponível em: <https://spectrum.ieee.org/automaton/robotics/home-robots/fribo-a-robot-for-people-who-live-alone>. Acesso em: 23 out. 2019. Citado na página 14.

AFFONSO, L. P. V. e Eliane Vendramini de Oliveira e Adriane Cavichiolli e E. P. Web services: IntegraÇÃo e padronizaÇÃo de serviÇos. *RETEC - Revista de Tecnologias*, v. 7, n. 1, 2015. Disponível em: <https://www.fatecourinhos.edu.br/retec/index.php/retec/article/view/78>. Citado na página 25.

ARAUJO, P. V.; MAZIERO, C. A.; NIEVOLA, J. C. Classificação automática de processos em sistemas operacionais. 2011. Citado na página 45.

BADENHORST, W. The singleton pattern. In: *Practical Python Design Patterns*. [S.l.]: Springer, 2017. p. 23–35. Citado na página 21.

BAILEY, T.; DURRANT-WHYTE, H. Simultaneous localization and mapping (slam): Part ii. *IEEE robotics & automation magazine*, IEEE, v. 13, n. 3, p. 108–117, 2006. Citado na página 29.

BAKKEN, D. Middleware. *Encyclopedia of Distributed Computing*, Kluwer Academic, Dodrecht, The Netherlands, v. 11, 2001. Citado na página 23.

BOILLOT, M. A. *Application programming interface (API) for sensory events*. [S.l.]: Google Patents, 2012. US Patent 8,312,479. Citado na página 25.

CHIJINDU, V. C.; INYIAMA, H. Social implications of robots an overview. *International Journal of Physical Sciences*, Academic Journals, v. 7, n. 8, p. 1270–1275, 2012. Citado na página 10.

COZMO. *Big brain. Bigger personality*. 2019. Disponível em: <https://www.anki.com/en-us-/cozmo>. Acesso em: 23 out. 2019. Citado na página 14.

CYLONJS. *Cylon.JS - Architecture - Made by Humans, for Humans.* 2016. Disponível em: <https://cylonjs.com/>. Acesso em: 19 out. 2020. Citado 3 vezes nas páginas 5, 18 e 19.

CYLONJS. *JavaScript Robotics, By Your Command*. 2016. Disponível em: <https://cylonjs-.com/>. Acesso em: 19 out. 2020. Citado na página 18.

DEACON, J. Model-view-controller (mvc) architecture. *Online][Citado em: 10 de março de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf*, 2009. Citado na página 22.

ELKADY, A.; SOBH, T. Robotics middleware: A comprehensive literature survey and attribute-based bibliography. *Journal of Robotics*, Hindawi, v. 2012, 2012. Citado 2 vezes nas páginas 5 e 24.

FREEMAN, J. *What is an API? Application programming interfaces explained*. [S.l.]: From Infoworld: https://www. infoworld. com/article/3269878/apis/what-is-an . . . , 2018. Citado 2 vezes nas páginas 24 e 25.

GALUPPO, F.; MATHEUS, V.; SANTOS, W. *Desenvolvendo Com C#*. BOOKMAN COMPANHIA ED, 2003. ISBN 9788536303468. Disponível em: <https://books.google.com-.br/books?id=A9eCPgAACAAJ>. Citado 2 vezes nas páginas 31 e 32.

GAZEBOSIM. *Gazebo - Robot simulation made easy.* 2020. Disponível em: <http:/-/gazebosim.org/>. Acesso em: 22 set. 2020. Citado na página 30.

GENERATIONROBOTS. *ROS vs ROS2*. 2019. Disponível em: <https://blog.generationrobots-.com/en/ros-vs-ros2/>. Acesso em: 09 set. 2020. Citado na página 28.

GOBOT. *What is Gobot?* 2019. Disponível em: <https://gobot.io/documentation/getting-started/>. Acesso em: 20 set. 2020. Citado na página 19.

GRAMMENS, JUSTIN. *Gobot Meets IoT : Using the Go Programming Language to Control The "Things" Around Us*. 2015. Disponível em: <https://image.slidesharecdn-.com/googledevconf2015-150322073523-conversion-gate01/95/gobot-meets-iot-using-the-go-programming-language-to-control-the-things-around-us-15-638.jpg?cb=1427011840>. Acesso em: 20 out. 2020. Citado 2 vezes nas páginas 5 e 20.

HALILI, F.; RAMADANI, E. Web services: a comparison of soap and rest services. *Modern Applied Science*, v. 12, n. 3, p. 175, 2018. Citado na página 26.

INFOQ. *Robótica Open Source: Começando com Gazebo e ROS 2*. 2020. Disponível em: <https://www.infoq.com/br/articles/ros-2-gazebo-tutorial/>. Acesso em: 03 nov. 2020. Citado na página 28.

JOHNNY-FIVE. *Johnny-Five - The JavaScript Plataform*. 2012. Disponível em: <http:/-/johnny-five.io/>. Acesso em: 20 out. 2020. Citado na página 20.

MCGINN, C. et al. Towards the design of a new humanoid robot for domestic applications. In: *2014 IEEE International Conference on Technologies for Practical Robot Applications (TePRA)*. [S.l.: s.n.], 2014. p. 1–6. Citado na página 10.

MICROSOFT. *Um tour pela linguagem C#*. 2019. Disponível em: <https://docs.microsoft-.com/pt-br/dotnet/csharp/tour-of-csharp/index>. Acesso em: 03 nov. 2019. Citado na página 32.

MICROSOFT. *Events*. 2020. Disponível em: <https://docs.microsoft.com/pt-br/dotnet-/standard/events/>. Acesso em: 20 aug. 2020. Citado na página 32.

MICROSOFT. *.NET Documentation*. 2020. Disponível em: <https://docs.microsoft-.com/pt-br/dotnet/standard/>. Acesso em: 18 aug. 2020. Citado 2 vezes nas páginas 5 e 32.

MICROSOFT. *Observer Pattern*. 2020. Disponível em: <https://docs.microsoft.com/pt-br-/dotnet/standard/events/observer-design-pattern>. Acesso em: 20 aug. 2020. Citado na página 32.

MICROSOFT. *What is ASP.NET*. 2020. Disponível em: <https://dotnet.microsoft.com/learn-/aspnet/what-is-aspnet>. Acesso em: 20 aug. 2020. Citado na página 33.

MICROSOFT AZURE. *O que é middleware?* 2020. Disponível em: <https://azure.microsoft-.com/pt-br/overview/what-is-middleware/>. Acesso em: 06 out.. 2020. Citado na página 16.

MILANI, A. *MySQL-guia do programador*. [S.l.]: Novatec Editora, 2007. Citado 2 vezes nas páginas 33 e 34.

MOZILLA. *DeepSpeech*. 2017. Disponível em: <https://github.com/mozilla/DeepSpeech>. Acesso em: 24 jun. 2020. Citado 2 vezes nas páginas 29 e 49.

O GLOBO. *O adeus de 'Jibo': 1º robô social deixará de funcionar por falta de servidor*. 2017. Disponível em: <https://oglobo.globo.com/economia/tecnologia/o-adeus-de-jibo-1-robo-social-deixara-de-funcionar-por-falta-de-servidor-23505029>. Acesso em: 03 nov. 2019. Citado na página 15.

OMG. *DATA DISTRIBUTION SERVICE (DDS)*. 2019. Disponível em: <https://www.omg-.org/omg-dds-portal/>. Acesso em: 09 set. 2020. Citado na página 28.

OPENROBOTICS. *Powering the world's robots*. 2020. Disponível em: <https://www-.openrobotics.org/>. Acesso em: 22 set. 2020. Citado na página 30.

ORACLE. *What Are Web Services?* 2013. Disponível em: <https://docs.oracle.com/javaee/6-/tutorial/doc/gijvh.htm>. Acesso em: 09 set. 2020. Citado na página 25.

ORACLE. *12.8 String Functions and Operators*. 2020. Disponível em: <https://dev.mysql-.com/doc/refman/5.6/en/string-functions.htmloperator_sounds-like>. Acesso em: 02 nov. 2020. Citado na página 34.

OSHWA. *Open Source Hardware Association - Definition (English*. 2020. Disponível em: <https://www.oshwa.org/definition/>. Acesso em: 17 set. 2020. Citado na página 22.

REDHAT. *O que é o open source?* 2020. Disponível em: <https://www.redhat.com/pt-br-/topics/open-source/what-is-open-source>. Acesso em: 17 set. 2020. Citado 2 vezes nas páginas 21 e 25.

RICARTE, I. L. M. Programação orientada a objetos: uma abordagem com java. *http://www. dca. fee. unicamp. br/cursos/PooJava/Aulas/poojava. pdf> Acesso em*, v. 29, n. 10, p. 2014, 2001. Citado na página 23.

ROBOTIS. *[ROS 1] SLAM*. 2020. Disponível em: <https://emanual.robotis.com/docs/en-/platform/turtlebot3/slam/>. Acesso em: 12 nov. 2020. Citado na página 62.

ROBOTIS. *TurtleBot*. 2020. Disponível em: <https://emanual.robotis.com/docs/en/platform-/turtlebot3/overview/>. Acesso em: 22 set. 2020. Citado na página 29.

ROS. *About ROS*. 2017. Disponível em: <https://www.ros.org/about-ros/>. Acesso em: 05 nov. 2019. Citado 2 vezes nas páginas 26 e 27.

ROS. *turtlebot3_slam*. 2018. Disponível em: <http://wiki.ros.org/turtlebot3_slam>. Acesso em: 25 nov. 2020. Citado na página 29.

ROS. *Distributions*. 2020. Disponível em: <https://index.ros.org/doc/ros2/Releases>. Acesso em: 09 set. 2020. Citado na página 28.

ROS. *ROS*. 2020. Disponível em: <https://www.ros.org/>. Acesso em: 05 nov. 2019. Citado na página 10.

ROS DISTRIBUTION. *Distribution*. 2010. Disponível em: <http://wiki.ros.org-/Distributions>. Acesso em: 30 out. 2019. Citado na página 27.

ROS SHARP WIKI. *User Documentation*. 2017. Disponível em: <https://github.com/siemens-/ros-sharp/wiki>. Acesso em: 30 out. 2019. Citado na página 33.

ROS WIKI. *TCPROS*. 2013. Disponível em: <http://wiki.ros.org/ROS/TCPROS>. Acesso em: 09 set. 2020. Citado na página 28.

ROS.ORG. *NavigatingTheFilesystem*. 2018. Disponível em: <http://wiki.ros.org/pt_BR/ROS-/Tutorials/catkin/NavigatingTheFilesystem>. Acesso em: 23 set. 2020. Citado na página 27.

ROS.ORG. *Topics*. 2019. Disponível em: <http://wiki.ros.org/Topics>. Acesso em: 23 set. 2020. Citado na página 27.

ROS.ORG. *Understanding ROS Nodes*. 2019. Disponível em: <http://wiki.ros.org/ROS-/Tutorials/UnderstandingNodes>. Acesso em: 23 set. 2020. Citado na página 27.

SELFA, D. M.; CARRILLO, M.; BOONE, M. D. R. A database and web application based on mvc architecture. In: IEEE. *16th International Conference on Electronics, Communications and Computers (CONIELECOMP'06)*. [S.l.], 2006. p. 48–48. Citado 2 vezes nas páginas 5 e 22.

SIEMENS. *Ros Sharp*. 2018. Disponível em: <https://github.com/siemens/ros-sharp>. Acesso em: 01 nov. 2019. Citado na página 33.

SOAPUI. *SOAP vs REST 101: Understand The Differences*. 2020. Disponível em: <https:-//www.soapui.org/learn/api/soap-vs-rest-api/>. Acesso em: 10 set. 2020. Citado na página 26.

SOURCEFORGE. *eSpeak text to speech*. 2016. Disponível em: <http://espeak.sourceforge-.net/>. Acesso em: 09 jun. 2020. Citado na página 28.

TECHWALLA. *The Definition of Web Service EndPoint*. 2018. Disponível em: <https://www-.techwalla.com/articles/the-definition-of-web-service-endpoint>. Acesso em: 26 jun. 2020. Citado na página 26.

TIME. *The 25 Best Inventions of 2017*. 2017. Disponível em: <https://time.com/magazine-/us/5027041/november-27th-2017-vol-190-no-22-u-s/>. Acesso em: 03 nov. 2019. Citado 2 vezes nas páginas 5 e 15.

TSARDOULIAS, E.; MITKAS, P. Robotic frameworks, architectures and middleware comparison. *arXiv preprint arXiv:1711.06842*, 2017. Citado 7 vezes nas páginas 10, 11, 16, 23, 24, 27 e 28.

TURTLEBOT. *TurtleBot*. 2014. Disponível em: <https://www.turtlebot.com/>. Acesso em: 22 set. 2020. Citado na página 29.

TURTLEBOT. *TurtleBot*. 2019. Disponível em: <https://www.turtlebot.com/assets-/images/turtlebot_family.png>. Acesso em: 22 set. 2020. Citado 2 vezes nas páginas 5 e 30.

VOORHEES, D. P. Introduction to model-view-controller. In: *Guide to Efficient Software Design*. [S.l.]: Springer, 2020. p. 175–179. Citado na página 22.

W3RESOURCE. *MySQL SOUNDS LIKE*. 2020. Disponível em: <https://www.w3resource-.com/mysql/string-functions/mysql-sounds_like-function.php>. Acesso em: 31 out. 2020. Citado na página 54.

# Anexos

# ANEXO A – Diagrama de Classes

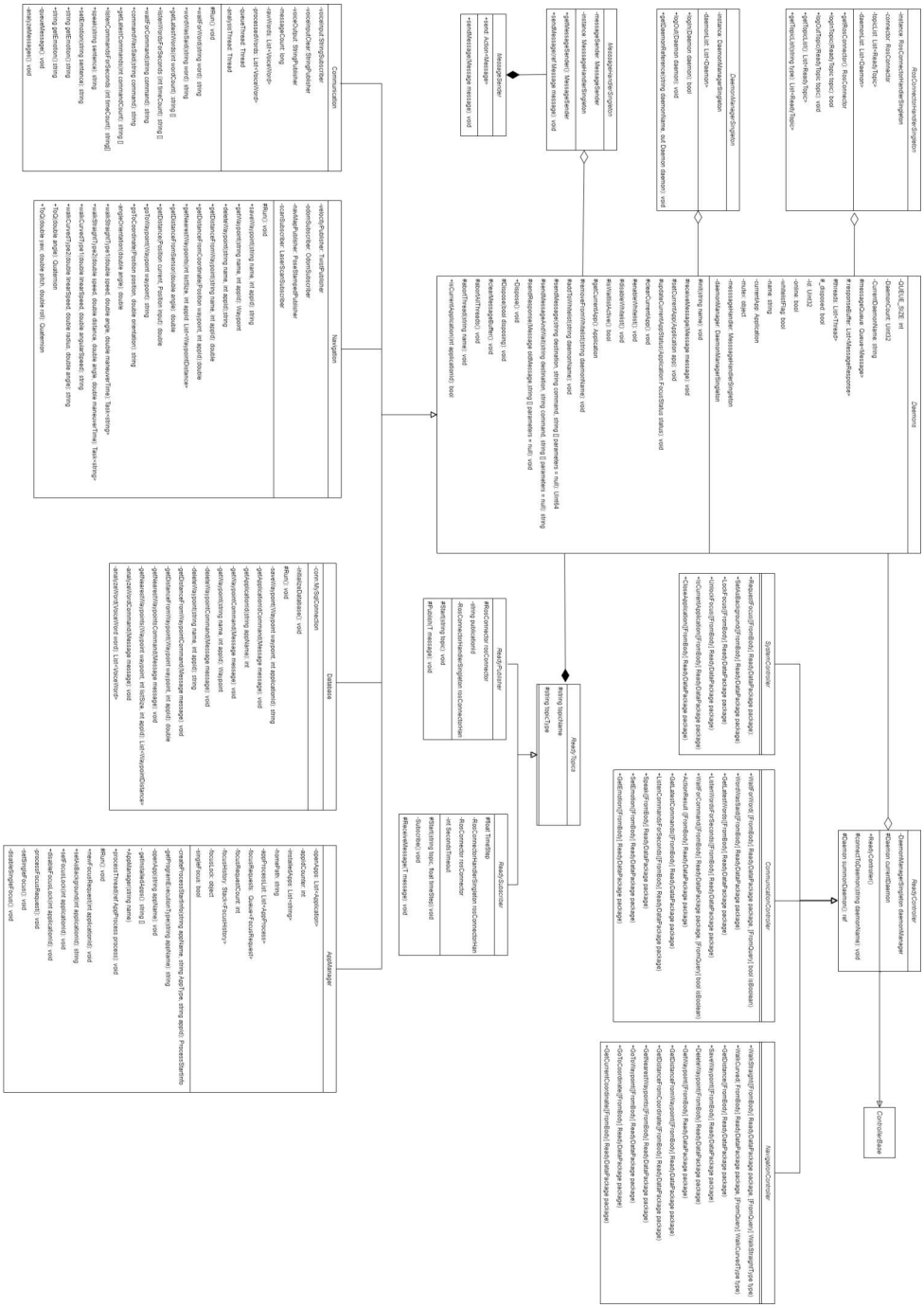


Figura 23 – Diagrama de Classes Completo, uma versão legível encontra-se no repositório do projeto: https://github.com/LuizHenriqueP/ReadyFramework - Fonte: Autoria Própria.

# ANEXO B – Funções da *Daemon* AppManager

- *openApp()*: Esta função é responsável por receber uma requisição de execução de uma aplicação específica instalada no sistema. Ela então busca na base de dados as informações sobre a aplicação e inicializa sua execução, enviando seu identificador único como parâmetro de inicialização. Este identificador único é necessário para viabilizar e validar a comunicação da aplicação com o sistema. Esta função pode ser acessada somente via comunicação interna.

- *newFocusRequest()*: Uma aplicação necessita invocar esta função para ter acesso aos recursos protegidos do *framework*. Como explicado anteriormente, uma requisição de acesso de Primeiro Plano é criada. Esta função é acessada através da *SystemController* pelo *endpoint NewFocusRequest*.

- *setAsBackground()*: Caso uma aplicação esteja em primeiro plano, ela pode requisitar voltar para segundo plano visando liberar os recursos restritos para outra aplicação. Esta função é acessada através da *SystemController* pelo *endpoint SetAsBackground*.

- *setFocusLock()*: Uma aplicação que está em Primeiro Plano pode restringir os recursos protegidos para si através desta função. Esta função é acessada através da *SystemController* pelo *endpoint SetFocusLock*.

- *disableFocusLock()*: Uma aplicação em Primeiro Plano que esteja restringindo os recursos protegidos do sistema deve ser capaz de liberá-los para o uso de outras aplicações, portanto esta função foi desenvolvida com este intuito. Esta função é acessada através da *SystemController* pelo *endpoint DisableFocusLock*.

- *closeApplication()*: Esta função foi desenvolvida para permitir que uma aplicação encerre sua própria execução através do *framework*. Esta função é acessada através da *SystemController* pelo *endpoint CloseApplication*.

# ANEXO C – Funções da *Daemon* de Iteração Humano-Computador

- *waitForWord()*: Esta função recebe como parâmetro uma palavra qualquer a ser esperada por um determinado tempo, em segundos, também enviado como parâmetro. Esta espera consiste em segurar a execução da requisição até que a palavra esteja disponível no *buffer* de mensagens recebidas, retornando a lista de palavras contida na mensagem em que a palavra foi encontrada, conforme o RF01. Esta função é acessível através do *endpoint WaitForWord*.

- *waitForWordBoolean()*: Função similar à *waitForWord*, mas neste caso seu retorno é apenas um valor *booleano* caso a palavra estiver no *buffer* de mensagens recebidas. Ela também é acessível através do *endpoint WaitForWord* enviando o parâmetro *isBoolean* como verdadeiro.

- *wordWasSaid()*: Uma palavra é enviada como parâmetro e a mesma é avaliada se já está disponível no *buffer* de mensagens ou não, visando cumprir o RF02. Esta função é acessada através da *Controller* pelo *endpoint WordWasSaid*.

- *getLatestWords()*: Esta função retorna à aplicação uma lista das últimas palavras detectadas pelo serviço de escuta, sendo informado pela aplicação via parâmetro a quantidade de palavras requisitadas. Seu objetivo é implementar o RF03 e seu acesso está disponibilizado no *endpoint GetLatestWords*.

- *listenWordsForSeconds()*: Esta função recebe como parâmetro um valor de tempo, em segundos, em que irá deixar a requisição em espera. Após passar o tempo definido no parâmetro de entrada, é retornada uma lista com palavras capturadas neste período, de acordo com o RF04. Esta função é acessada através da *Controller* pelo *endpoint ListenWordsForSeconds*.

- *waitForCommand()*: Uma palavra é enviada como parâmetro e a requisição é deixada em espera por um determinado tempo, em segundos, também enviado como parâmetro, até a palavra estar disponível no *buffer* de palavras processadas. Esta palavra deve ser classificada como um comando da aplicação que fez a requisição, conforme especificado no RF05, retornando uma lista de palavras contendo a mensagem original recebida. É acessada através da *Controller* pelo *endpoint WaitForCommand*.

- *waitForCommandBoolean()*: Similar à waitForCommand, está função retorna somente um valor booleano caso o comando se encontre no *buffer* de palavras processadas. Tam-

bém é acessível através do *endpoint WaitForCommand* enviando o parâmetro *isBoolean* como verdadeiro.

- *commandWasSaid()*: Esta função verifica se um comando de aplicação especificado via parâmetro se encontra no *buffer* de palavras processadas e foi devidamente classificado, de acordo com o RF06. O acesso está disponível através da *Controller* pelo *endpoint CommandWasSaid*.

- *getLatestCommands()*: Verifica os últimos comandos de aplicação disponíveis no *buffer* de palavras processadas, sendo a quantidade de comandos retornados enviada via parâmetro, conforme RF07. A função pode ser acessada via *endpoint /GetLatestCommands*.

- *listenCommandsForSeconds()*: Nesta função, um valor de tempo, em segundos, é recebido como parâmetro para que a requisição seja deixada em espera. Após a espera, é retornada uma lista com comandos de aplicação disponíveis no *buffer* de palavras processadas neste período, de acordo com o RF08. Esta função é acessada através da *Controller* pelo *endpoint ListenCommandsForSeconds*.

- *speak()*: Esta função recebe como parâmetro uma mensagem da aplicação. Esta mensagem será então enviada ao serviço de síntese de voz, como especificado no RF09. O acesso está disponível através da *Controller* pelo *endpoint /Speak*.

# ANEXO D – Funções da Daemon de Navegação

- *walkStraight():* Esta função possui duas variações. Elas tem como objetivo comandar o robô em linha reta, em um determinado ângulo em relação a si mesmo. A primeira variação aceita como parâmetros os argumentos *Velocidade*, *Ângulo* e *Tempo de manobra*, e a segunda possui como dados de entradas os mesmos argumentos, mais a *Distância*, com o objetivo delimitar a distância que este movimento deve durar.
- *walkCurved()*: Esta função também possui duas variações, sendo seu objetivo manobrar o robô utilizando trajetórias curvas. A primeira variação utiliza como parâmetros de entrada as velocidades linear e angular, permitindo desenvolver uma curva diferencial. A segunda variação realiza a manobra sobre um raio de um determinado ponto, permitindo que curvas sejam realizadas em torno de algum referencial em relação ao robô.
- *getCurrentCoordinate():* Esta função retorna ao desenvolvedor a atual posição ao robô utilizando os dados dos serviços de navegação.
- *saveWayPoint()*: Esta *Daemon* é capaz de armazenar pontos de interesse que podem ser utilizados e interpretados pelos desenvolvedores de aplicações como pontos de referencia ou até mesmo alguns objetivos. Os *waypoints* são uma estrutura composta por uma coordenada, um nome e o número de identificação da aplicação que a salvou. Deste modo se faz necessário desenvolver um método que permita salvá-los, como descrito no RF05.
- *getWayPoint()*: Esta função tem como objetivo permitir que as aplicações conheçam os *waypoints* salvos no sistema.
- *getDistanceFromWaypoint()*: Esta função permite que o desenvolvedor estime a distância do robô em relação à um determinado ponto de interesse conhecido pelo robô.
- *getDistanceFromCoordinate()*: Esta função permite que o desenvolvedor calcule a distância de uma determinada coordenada em relação ao mapa que robô esteja utilizando.
- *getNearestWaypoints()*: Esta função permite o desenvolvedor obter uma lista de *waypoints* ordenados pela distância utilizando a atual posição do robô
- *gotoWayPoint()*: Permite utilizar a navegação do robô com o objetivo de guiá-lo de maneira autônoma à um *waypoint* conhecido.
- *goToCoordinate()*: Permite utilizar a navegação do robô com o objetivo de guiá-lo de maneira autônoma à uma determinada coordenada em relação ao mapa utilizado pelo robô

# ANEXO E – Funções da *Daemon* de Registros

- *saveWaypoint()*: Esta função é responsável por armazenar um *Waypoint* enviado pela aplicação. O acesso é somente garantido pela comunicação interna do sistema.
- *getApplicationId()*: Esta função retorna o valor de referência armazenado de uma aplicação, dado um nome. Utilizado apenas entre *Daemons*, sendo acessado somente via comunicação interna.
- *getWaypoint():* Responsável por retornar um *Waypoint* registrado da aplicação. É necessário um nome como parâmetro para realizar a busca. O acesso é limitado à chamadas via comunicação interna.
- *deleteWaypoint()*: Função com o objetivo de deletar um *Waypoint* registado pela aplicação utilizando um nome como parâmetro.O acesso é limitado à chamadas via comunicação interna pela própria aplicação.
- *getDistanceFromWaypoint()*: Utiliza informações da posição atual robô e verifica qual é o *Waypoint* da aplicação mais próximo de sua localização atual. Esta função é somente acessada via comunicação interna.
- *getNearestWaypoints()*: A partir da posição atual do robô, são retornados os *Waypoints* mais próximos do mesmo, sendo a quantidade de *Waypoints* especificada pela aplicação. Esta função é somente acessada via comunicação interna.
- *analyzeWord()*: Esta função tem a responsabilidade de analisar uma palavra informada via parâmetro e identificar as possíveis classificações em que ela pode se enquadrar no sistema, sendo elas *SystemCommand, ApplicationCommand, CallWord* e *Alias*. É retornada uma lista com todas as entradas na base de dados que são similares à palavra analisada. Esta funcionalidade pode ser acessada somente via comunicação interna.
- *getSystemCommand()*: Função responsável por retornar as especificações de ação de um comando do sistema passado como parâmetro, acessada somente via comunicação interna.

# ANEXO F – Aplicação *Contacts* - Diagrama de Sequência

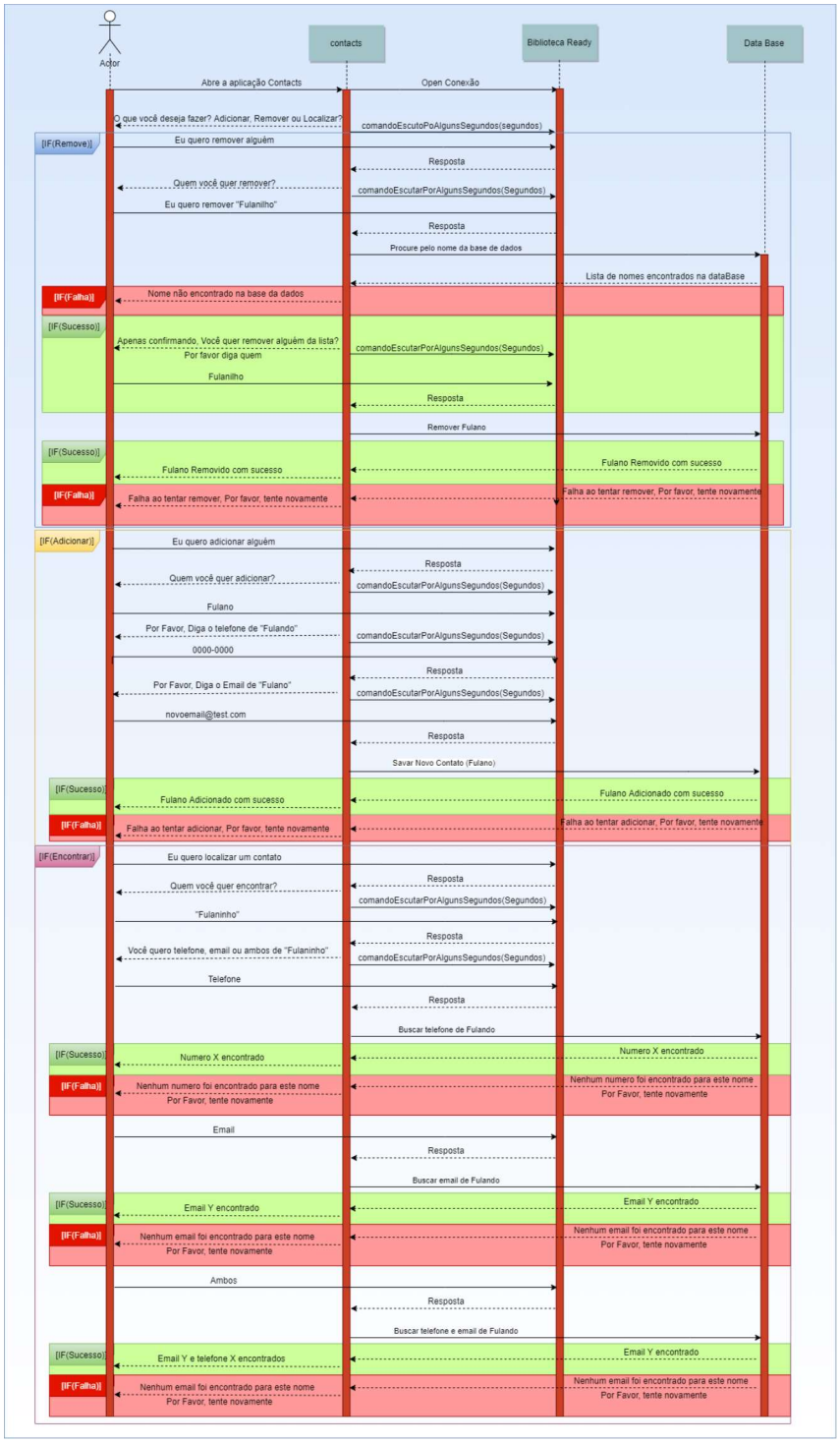



Figura 24 – Diagrama de Sequencia - Fonte: Autoria Própria.

# ANEXO G – Código da Aplicação *Contacts*

```
using System;
using Ready;
using Ready.Models;
using ReadyAppContacts.Models;
using System.Collections.Generic;
using System.Linq;

namespace ReadyAppContacts{
    class Program{
        static void Main(string[] args){
            private const int TimeForListen = 30;
            int id = 2;
            var connection = new ReadyConnection(id, "http://192.168.100.142:5123/");
            var commHandler = new ReadyCommunication(connection);
            var sysHandler = new ReadySystem(connection);
            var contactsDatabase = new ContactsDatabase();
            var closeAppList = new List<string>();

            do{
                commHandler.speak("What do you wish? Remove, Include or Search?");
                var words = commHandler.listenCommandsForSeconds(TimeForListen);
                var wordsList  = new List<string>(words);
                if(wordsList.Exists(x => x == "remove")){
                    commHandler.speak("Who do you wanna remove?");
                    var nameArray = commHandler.listenWordsForSeconds(TimeForListen);
                    var nameList = new List<string>(nameArray);
                    foreach(string name in nameList){
                        var resultNames = contactsDatabase.getContactsName(name);
                        //display phones
                        if(resultNames != null){
                            commHandler.speak("Just for confirm, Do you wanna remove
                            ↪  anyone to this list?");
                            foreach(string eachName in resultNames){
                                commHandler.speak(eachName+", ");
                            }
                            commHandler.speak(" Please, say me who");
                        }
                        else{
                            commHandler.speak("We don't find nothing in your
                            ↪  contacts");
                            break;
```

```
40                          }
41                          var answer  =
                            ↪  commHandler.listenWordsForSeconds(TimeForListen);
42                          var answerList = new List<string>(answer);
43
44                          foreach (string eachName in answerList){
45                              var check = answerList.Where(x => x == eachName
                                ↪  ).FirstOrDefault();
46                              if(check != null){
47                                  bool removed =
                                    ↪  contactsDatabase.deleteContact(eachName);
48                                  if(removed)
49                                      commHandler.speak(eachName +" was removed with
                                        ↪  success");
50                                  else
51                                      commHandler.speak("Failed to remove "+eachName+",
                                        ↪  Please, try again.");
52                                  break;
53                              }
54                          }
55                      }
56                  }
57
58                  if(wordsList.Exists(x => x == "Include")){
59                      commHandler.speak("Who do you wanna include?");
60                      var nameArray = commHandler.listenWordsForSeconds(TimeForListen);
61                      var nameList  = new List<string>(nameArray);
62                      commHandler.speak("Please, tell me the phone number of the "+
                        ↪  nameList);
63                      var numberArray =
                        ↪  commHandler.listenWordsForSeconds(TimeForListen);
64                      var numberList  = new List<string>(numberArray);
65                      commHandler.speak("Please, tell me the email of the "+ nameList);
66                      var emailArray =
                        ↪  commHandler.listenWordsForSeconds(TimeForListen);
67                      var emailList  = new List<string>(emailArray);
68                      bool add1 = contactsDatabase.saveNewContact("aa","bb","cc");
69                      var i=0;
70                      var tamNumber= numberList.Count;
71                      var tamEmail= emailList.Count;
72                      bool add = false;
73                      foreach(string name in nameList){
74                          if(i<tamNumber && i<tamEmail)
75                              add = contactsDatabase.saveNewContact(name, numberList[i],
                                ↪  emailList[i]);
76                          else if(tamNumber<i)
77                              add = contactsDatabase.saveNewContact(name, "",
                                ↪  emailList[i]);
```

```
78                          else if(tamEmail<i)
79                              add = contactsDatabase.saveNewContact(name, numberList[i],
                                ↪  "");
80                          i++;
81                          if(add)
82                              commHandler.speak(name +" was included with success");
83                          else
84                              commHandler.speak("Failed to try include "+name+",
                                ↪  Please, try again.");
85                      }
86                  }
87
88                  if(wordsList.Exists(x => x == "Search")){
89                      commHandler.speak("Who do you wanna find?");
90                      var nameArray = commHandler.listenWordsForSeconds(TimeForListen);
91                      var nameList  = new List<string>(nameArray);
92                      commHandler.speak("Do you wanna the number, email or both?");
93                      var whatSearch =
                        ↪  commHandler.listenWordsForSeconds(TimeForListen);
94                      var whatSearchList  = new List<string>(whatSearch);
95
96                      if(whatSearchList.Exists(x => x == "number")
                        ↪  whatSearchList.Exists(x=>x=="both")){
97                          foreach(string name in nameList){
98                              var number = contactsDatabase.getNumber(name);
99                              commHandler.speak("The "+name+"'s Number is "+number);
100                         }
101                         if(whatSearchList.Exists(x=>x=="both"))
102                             commHandler.speak(" and ");
103                     }
104                     if(whatSearchList.Exists(x => x == "email")
                        ↪  whatSearchList.Exists(x=>x=="both")){
105                         foreach(string name in nameList){
106                             var email = contactsDatabase.getEmail(name);
107                             commHandler.speak("The "+name+"'s email is "+email);
108                         }
109                     }
110                 }
111                 commHandler.speak(|\colorbox{yellow}{("Do you wanna do something more
                    ↪  with the contacts? Please, answer with Yes or No");
112                 var closeApp = commHandler.listenWordsForSeconds(TimeForListen);
113                 closeAppList  = new List<string>(closeApp);
114             }while(closeAppList.Exists(x => x != "No"));
115             sysHandler.closeApplication();
116         }
117     }
118 }
```

# ANEXO H – Código da Base de Dados da Aplicação *Contacts*

```
using System;
using System.Collections.Generic;
using ReadyAppContacts.Models;
using MySql.Data.MySqlClient;

namespace ReadyAppContacts{

    public class ContactsDatabase{
        private  MySqlConnection conn;
        public ContactsDatabase(){
            initializeDatabase();
        }
        ~ContactsDatabase(){
        }

        private void initializeDatabase(){
        string connStr =
        ↪  "server=localhost;user=root;database=Contacts;port=3306;password=10";
        conn = new MySqlConnection(connStr);
        }

        #region getPhoneNumber
        public List<PhoneNumber> getPhoneNumber(string[] words){
            var query = "SELECT phone FROM Contacts @parameters";

            conn.Open();
            var results = new List<PhoneNumber>();
            var parameters = "";

            if(words.Length > 0){
                parameters = String.Format("WHERE {0}",String.Join(" OR ",words));

                using(var cmd = new MySqlCommand(query,conn)){
                    cmd.Parameters.AddWithValue("parameters",parameters);
                    var rdr = cmd.ExecuteReader();

                    while(rdr.Read()){
                        var phone = new PhoneNumber();
                        phone.Number = rdr["phone"] != DBNull.Value ?
                        ↪  rdr["phone"].ToString() : null;
                        phone.Email = rdr["email"] != DBNull.Value ?
                        ↪  rdr["email"].ToString() : null;
```

```
40                          phone.Name = rdr["name"] != DBNull.Value ?
                            ↪  rdr["name"].ToString() : null;
41                          results.Add(phone);
42                      }
43                  }
44              }
45              conn.Close();
46              return results;
47          }
48          #endregion
49
50          #region getNumber
51          public string[] getNumber(string name){
52              var query = "SELECT phone FROM Contacts WHERE name  bz LIKE @name";
53
54              conn.Open();
55              List<string> phone = new List<string>();
56              using(var cmd = new MySqlCommand(query,conn)){
57                  cmd.Parameters.AddWithValue("name",name);
58                  var rdr = cmd.ExecuteReader();
59                  while(rdr.Read()){
60                      var phoneToList = rdr["phone"] != DBNull.Value ?
                        ↪  Convert.ToString(rdr["phone"]) : null;
61                      phone.Add(phoneToList);
62                  }
63              }
64              conn.Close();
65              return phone.ToArray();
66          }
67          #endregion
68
69          #region getEmail
70          public string[] getEmail(string name){
71              var query = "SELECT email FROM Contacts WHERE name SOUNDS LIKE @name";
72
73              conn.Open();
74              List<string> email = new List<string>();
75              using(var cmd = new MySqlCommand(query,conn)){
76                  cmd.Parameters.AddWithValue("name",name);
77                  var rdr = cmd.ExecuteReader();
78
79                  while(rdr.Read()){
80                      var emailToList = rdr["email"] != DBNull.Value ?
                        ↪  Convert.ToString(rdr["email"]) : null;
81                      email.Add(emailToList);
82                  }
83              }
84              conn.Close();
85              return email.ToArray();
```

```
86          }
87          #endregion
88
89          public string[] getContactsName(string name){
90              var query = "SELECT name FROM Contacts WHERE name SOUNDS LIKE @name";
91              conn.Open();
92              List<string> responseName =  new List<string>();
93              using(var cmd = new MySqlCommand(query,conn)){
94                  cmd.Parameters.AddWithValue("name",name);
95                  var rdr = cmd.ExecuteReader();
96
97                  while(rdr.Read()){
98                      var nameToAdd = rdr["name"] != DBNull.Value ?
                        ↪  Convert.ToString(rdr["name"]) : null;
99                      responseName.Add(nameToAdd) ;
100                 }
101             }
102             conn.Close();
103             return responseName.ToArray();
104         }
105
106         public bool deleteContact(string name){
107             var query = @"DELETE FROM Contacts WHERE name = @name";
108
109             conn.Open();
110             try{
111                 using(var cmd = new MySqlCommand(query,conn)){
112                     cmd.Parameters.AddWithValue("name",name);
113                     var rdr = cmd.ExecuteReader();
114                 }
115             }catch (System.Exception){
116                 conn.Close();
117                 return false;
118             }
119             conn.Close();
120             return true;
121         }
122
123         public bool saveNewContact(string name, string email, string number){
124             var query = @"INSERT INTO Contacts(name,phone,email)
125             VALUES (@name, @phone, @email)";
126
127             conn.Open();
128             bool saved = false;
129
130             try{
131                 using(var cmd = new MySqlCommand(query,conn)){
132                     cmd.Parameters.AddWithValue("name",name);
133                     cmd.Parameters.AddWithValue("phone",number);
```

```
134                     cmd.Parameters.AddWithValue("email",email);
135                     var rdr = cmd.ExecuteReader();
136                 }
137             }
138             catch (System.Exception ex){
139                 Console.WriteLine(ex.Message);
140                 conn.Close();
141                 return saved;
142             }
143             conn.Close();
144             saved = true;
145             return saved;
146         }
147     }
148 }
```

# ANEXO I – Código da Aplicação *WatchBot*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Ready;
using Ready.Models;
using System.Threading;

namespace ReadyTest{
    class Program{
        static void Main(string[] args){
            int id = 1;
            var connection = new ReadyConnection(id, "http://192.168.100.142:5123/");
            var commHandler = new ReadyCommunication(connection);
            var navHandler = new ReadyNavigation(connection);
            var sysHandler = new ReadySystem(connection);
            var orbit = new Orbit();
            var waypoint = navHandler.getWaypoint("table");

            orbit.setOrbitAroundWaypoint(1.0, Orbit.Orientation.Clockwise, waypoint);
            var responseTest = sysHandler.requestFocus();

            while (!sysHandler.isCurrentApplication()){
                Console.WriteLine("nao sou");
            };
            Console.WriteLine("sou");

            var lockAnswer = sysHandler.lockFocus();
            var coordinate = orbit.getCurrentCoordinate();
            navHandler.goToCoordinate(coordinate.Position, coordinate.Orientation);

            while (true){
                var distace =
                ↪  navHandler.getDistanceFromCoordinate(orbit.getCurrentCoordinate().Position);
                if (distace <= 0.3){
                    coordinate = orbit.setCurrentCoordinate();
                    navHandler.goToCoordinate(coordinate.Position,
                    ↪  coordinate.Orientation);
                    var speakResponse = commHandler.speak("going to the next
                    ↪  position!");
```

```
39                      Console.WriteLine(speakResponse);
40                  }
41                  Thread.Sleep(100);
42              }
43              Console.ReadKey();
44          }
45      }
46  }
```

# ANEXO J – Código da Órbita Aplicação *WatchBot*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Collections.Generic;
using Ready.Models;

namespace ReadyTest{
    class Orbit{
        private Waypoint waypoint;
        private Queue<Coordinate> coordinates = new Queue<Coordinate>();
        private Coordinate currentCoordinate;
        public enum Orientation {Clockwise,CounterClockwise};
        private Orientation currentOrientation;
        private  double [] anglesCounter = new double[] { 0.0,
        ↪  45.0,90.0,135.0,180.0,225.0,270.0,315.0};
        private double[] angles = new double[] { 0.0, 315.0, 270.0, 225.0,
        ↪  180.0,135.0, 90.0, 45.0};
        public Orbit(){}

        public Orbit(Waypoint waypoint, Queue<Coordinate> coordinates){
            this.waypoint = waypoint;
            this.coordinates = coordinates;
        }

        public void setWaypoint(Waypoint waypoint){
            this.waypoint = waypoint;
        }
        private void setCoordinates(Queue<Coordinate> coordinates){
            this.coordinates = coordinates;
        }

        public void setOrbitAroundWaypoint( double radius, Orientation orientation,
        ↪  Waypoint waypoint = null){
            if (waypoint != null) {
                setWaypoint(waypoint);
            }
            var coordinates = getOrbitAroundCurrentWaypoint(radius, orientation);
            setCoordinates(coordinates);
            setCurrentCoordinate();
        }
```

```
40
41          public void setCurrentCoordinate(Coordinate coordinate){
42              this.currentCoordinate = coordinate;
43          }
44          public Coordinate setCurrentCoordinate(){
45              var coordinate = coordinates.Dequeue();
46              this.currentCoordinate = coordinate;
47              coordinates.Enqueue(coordinate);
48              return coordinate;
49          }
50
51          public Coordinate getCurrentCoordinate(){
52              return currentCoordinate;
53          }
54
55          private Queue<Coordinate> getOrbitAroundCurrentWaypoint(double radius,
            ↪  Orientation orientation){
56              var coordinates = new Queue<Coordinate>();
57              double rotation = 0;
58              if (orientation == Orientation.CounterClockwise){
59                  rotation = rotation + 90;
60                  foreach (double angle in anglesCounter){
61                      var coordinate = new Coordinate();
62                      coordinate.Position.x = waypoint.Position.x + radius *
                        ↪  Math.Cos(AngleToRadians(angle));
63                      coordinate.Position.y = waypoint.Position.y + radius *
                        ↪  Math.Sin(AngleToRadians(angle));
64                      coordinate.Position.z = 0.0;
65                      coordinate.Orientation = rotation;
66                      rotation += 45;
67                      coordinates.Enqueue(coordinate);
68                  }
69              }
70              else{
71                  rotation = rotation - 90;
72                  foreach (double angle in angles){
73                      var coordinate = new Coordinate();
74                      coordinate.Position.x = waypoint.Position.x + radius *
                        ↪  Math.Cos(AngleToRadians(angle));
75                      coordinate.Position.y = waypoint.Position.y + radius *
                        ↪  Math.Sin(AngleToRadians(angle));
76                      coordinate.Position.z = 0.0;
77                      coordinate.Orientation = rotation;
78                      rotation -= 45;
79                      coordinates.Enqueue(coordinate);
80                  }
81              }
82              return coordinates;
83          }
```

```
84
85          private double AngleToRadians(double angle){
86              return (Math.PI / 180) * angle;
87          }
88      }
89  }
```