FACULDADE DE TECNOLOGIA

SANTO ANDRÉ

Tecnologia em Mecatrônica Industrial

Guilherme Arcas Daniluski

Lucca Baratera Rosa

ROBÔ GUIADO MAPEADOR DE AMBIENTES INTERNOS

Santo André

2018

Guilherme Arcas Daniluski

Lucca Baratera Rosa

ROBÔ GUIADO MAPEADOR DE AMBIENTES INTERNOS

Trabalho de Conclusão de Curso apresentado ao Curso Superior de Tecnologia em Mecatrônica Industrial da FATEC Santo André, orientado pelo Prof. Dr. Edson C. Kitani, como requisito parcial para obtenção do título de Tecnólogo em Mecatrônica Industrial.

Santo André

2018

FICHA CATALOGRÁFICA

D186r

Daniluski, Guilherme Arcas

Robô guiado mapeador de ambientes internos / Guilherme Arcas Daniluski, Lucca Baratera Rosa. - Santo André, 2018. – 63f: il.

Trabalho de Conclusão de Curso – FATEC Santo André.
Curso de Tecnologia em Mecatrônica Industrial, 2018.

Orientador: Prof. Dr. Edson Caoru Kitani

1. Mecatrônica. 2. Robô mapeador. 3. Desenvolvimento. 4. Construção. 5. Automação. 6. Sensores. 7. Ambiente interno. I. Rosa, Lucca Baratera. II Robô guiado mapeador de ambientes internos.

629.89

Fatec CENTRO PAULA SOUZA GOVERNO DO ESTADO DE SÃO PAULO

Faculdade de Tecnologia de Santo André

LISTA DE PRESENÇA

Santo André, 02 de Julho de 2018

LISTA DE PRESENÇA REFERENTE À APRESENTAÇÃO DO TRABALHO DE CONCLUSÃO DE CURSO COM O TEMA: **"ROBÔ GUIADO MAPEADOR DE AMBIENTES INTERNOS"** DOS ALUNOS DO 6º SEMESTRE DESTA U.E.

BANCA

PRESIDENTE:
PROF. EDSON CAORU KITANI_____

MEMBROS:
PROF. LUIZ CELIBERTO_____

PROF. FERNANDO GARUP DALBO_____

ALUNOS:
GUILHERME ARCAS DANILUSKI_____

LUCCA BARATERA ROSA_____

Resumo

Este projeto tem como proposta o desenvolvimento e construção de um robô mapeador de ambientes internos, que será guiado remotamente por um piloto. O principal objetivo do projeto é disseminar o conhecimento de ferramentas, como o ROS e técnicas, como o *SLAM*, que são tecnologias emergentes usadas em muitos dos projetos de automação e que têm surgido nos últimos anos. Este protótipo foi confeccionado utilizando uma câmera Kinect, que colhe informações do ambiente, que posteriormente seriam mandadas para um *Raspberry PI*. Este por sua vez se comunicaria com um computador externo para realizar o processamento das informações. Para poder escolher o sensor que iria captar informações do ambiente, fizemos diversos tipos de testes com diferentes tipos de sensores, como o infravermelho e o ultrassom, porém, seria muito difícil conseguir extrair dados precisos deles devido as suas limitações, por isso, foi escolhido o *Kinect* como o leitor de ambiente. Como resultado, é possível destacar que é possível gerar um mapa a partir dos dados extraídos de um local qualquer, porém, para automatizá-lo é preciso melhorar o sistema odométrico já presente no robô.

Palavras-chave: *SLAM*; Localização e Mapeamento Simultâneos; Sensores; ROS.

Abstract

This project proposes the development of an internal mapping robot, which will be remotely guided by a pilot and the main purpose of the project is to disseminate knowledge of tools such as ROS and techniques such as *SLAM*, which are emerging technologies used in many of the automation projects that have emerged in recent years. This prototype was made using a Kinect, that collects information from the environment, which would later be sent to a *Raspberry PI*, that in turn would communicate with an external computer to perform the information processing. In order to choose the sensor that would capture information from the environment, we did several types of tests with different types of sensors, such as infrared and ultrasound, but it would be very difficult to extract accurate data of them due to their limitations, so it was chosen the Kinect as the environment reader. As a result, it is possible to emphasize that it is possible to generate a map from data extracted from any place, but to automate it, is necessary to improve the odometer system already present in the robot.

Keywords: *SLAM*; Simultaneous Localization and Mapping; Sensors; ROS.

LISTA DE ABREVIATURAS E SIGLAS

SLAM	*Simultaneous Localization And Mapping*
ROS	Robotic Operation System
LiDAR	Light Detection and Ranging
HDMI	High-Definition Multimedia Interface
GPIO	General Port Input/Output
GNU	Gnu's Not Unix
SO	Sistema Operacional
MIT	Massachusetts Institute of Technology
AT&T	American Telephone and Telegraph
RAM	Random Access Memory
FTP	File Transfer Protocol
IDL	Interface Definition Language
IEEE	Instituto de Engenheiros Eletricistas e Eletrônicos
IA	Inteligência Artificial
EKF	Extended Kalman Filter
SDD	Sistema de Direção Diferencial
IF	Information Filters
PF	Particle Filter
TTL	Transistor-transistor logic
BEC	Battery Eliminator Circuit
USB	Universal Serial Bus
SDK	Software Development Kit
LED	Light Emitting Diode
RTAB-MAP	Real-Time Appearance-Based Mapping
FAST	Features from Accelerated Segment Test

Lista de Tabelas

Lista de Ilustrações

SUMÁRIO

1. Introdução

Com o crescente desenvolvimento da robótica autônoma, os principais desafios do desenvolvimento de tais sistema estão sendo cada vez mais comuns fora dos ambientes acadêmicos. Nesses sistemas é necessário que o robô tenha conhecimento do espaço ao seu redor para planejar seus movimentos.

A percepção do ambiente por parte de um sistema robótico autônomo deve-se a informações obtidas por sensores tais como: sensores ultrassónicos ou sistemas de visão computacional, as quais podem ser convertidas em representações gráficas.

As representações gráficas proporcionam meios para que o robô possa planejar seu curso. A aplicação desse conceito aparece de diversas formas na nossa sociedade, como os atuais aspiradores de pó, robôs ajudantes domésticos, robôs exploradores ou até mesmo carros autônomos.

Nos sistemas de mapeamento tradicionalmente implementados em robôs autônomos utiliza-se visão computacional como uma alternativa a sensores mais caros como *LIDAR (Light Detection And Ranging*, detecção e medição de distância através da luz).Este trabalho utiliza *frameworks* voltados para robótica e *SLAM (Simultaneous Localization and Mapping, localização e mapeamento simultâneos)* cuja principal característica é o uso do sensor *RGB-D* Kinect e o método de localização chamado odometria visual, como uma alternativa aos amplamente utilizados *encoders*. Proporcionando alternativas de menor custo ao sistemas que utilizam o sensor *LIDAR* como base.

1.1. Objetivo

Os objetivos deste trabalho são pesquisar e estudar técnicas de desenvolvimento de robótica embarcada para criar um robô autônomo capaz de detectar obstáculos e criar um mapa com as áreas acessíveis e não acessíveis para o mesmo. Este trabalho funciona como uma introdução ao *SLAM (Simultaneous Localization and Mapping, localização e mapeamento simultâneos)* para os integrantes do grupo da Fatec Santo André. O principal objetivo é dar condições de estudo a sistemas mais complexos e modernos relacionados com a robótica autônoma. Adicionalmente, pretende-se contribuir com pesquisas relacionadas a sistemas autônomos dentro da Instituição, bem como o meio acadêmico nacional.

1.2. Metas

As principais metas desse projeto são:

- Criar um sistema de odometria capaz de estimar o deslocamento do robô em um gráfico cartesiano.
- Criar um sistema de sensoriamento capaz de estimar a localização de obstáculos na trajetória do robô.
- Com base no deslocamento do robô e nos obstáculos detectados, criar um mapa em duas dimensões do ambiente.

2. Fundamentação Teórica

Este capítulo aborda todos os conteúdos necessários para o entendimento do trabalho, utilizando como base trabalhos de outros autores que se aprofundam no tema de robótica móvel, mapeamento e localização simultânea e sistemas embarcados.

A partir desses estudos será possível ter melhor compreensão dos temas abordados e assim tornar possível o desenvolvimento do protótipo para então consolidar a realização desse trabalho.

2.1. Trabalhos semelhantes

O estudo feito por dos Santos (2010) para sua tese de mestrado, em 2010, relata o desenvolvimento e aperfeiçoamento do *SLAM* embarcado num robô autônomo e a criação de mapas 3-D a partir das imagens capturadas de uma câmera. Em seu trabalho, é mencionado que é utilizada uma técnica de *SLAM* visual baseada no Filtro de Kalman Extendido, encontrando uma série de características presentes em uma sequência de imagens que serão utilizadas como *landmarks*. Como resultado, ele pôde perceber que as técnicas de mapeamento e localização simultâneos funcionam de modo satisfatório, visto que foi possível estimar a posição atualizada do robô a cada passo.

Já o estudo de Rostock, et al (2016) relata a construção de um robô comandado remotamente capaz de realizar o *SLAM* em um ambiente fechado e plano, dividido em células, e estas terão o estado atualizado por um Microsoft Kinect, sendo depois alinhadas com o mapa para estimar a localização de um robô. Com os resultados obtidos, foi possível analisar que a odometria proveniente de *encoders* não comprometeu o funcionamento do projeto, porém foi possível perceber que o Microsoft Kinect apresenta certa limitação tais como ruídos com distâncias maiores do que 5 metros.

2.2. Raspberry PI

Raspberry PI é um computador do tamanho de um cartão de crédito, criado por Eben Upton. Por ter uma comunidade extensa e bem ativa, custar cerca de 40 dólares e ser bem compacto, o *Raspberry PI* é uma das plataforma de desenvolvimento escolhida para a realização deste trabalho.

Com a popularização de tablets e celulares os microprocessadores ficaram cada vez mais baratos e sua tecnologia foi desenvolvida, foi possível inserir o *Raspberry PI* no mercado como uma placa de computador extremamente útil e barata.

2.3. Aplicações

Por ser um computador totalmente *hackeavel*, os desenvolvedores podem utilizá-lo em diferentes ramos de atuação para construir projetos tecnológicos. Por exemplo, se for necessário fazer um termostato digital é indicado a utilização de um micro controlador comum como o Arduino Uno. Porém, se for preferível fazer um termostato com acesso remoto pela internet, é conveniente utilizar um computador como o *Raspberry PI*, pois a placa possui facilidade em se conectar à *internet*.

Segundo Richardson e Wallace (2013) O *Raspberry PI* não tem apenas uma forma de ser utilizado. É possível usá-lo para assistir vídeos na *internet* ou modifica-lo, tornando-o uma plataforma muito flexível. Algumas das aplicações são:

Computação de propósito geral: É possível usar o *Raspberry Pi* como um computador pessoal para o dia a dia, já que ele possui internet, tem saídas HDMI para vídeo e um sistema operacional bem utilizado, como o Ubuntu.

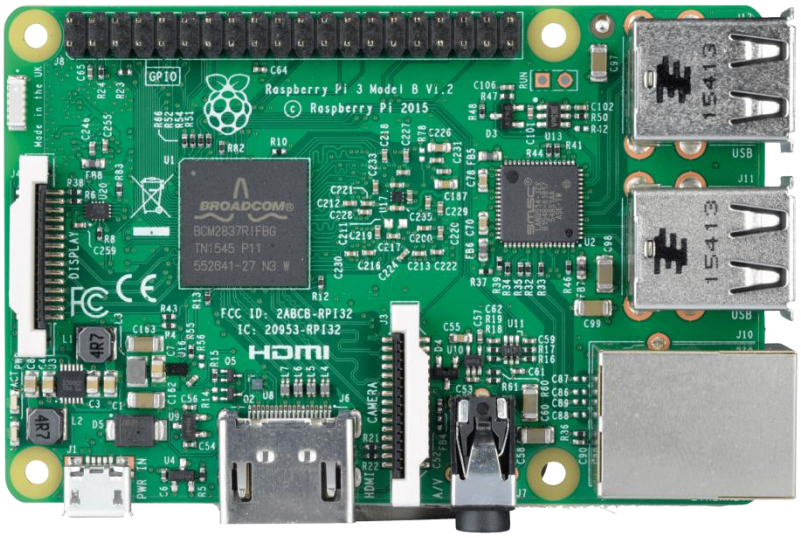
Aprendizado de programação: Por ter sido criado para ser uma ferramenta educacional e incentivar novos desenvolvedores a criarem seus próprios projetos, o *Raspberry PI* vem com alguns compiladores (um programa que traduz linguagem de programação em linguagem de máquina) e interpretadores das linguagens de programação Python, Ruby, C, Perl e Java.

Plataforma de projetos: O Raspberry Pi tem em seu hardware diversos GPIO(*General Port Input/Output*) para serem utilizados em projetos que necessitem de controladores, fazendo com que ele seja muito bom para ser utilizado em projetos.

Prototipagem de produtos: Por ser um computador portátil, o *Raspberry Pi* é bem utilizado na prototipagem de projetos que necessitam de um sistema GNU/Linux embarcado)

2.4. Estrutura do *Raspberry PI*

Figura 1 - Representação dos componentes do Raspberry PI.



Fonte: www.raspberrypi.org.

Tabela 1 - Dados do Raspberry PI

Chip	Broadcom BCM2837
CPU	4× ARM Cortex-A53, 1.2GHz
GPU	Broadcom VideoCore IV
RAM	1GB LPDDR2 (900 MHz)
Comunicação	10/100 Ethernet, 2.4GHz 802.11n sem fio
Bluetooth	*Bluetooth 4.1 Classic*, *Bluetooth Low Energy*
Armazenamento	micros
GPIO	40 terminais macho
Porta	HDMI, 3.5mm conector de áudio-vídeo analógico, 4× USB 2.0, *Ethernet*, *Camera Serial Interface* (CSI), *Display Serial Interface* (DSI)

2.5. Sistema Operacional

Como qualquer outro computador, o *Raspberry PI* tem um sistema operacional, e esse SO é uma versão do GNU/Linux chamada de Raspbian. O Linux se comporta muito bem com *Raspberry PI*, pois é gratuito e de código aberto. Porém, não é limitado apenas ao Raspbian, é possível utilizar outras distribuições de Linux.

2.6. Unix

Segundo Frank Fiamingo, Linda DeBula e Linda Condron (1998), em meados do ano de 1965, a *Bell Laboratories* (departamento de engenharia da empresa norte-americana AT&T) se juntou ao MIT e à General Eletric no intuito de criar um novo sistema operacional multiusuário e multinível chamado Multics. Porém, em 1969, a empresa americana se retirou do projeto, pois não estava satisfeita com o progresso até então.

Sendo assim, alguns programadores da AT&T que estavam no antigo projeto puderam participar de um novo chamado Unix, que seria coordenado por Ken Thompson e Dennis Ritchie. O sistema foi desenvolvido em Assembly para o computador PDP-11(uma máquina de 16KBytes de memória RAM e 512Bytes de espaço em disco). Logo após o desenvolvimento, o sistema operacional foi muito difundido especialmente porque havia uma interface de usuário muito simples, além de ser multiusuário e ter uma arquitetura independente e transparente para o cliente.

Em 1973, a maior parte do código do Unix foi reescrito na linguagem C (criada também por Dennis Ritchie), para que o sistema operacional pudesse se adaptar facilmente a diversos tipos de máquina. A partir daí foram criados muitos projetos baseados em Unix, como os GNU (acrônimo *de GNU's Not Unix*).

2.7. GNU

Segundo Stallman(1998), a história do GNU (O termo GNU foi escolhido, segundo uma tradição *hacker*, como um acrônimo recursivo de "GNU's Not Unix".), se inicia em janeiro de 1984 quando Richard Stallman se propôs a criar um sistema operacional compatível com Unix, para que seus usuários pudessem migrar para o novo *software* livre. Sendo assim, Stallman deixa seu emprego como *hacker* de inteligência artificial no MIT e passa a dedicar seu tempo ao novo Sistema Operacional, pois se ele ainda integrasse a equipe, a universidade poderia reivindicar seu trabalho e assim impor seus próprios termos de distribuição, ou até mesmo transformá-lo em um *software* privado. No entanto, um professor do local o convidou para continuar utilizando as instalações do local.

Inicialmente, o dono do projeto tentou se apoiar no compilador de códigos Pastel e adicionou um *front-end* (parte de um *software* que coleta os dados do usuário) em C em um computador Motorola 68000, mas foi forçado a abandonar o compilador, pois eram necessários alguns *Megabytes* de memória e as máquinas da época não eram capazes de fornecer.

Sendo assim, Richard Stallman estudou o funcionamento do compilador utilizado e então criou um próprio totalmente diferente, o chamado GCC (Coleção de Compiladores GNU) e ainda sim adaptou seu código de *front-end* feito em C.

Em setembro de 1984, Stallman criou o GNU Emacs, um editor de texto para que ele pudesse usar os sistemas Unix para fazer a edição de seus códigos. Logo após, o editor de textos foi colocado em um servidor FTP (Protocolo de Transferência de Arquivos). Contudo, muitas pessoas interessadas não tinham acesso à *internet*, então Stallman começou a vender fitas com o Emacs por um valor de U$150,00; sendo um dos percussores das empresas que hoje distribuem sistemas de GNU/Linux completos.

Como o interesse no Emacs foi crescendo, outras pessoas se envolveram no projeto GNU, e assim foi decidido que era necessário buscar financiamento novamente, e assim foi criada em 1985 a Fundação do Software Livre (FSF em inglês). A FSF assumiu a venda das fitas do editor de textos e mais adiante foi estendido a outros *softwares* e vendas de manuais.

Em 1990, o sistema GNU estava quase completo, mas ainda carecia de um *kernel*. Então foi decidido que o núcleo do sistema operacional seria aplicado em um micro*kernel* desenvolvido na Universidade de Utah chamado Mach. Porém, o desenvolvimento do Mach ficou muito demorado, pois as linhas de execução se tornaram muito complexas de se depurar. Felizmente, em 1992 Linus Torvalds tornou o Linux um *software* livre e assim pôde completar um sistema operacional livre completo.

Porém, fazer os dois funcionarem juntos não foi fácil, algumas ferramentas GNU precisaram ser totalmente modificadas, além de integrar um sistema completo como uma distribuição que funcionasse de maneira genérica.

2.8. Linux

De acordo com Negus(2015)O Linux foi uma das maiores invenções do século XX, sendo que ele impulsionou a criação da internet e encorajou o desenvolvimento e pesquisas para dispositivos computadorizados.

O projeto é um sistema operacional (SO), que consiste no *software* que gerencia o computador e torna possível o uso de aplicações dentro dele. As características que tornam o Linux similar aos outros SO são:

- **Detectar e preparar o *hardware*:** Quando o computador é ligado, o Linux é responsável por fazer toda a interação entre hardware e software, carregando todos os *drivers* e módulos do computador.
- **Gerenciar processos**: O sistema operacional é o responsável por manter diferentes processos funcionando ao mesmo tempo e decidir quando e como eles vão utilizar a unidade de processamento.
- **Gerenciar memória:** O Linux é o encarregado de decidir de que maneira as aplicações usarão a memória RAM para alocar seus dados
- **Prover interface do usuário:** Para o usuário poder acessar o sistema, é necessária uma interface gráfica, e o Linux, por mais que ainda seja comumente utilizado por linha de comando, tem essa funcionalidade disponível.
- **Controlar arquivos:** O sistema operacional é o responsável por controlar a propriedade e o acesso aos diretórios de um sistema.
- **Prover acesso ao usuário e identificação:** O sistema operacional permite que haja a criação de novos usuários e faz com que seja possível a definição de limite entre eles.
- **Oferecer utilidades administrativas:** O Linux possui inúmeros comandos disponíveis, como monitorar a rede, instalar *softwares,* adicionar usuários, etc.

- **Serviços:** O Linux oferece diversas maneiras para manipular as *daemon process* (programas que rodam em um *background*, ao invés de interagir diretamente com o usuário).
- **Agrupamento:** O Linux pode ser configurado para trabalhar em *clusters*, que são diversos sistemas trabalhando juntos em um processo ou em um grupo de processos.
- **Virtualização:** O Linux permite a utilização de maquinas virtuais, no intuito de gerenciar os recursos do computador de forma mais eficiente. Com isso, é possível manejar outros sistemas Linux, Windows e diversos outros sistemas operacionais.
- **Computação em nuvem:** Para gerenciar ambientes virtuais em larga escala, o Linux é bastante utilizado por ser um SO leve, de distribuição livre e desempenho alto.
- **Computação em tempo real:** O Linux pode ser utilizado como um sistema de alto desempenho, para processos de alta prioridade que precisem de muita confiabilidade e respostas muito rápidas.

2.9. História

O *kernel* Linux começou a ser projetado em 1991, quando Linus Torvalds era um estudante na Universidade de Helsinki, Finlândia. Ele tinha como propósito criar um *kernel* (núcleo do sistema, que serve para fazer a interação entre *software* e *hardware*) com base em UNIX, para que pudesse ter mais interação com o Minix (Sistema Operacional baseado em UNIX) naquela época. Assim, Linus lançou sua primeira versão do *kernel* em meados de setembro de 1991. Porém, a falta de compatibilidade com diferentes máquinas fez com que uma segunda versão fosse lançada em cinco de outubro daquele mesmo ano, dessa vez escrita em linguagem C.

Hoje em dia, Torvalds gerencia a Linux Foundation, empresa que visa o desenvolvimento do Linux, assim como a criação de novas ferramentas para o *kernel*. A fundação é apoiada por diversas empresas, entre elas podemos citar a IBM, Oracle, Dell e HP.

O Linux também faz parte do *Open Source Foundation,* que permite outros desenvolvedores alterarem linhas de código de acordo com o que eles precisam para

poder criar novas aplicações. A maioria das regras a seguir servem para proteger a integridade e liberdade dos códigos abertos:

- **Distribuição Livre:** As empresas de código fonte aberto não cobram taxa para ninguém que revenda o *software*
- **Código Fonte:** O código fonte não pode ser restringido em relação à redistribuição
- **Integridade do código fonte do autor:** Se o código fonte for alterado, o autor poderá retirar o nome original do projeto.
- **Sem discriminação contra pessoas ou grupos:** Todas as pessoas podem igualmente utilizar do código fonte.
- **Distribuição da Licença:** Não poderá haver nenhum tipo de licença adicional para usar ou redistribuir o *software*
- **Licença tecnologicamente neutra:** A licença não pode restringir métodos em que o código fonte pode ser redistribuído.

2.10. Robotic Operating System (ROS)

Segundo Quigley, et al (2009) ROS ou Robotic Operating System (Sistema Operacional Robótico) é um framework, ou seja, um conjunto de bibliotecas, módulos, convenções e ferramentas que almejam simplificar o desenvolvimento da robótica. Ele é projetado para atender uma série de desafios encontrados no desenvolvimento de robôs assistentes pessoais como parte do projeto STAIR (*Stanford Artificial Intelligence Robot*) na universidade de Stanford e o Programa de robôs pessoais na Willow Garage. Porém, a arquitetura resultante pode ser utilizada de forma muito mais abrangente do que apenas para robôs assistentes.

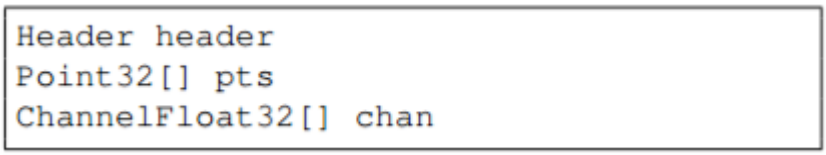
2.11. Características do Framework

Segundo Quigley, et al (2009) O ROS utiliza um sistema que consiste em um número de processos, opcionalmente contendo um número de diferentes *hosts*, conectados em uma topologia ponto-a-ponto, de forma que todos os processos possam se comunicar uns com os outros. Podendo com isso dividir suas tarefas em núcleos isolados, aumentando a manutenibilidade, reciclagem de algoritmos e simplicidade do sistema. A topologia ponto-a-ponto requer alguns mecanismos para

que os processos achem uns aos outros durante a execução, que são *name, service* e *master*.

Segundo Quigley, et al (2009) o ROS também é projetado para ser neutro em relação as linguagens de programação. O ROS tem suporte para quatro linguagens, sendo elas, C++, Python, Octave e LISP, com o suporte para outras linguagens ainda em desenvolvimento. O fato é que o ROS além de permitir que o usuário programe na linguagem de sua preferência, o *framework* permite que processos que utilizam diferentes linguagens se comuniquem de forma harmoniosa no sistema. Para permitir isso o ROS conta com um sistema IDL (*language-neutral interface definition language*) para definir uma estrutura padrão para o envio de mensagens entre os processos. O IDL usa um pequeno arquivo de texto para descrever os campos de cada mensagem e permitir a composição das mensagens, como mostrado na figura 2.

Figura 2 - Estrutura utilizada pelo sistema IDL



Fonte: (QUIGLEY, GERKEY, et al.*, 2009).*

Existe um gerador de código para cada linguagem de programação que “traduz” cada mensagem para a sua respectiva linguagem. Tal sistema reduz o tempo necessário para fazer a interligação entre as linguagens uma vez que essas três linhas acima são expandidas pelo IDL em 137 linhas em C++ por exemplo. Pelo fato de tais mensagens serem geradas automaticamente pelo IDL de um arquivo tão simples, se torna fácil criar novos tipos de mensagens. O resultado é um mecanismo de envio de mensagens neutro em relação às linguagens de programação utilizadas (Quigley, et al.,2009).

Segundo Quigley, et al (2009) com o intuito de gerenciar sua complexidade, o ROS oferece uma gama de pequenas ferramentas para ajudar no gerenciamento do sistema. Tais ferramentas realizam várias tarefas como, por exemplo, navegar pelos arquivos que compõem o sistema, buscar e configurar parâmetros, visualizar a conexão de topologia ponto-a-ponto, medir a utilização de banda larga, criar gráficos

do fluxo de informação entre os processos, automaticamente gerar documentação, ou até tarefas mais especificas como a biblioteca *tf* que é usada para calcular e mapear as juntas dos robôs por exemplo.

Muitos dos *softwares* de robótica existentes contêm *drivers* ou algoritmos os quais podem ser reutilizados em diversos projetos. Infelizmente por uma série de razões, esses algoritmos estão se tornando tão atrelados ao sistema que é difícil extrair suas funcionalidades e usa-las em outras aplicações. De acordo com Quigley, et al.(2009), para combater essa tendência o ROS incentiva que todo o desenvolvimento de *drivers* ou algoritmos ocorram em bibliotecas *stand alone (*completamente autossuficientes*)* sem nenhuma dependência com o *framework*. O ROS utiliza o *CMAKE* (Gerenciador de arquivos) para criar módulos independentes onde o usuário pode inserir toda a complexidade de suas bibliotecas sem alterar o sistema ou outros módulos, e esses módulos são acessados pela forma de pequenos executáveis.

Essa característica confere ao ROS a capacidade de reusar códigos de inúmeros outros *softwares* e *frameworks*. E para se beneficiar das melhorias continuas da comunidade, o ROS é capaz de automaticamente atualizar seu código fonte de repositórios externos, aplicar pacotes de atualização e etc.

O ROS é um *framework* grátis e de código aberto, isso significa que é acessível a todos, tanto para utiliza-lo ou desenvolve-lo. Ele é distribuído sob os termos de licença BDS (*Berkeley Software Distribution*), uma licença que impõe poucas restrições quando comparada com as outras licenças, o que permite que sistemas desenvolvidos com o ROS possam ser utilizados para fins comerciais ou não comerciais. (Quigley, et al,2009)

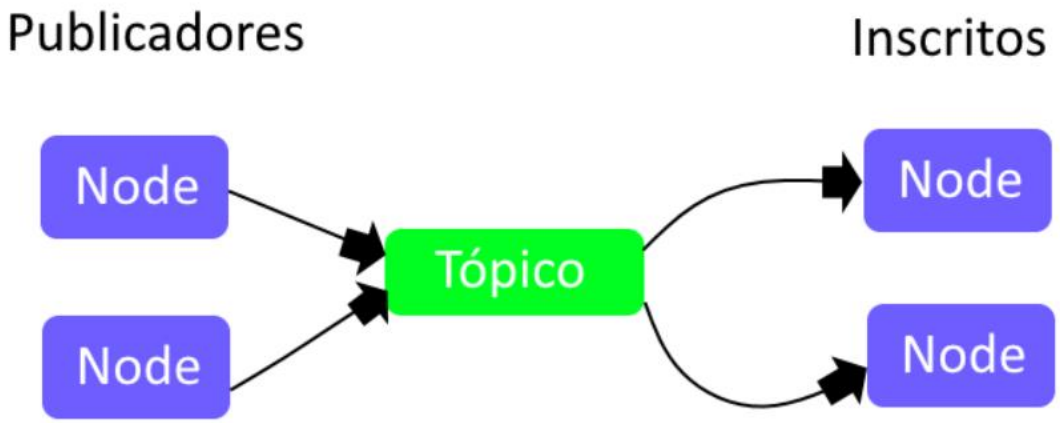
2.12. Estrutura de rede do ROS

A estrutura de rede do ROS é dividida em *Nodes*, mensagens, *topics and services*. Segundo Quigley, et al.(2009), um *node* é um núcleo de processamento, uma unidade modular de código com processamento independente. Um sistema com ROS é tipicamente composto por vários *nodes* sendo que cada *node* pode ter uma atividade específica e independente.

Os *nodes* se comunicam uns com os outros passando mensagens. Uma mensagem é uma simples estrutura de dados, sendo essa estrutura predefinida na elaboração do sistema.

As mensagens suportam desde tipos primitivos (números inteiros, flutuantes, booleanos) até vetores contendo até outras mensagens, ou estruturas.

Figura 3 - Representação do fluxo de dados no sistema de tópicos

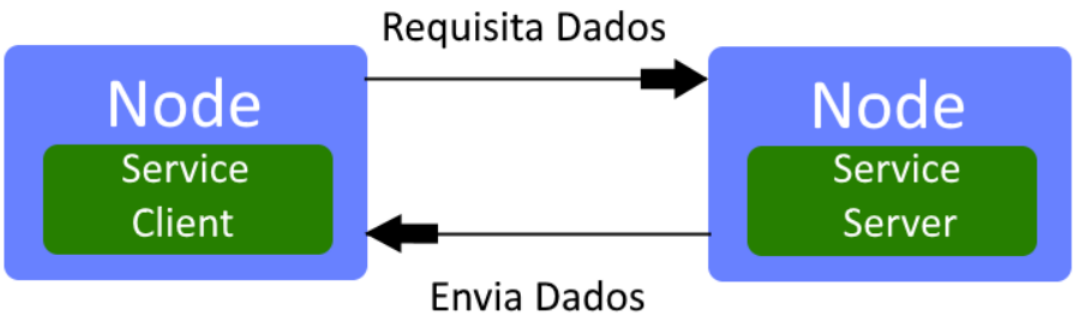


Fonte: Autores

Como ilustrado na figura 3, um *node* envia uma mensagem publicando ela em um tópico, que não é nada mais do que uma *string* (sequência de caracteres, geralmente utilizada para representar palavras, frases ou textos de um programa) como “velocidade” ou “odometria”. Um *node* que está interessado em algum tipo de informação só precisa se inscrever no tópico referente àquela informação. Podem haver diversos *nodes* publicando ou se inscrevendo no mesmo tópico, ao mesmo tempo que um único *node* pode se inscrever ou publicar em diversos tópicos. Geralmente *nodes* inscritos ou que publicam não sabem da existência uns dos outros. Uma vez que o sistema de comunicação via tópicos é altamente flexível, este esquema de difusão de informação não é apropriado para sistemas que requerem sincronia na troca de informações.

No ROS isto é chamado de *service*, que é definido por um nome e um par de mensagens predefinidas: uma para requisição de dados e outra para resposta. Diferente de um tópico, somente um único *nodes* pode oferecer um *service* sob um nome particular. Como ilustrado na figura 4, o *service* do sistema ROS é similar ao *WebService*, sendo que são necessários um servidor e um cliente para haver a comunicação.

Figura 4 - Representação do fluxo de dados no sistema de service.



Fonte: Autores

A estrutura de rede, as ferramentas de visualização e simulação, assim como o fato de ser código aberto, são o que fazem o ROS ser amplamente utilizado pela comunidade de pesquisa. Para mais informações

2.13. Localização e mapeamento simultâneos (*SLAM*)

Localização e mapeamento simultâneos, geralmente citado como *SLAM* (*Simultaneous Localization and Mapping*) é um termo muito abrangente, pode significar um recurso de um robô, uma área da robótica ou uma técnica, mas muitas vezes significa um desafio. Um sistema com *SLAM* consiste em ser capaz de criar um mapa, enquanto se baseia nesse mapa para planejar sua navegação.

Um dos principais desafios da robótica autônoma é fazer com que o robô conheça o ambiente no qual ele atua, para isso ele precisa através de sensores, criar mapas que sejam capazes de lhe informar sua posição perante os obstáculos para assim criar rotas de locomoção.

Em 1986 em uma convenção do IEEE (Instituto de Engenheiros Eletricistas e Eletrônicos) pesquisadores procuravam métodos para resolver os problemas de erros e incerteza provenientes dos sistemas de mapeamento e localização. Naquela época métodos probabilísticos estavam apenas começando a serem implantados em IA (Inteligência Artificial) e na robótica. Muitos pesquisadores se juntaram a essa discussão, e eles chegaram à conclusão de que os problemas envolvidos em mapeamento e localização devem ser tratados como problemas probabilísticos, com maiores dificuldades conceituais e computacionais a serem abordadas.

De acordo com Durrant-Whyte e Bailey(2006), no início, o mapeamento e localização eram tratados como problemas diferentes, mesmo com ambos sendo tratados como problemas de estimativa, mas com o tempo os pesquisadores passaram a enxergar que esses problemas eram convergentes. Uma das conclusões que mais teve impacto sobre a busca do *Santo Graal* do *SLAM* foi em relação aos *landmarks* (pontos de referência).

Um *landmark* é um ponto, uma localização ou um objeto, que é tomado como referência para auxiliar o robô a estimar sua localização O número de *landmarks* está relacionado com parâmetros como resolução do mapa, tolerância de erro admitido, etc. No início os pesquisadores se viram encorajados a criar uma série de aproximações das relações entre os *landmarks* devido à complexidade computacional e a falta de conhecimento do comportamento das convergências do mapa. E esse foi um ponto em que a pesquisa sobre *SLAM* ficou estagnada por um tempo, mas então as coisas começaram a mudar com a abordagem em que o mapeamento e a localização eram convergentes, e que as relações entre os *landmarks*, que até então os pesquisadores procuravam meios de diminuir, era uma parte crítica do problema e, ao contrário do que se pensava, quanto mais as relações entre os *landmarks* crescia menor era o erro apresentado pelo sistema final.

A estrutura do problema do *SLAM*, assim como este acrônimo, foi primeiramente apresentada em 1995 no *International Symposium on Robotics Research*, e nesse tempo o foco das pesquisas era em relação a melhorar o desempenho computacional e os problemas de associação de dados.

Segundo Cadena, et al (2016) esses primeiros 20 anos do *SLAM* são conhecidos como a era clássica (1986-2004). A era clássica viu a introdução das principais formulações probabilísticas do *SLAM*, assim como as primeiras aproximações utilizando filtro de Kalman estendido entre outros. O período subsequente é chamado de analise-algorítmica, no qual se aborda os três principais fundamentos do *SLAM*: observabilidade, convergência e consistência.

Observabilidade: segundo a teoria de controle define um sistema que se possa encontrar seu estado inicial, qualquer que sejam seus dados de entrada e saída. No *SLAM,* segundo Dissanayake (2011), a observabilidade implica diretamente no que diz respeito a capacidade de se resolver o problema do *SLAM*, ou seja, se as

informações disponíveis são suficientes para se determinar a localização e o estado atual do robô mesmo se as características do ambiente são dinâmicas ou não.

Referente à sua observabilidade o *SLAM* é dividido em dois grupos, o *SLAM* absoluto (*world-centric) SLAM* e o *bearing SLAM*. O *SLAM* absoluto é aquele onde a localização do robô é definida pela posição em relação ao globo, se analisado diante dos métodos para identificar a observabilidade de um sistema propostos pela teoria de controle, o *SLAM* absoluto não é observável, mas existem alguns meios de se determinar a localização exata do robô em relação ao globo, que é o caso de um GPS. O *bearing SLAM* consiste na definição da localização inicial do robô ao longo do tempo, baseado em seu comportamento em relação ao mapa utilizando os *landmarks.*

Segundo Wang e Dissanayake (2008) no *SLAM*, o método utilizado para determinar sua observabilidade é chamado *Fisher Information Matrix*. Uma vez que a observabilidade é constatada, o problema do *SLAM* se resume a um problema de mapeamento.

Segundo Dissanayake (2011) O aspecto da convergência trata se a incerteza proveniente das observações dos sensores converge para um número finito, dado um número significativo de leituras e de um período de tempo. Ou seja, a velocidade do sistema, quantas leituras são necessárias se fazer para adquirir a precisão desejada. Tais aspectos são cruciais para o desenvolvimento de aplicações práticas de *SLAM*, como os carros autônomos que necessitam realizar a varredura dos sensores e o tratamento de seus dados em questões de milissegundos. Neste ponto chegamos ao atual problema do *SLAM*, como obter um mapa com dada precisão em tempo mínimo, ou como maximizar a área de cobertura dos sensores com um determinado tempo e uma qualidade do mapa adquirida.

Já a consistência de um mapa criado com *SLAM* está relacionada a um fator, se o estimador é imparcial ou não. Devido ao estudo da consistência, ficou claro que as soluções do *SLAM* baseadas no *EKF* como também as baseadas no filtro de partículas, que são as soluções mais comuns, produzem estimativas inconsistentes, porém mesmo a consistência sendo cientificamente importante, é necessário ressaltar que até mesmo aplicações que envolvam interações com o ser humano podem tolerar certo grau de inconsistência.

Mesmo com os grandes avanços obtidos pela comunidade de pesquisadores dedicados a resolver o problema do *SLAM*, muitas questões precisam ainda ser respondidas e muitos aspectos melhorados, que vão desde aspectos como velocidade dos sistemas de *SLAM* a até conceitos fundamentais. O fato é que cada vez mais o *SLAM* está se tornando acessível, e consequentemente mais pessoas estarão empenhadas em seu desenvolvimento.

2.14. Localização

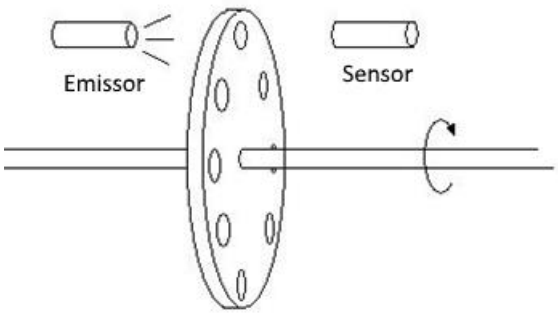
Determinar a localização de um robô no *SLAM* significa determinar sua posição em um mapa, no *SLAM* os métodos mais usados são: Odometria e a Navegação por *Landmarks*.

2.15. Odometria

A odometria é o uso de sensores de movimento para determinar sua mudança de posição ao longo do tempo. Utilizam-se sensores para captar o deslocamento das rodas, com o diâmetro conhecido, é possível estimar o quanto o robô se moveu, assim como sua orientação. A principal ferramenta para se realizar a odometria é o *encoder.* Um *encoder* consiste em um disco que é acoplado ao eixo, nele existem furos ou superfícies reflexivas e não reflexivas que permitem que sensores ópticos sejam capazes de identificar o deslocamento angular da roda.

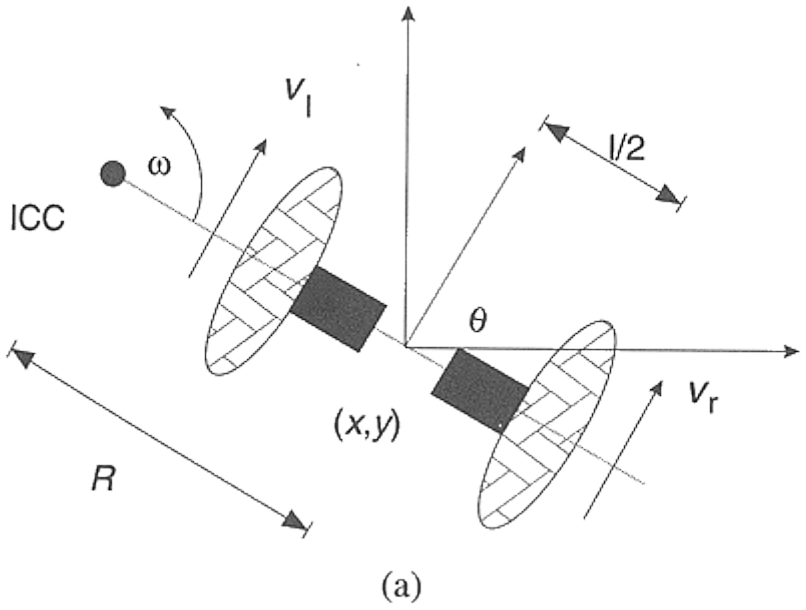
O disco gera pulsos que pela sua quantidade ao longo do tempo é possível saber o quanto a roda se deslocou.

Figura 5 - Representação Encoder.



Fonte: (Dudek e Jenkin, 2000).

Figura 6 - Modelo de um Sistema de Direção Diferencial.



Fonte: (Dudek e Jenkin, 2000).

Segundo Dudek e Jenkin (2000) A partir desse conceito é possível criar modelos do robô capazes de reproduzir a relação entre a movimentação do robô e os *encoders*, o principal modelo utilizado na robótica é o sistema de direção diferencial. Tal sistema consiste em utilizar duas rodas motoras independentes na qual a diferença de velocidade entre as rodas proporciona a mudança de direção do robô. O SDD (Sistema de Direção Diferencial) se baseia na hipótese de que sempre o robô estará percorrendo uma trajetória em formato de arco, a uma distância R do centro da circunferência que prescreve tal arco, e que o que determina R é a diferença da velocidade dos motores. Se as velocidades forem iguais, R tende ao infinito, então o robô percorre uma trajetória em linha reta. Se as velocidades forem iguais, mas em sentidos opostos, R é igual a zero então o robô gira em torno do próprio eixo. Para quaisquer outros valores que as velocidades assumam significa que a trajetória forma um arco em torno do ponto ICC mostrado na figura 6. Na maioria dos casos opta-se por não realizar trajetórias em arco, e sim linhas retas e revoluções para ajustar o ângulo θ, a orientação do robô.

2.16. Navegação baseada em *Landmarks*

Segundo Betke e Gurvits (1997), Dudek e Jenkin (2000) A odometria somente é incapaz de determinar a localização de um robô com precisão, isto porque vários tipos

de ruídos podem afetar as estimativas, ruídos como escorregamento das rodas, irregularidades no terreno e até mesmo as leituras dos *encoders* que podem apresentar erros devidos ao intervalo entre os pulsos gerados. Os principais sistemas de *SLAM* utilizam os *landmarks* juntos com a odometria para uma melhor estimativa da posição do robô em relação ao mapa do ambiente.

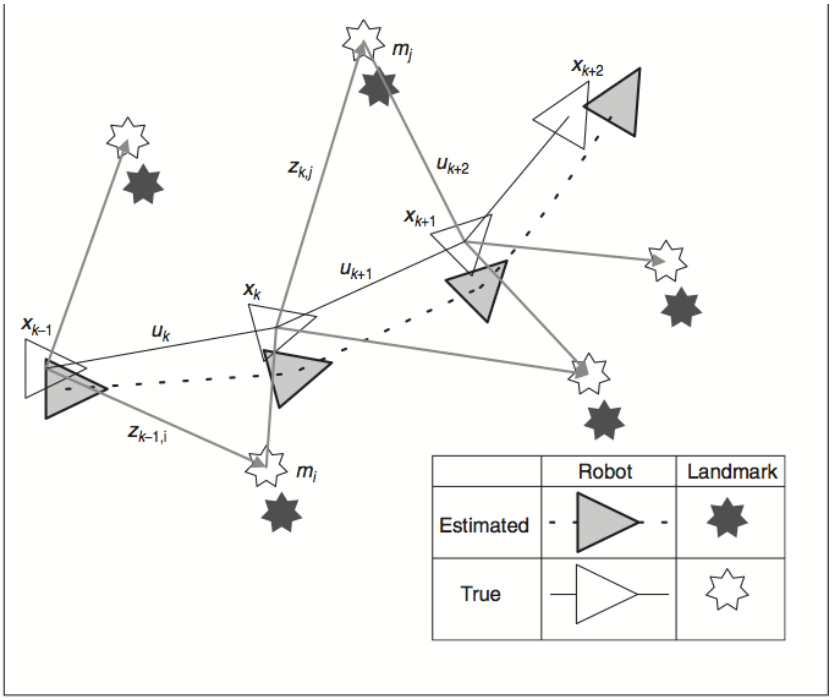
Para utilizar tal técnica é necessário que o robô seja capaz de realizar as seguintes tarefas:

- Identificar *landmarks* no ambiente;
- Encontrar a localização desses *landmarks* no mapa;
- Medir sua distância dos *landmarks*;
- Medir a distância entre os *landmarks*;

Se tais tarefas forem executadas sem erros, três *landmarks* são suficientes para determinar a localização do robô, a menos que o robô e os *landmarks* estejam dispostos de maneira a formar um círculo ou uma linha. Porém, na prática os erros dificilmente serão eliminados e é necessário minimizar seus efeitos.

De acordo com Durant-Whyte e Bailey(2006), os principais erros provenientes das medições do robô, como mostrado na figura 7, ocorrem devido ao fato de que alguns ângulos são medidos com pequenos erros que se acumulam diante das estimativas feitas pelo robô, e muitas vezes dependendo do sensor ou técnica utilizados para identificar os *landmarks,* podem fazer com que o os *landmarks* sejam confundidos com alguma característica ou estrutura do ambiente que não seja um *landmark*.

Figura 7 - Representação da diferença entre a posição estimada e a posição real do robô, causada pelo erro.



Fonte: (DURRANT-WHYTE e BAILEY, 2006)

O robô utiliza um sistema de triangulação para estimar sua localização, porém utilizar tal técnica com leituras ruidosas pode gerar erros consideráveis nas estimativas. Uma maneira de reduzir o efeito de tais ruídos é aumentando o número de *landmarks* utilizados. Porém, isso implica diretamente no poder computacional e na capacidade dos sensores de monitorar um número crescente de *landmarks*. Outro jeito de reduzir os erros de localização é com a implementação de técnicas probabilísticas.

2.17. Técnicas de *SLAM*

Segundo Aulinas, et al (2008) uma vez que o problema do *SLAM* está diretamente relacionado à incerteza e sensores ruidosos, a comunidade tem procurado utilizar técnicas probabilísticas em sua jornada para solucionar o problema. E essas técnicas atacam diretamente o problema visto que explicitamente modelam as fontes de ruídos e seus efeitos nas leituras dos sensores. As técnicas mais famosas são: Filtro de Kalman e Filtro Estendido de Kalman (KF/EKF); Filtro de Kalman Estendido Comprimido (CEKF); Filtros de Informação (IF); Filtro de Partículas (PF); Algoritmo de Maximização da expectativa (EM);

Tabela 2 - Tabela de Comparação entre as principais técnicas de SLAM.

Fonte: (AULINAS, PETILLOT, et al.*, 2008)*

Vantagens	Desvantagens
Filtro de Kalman / Filtro de Kalman Estendido (KF/EKF)	
Alta Convergencia Capacidade de lidar com a incerteza	Suposição Gaussiana Lento em mapas de grande dimensão
Filtro de Kalman Estendido Comprimido (CEKF)	
Incerteza Reduzida Redução do Uso de Memória Consegue lidar com grandes áreas Melhora a consistência do mapa	Requer características robustas Problema de associação de dados Requer a mesclagem de multiplos mapas
Filtros de Informação (IF)	
Estável e Simples Preciso Rapido para mapas de grande dimensão	Problema de associação de dados Pode precisar recuperar a estimativa de dados Em alta definição requer grandes recursos computacionais
Filtro de Partículas (PF)	
Capacidade de lidar com não-linearidades Capacidade de lidar com ruídos não-gaussianos	Complexibilidade
Maximização de Expectativa (EM)	
Ótimo para mapeamento de edifícios Resolve o problema de associação de dados	Ineficiente, caro Instável em grandes cenários Somente bem-sucedido no mapeamento de edificios

Segundo Petttilot, et al. (2008), mesmo que o problema do SLAM tenha sido dado como resolvido em algumas aplicações, não existe somente uma solução, existe a que se encaixa melhor para uma determinada aplicação, sendo que cada solução tem vantagens e desvantagens.

3. Metodologia

Um projeto para a construção de um robô abrange diversas áreas de estudo e cada parte do projeto é passível de mudanças que podem ocorrer de forma repentina devido a uma necessidade que não foi prevista. Por isso, todo o desenvolvimento, mesmo que dividido e classificado em determinadas áreas de estudo ou por meio de uma classificação de prioridade, ocorre de forma paralela, pois os componentes do projeto são dependentes entre si. Com isso a definição do escopo do projeto exige flexibilidade, pois depende dos objetivos do projeto que mudam sempre em função de obstáculos e metas não alcançadas. Este projeto foi dividido em duas partes, projeto do *hardware* e projeto do *software*.

3.1. Projeto do Hardware

Esta parte do projeto consiste em definir todos os componentes físicos do robô. Os elementos de *hardware* e *software* serão descritos a seguir.

3.1.1. Sensoriamento

A primeira etapa do projeto foi a escolha do método de sensoriamento que seria utilizado para a adquirir dados do ambiente, pois o modelo de sensor e o tipo de dado que seria adquirido implica diretamente na escolha do método de *SLAM* que será implementado. Os sensores podem ser classificados em dois tipos: sensores para mapeamento; sensores para localização.

3.1.1.1. Sensores para o Mapeamento

Os sensores para mapeamento são os sensores utilizados para detectar objetos e/ou medir distâncias, são os famosos *range finders*. Eles são responsáveis por toda a medição, e que no final do projeto resultaria nos elementos/obstáculos apresentados no mapa.

Para a escolha destes sensores foram utilizados alguns critérios tais como:

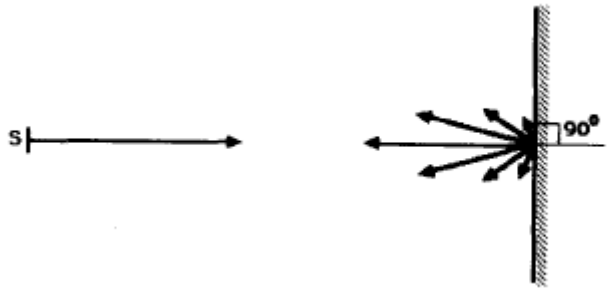
- Preço
- Precisão
- Alcance
- Tamanho
- Facilidade de Implementação

3.1.1.2. Sensores Ultrassônicos

Atualmente, os sensores ultrassónicos medidores de distâncias são amplamente utilizados em diversos campos da engenharia. Geralmente este tipo de sensor detecta o tempo de propagação ultrassónica e usa esse tempo junto com a velocidade de propagação para estimar a distância (YANG, 2008).

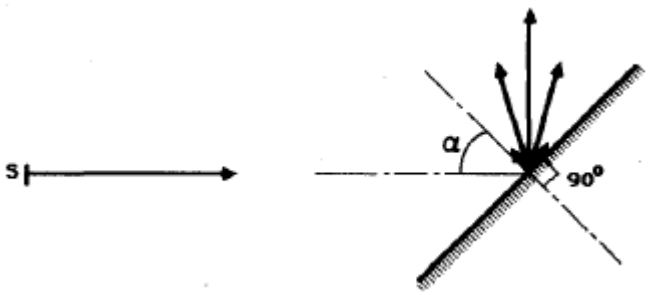
Segundo Borenstein (1986) os sensores ultrassónicos sofrem de algumas desvantagens naturais que limitam a utilização desses dispositivos em mapeamento ou qualquer outra tarefa que requer alta precisão em ambientes internos. Estas desvantagens não estão relacionadas a um produto de um fabricante específico, mas são inerentes ao princípio dos sensores ultrassónicos e seu comumente utilizado comprimento de onda. A seguir serão listados dois casos onde tais desvantagens são apresentadas.

Figura 8 - Reflexão de ondas sonoras em uma superfície lisa e perpendicular.



Fonte:(BORENSTEIN; KOREN, 1986)

Figura 9 - Ondas sonoras refletidas não são detectadas por S quando o angulo α é grande.

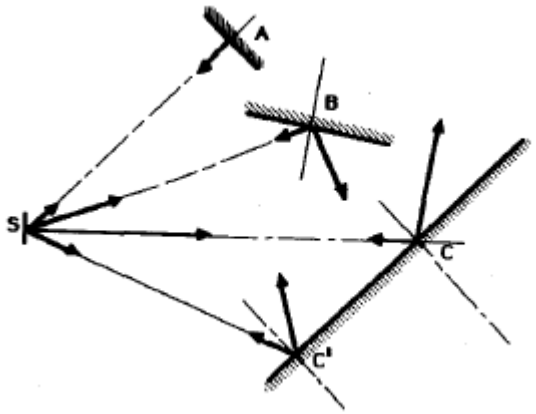


Fonte:(BORENSTEIN; KOREN, 1986)

1º - De acordo com Borenstein(1986), a figura 8 mostra uma parte da onda sonora emitida pelo transceptor ultrassónico S em direção a uma superfície paralela de um obstáculo. Podemos notar que grande parte da energia sonora será

refletida de forma perpendicular à superfície e será detectada por S, enquanto apenas uma pequena porcentagem de energia será refletida em outras direções. No entanto, se a superfície do obstáculo está inclinada em relação ao eixo acústico de S como ilustrado na figura 9, então somente uma parte da energia será refletida em direção a S.

Figura 10 - Incerteza direcional devido ao angulo de abertura da emissão de ondas.



Fonte: (BORENSTEIN; KOREN, 1986)

2º - Outro problema comum é quando a direção de um certo obstáculo tem que ser encontrada precisamente. A emissão das ondas sonoras não é focalizada, as ondas são propagadas em forma de cone e tem um ângulo de abertura de 20-30º. A figura 10 apresenta dois problemas relacionados a este fato. Obstáculo A está na borda do cone acústico e com isso recebe apenas uma pequena parte de energia emitida por S, onde sua orientação é perpendicular as ondas incidentes, resultando em uma otimização de reflexão. Obstáculo B por outro lado recebe mais energia de S pelo fato de estar mais perto do eixo acústico. Porém, a reflexão é pequena devida a orientação não-favorável. Com isso não fica claro qual - ou se algum ao menos - obstáculo foi detectado.

Tais problemas podem ser minimizados utilizando dispositivos que diminuam o ângulo de abertura de emissão, que focalizem as ondas, como por exemplo lentes acústicas ou transceptores dimensionados com uma maior direcionalidade. Porém, se um grande campo de visão é necessário, como no caso do robô tem

que continuamente detectar toda a frente dele, muitos sensores com grande direcionalidade serão necessários para garantir o mapeamento (Borestein1988).

Mesmo com todas essas desvantagens, mas devido ao fato de ser um sensor relativamente barato e de fácil acesso, o sensor HC-SR04 foi adquirido e testes foram realizados.

Figura 11 - Foto do sensor HC-SR04.

Fonte: https://goo.gl/UgJcjV

O HC-SR04 tem o emissor e o detector fisicamente separados e seu funcionamento acontece da seguinte maneira: após o sinal de gatilho, o emissor emite 8 pulsos de onda a 40 kHz, o receptor coleta o eco dessas ondas e emite um sinal TTL cuja duração é tempo entre a emissão do pulso e a recepção do eco, o qual é o tempo de propagação da onda sonora. No capítulo 4 serão apresentados os resultados dos testes realizados com o sensor HC-SR04.

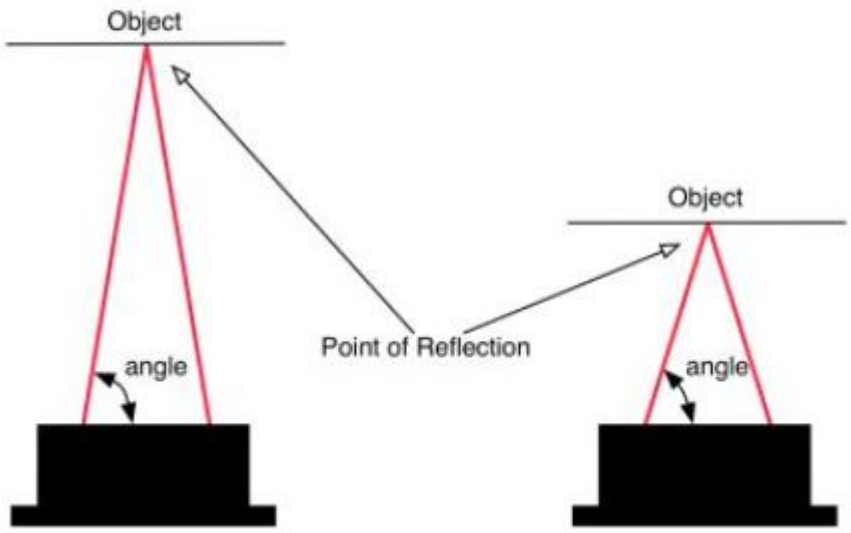
É necessário ressaltar ainda que os sensores ultrassónicos ainda podem ser usados no *SLAM* para evitar colisões visto que podem ser colocados em pontos cegos dos sensores principais.

3.1.1.2.1. Sensores Infravermelho

Sensores infravermelhos podem ser utilizados como alternativa para os sensores ultrassónicos devido a maior precisão de uma maneira geral.

Diferentemente dos sensores ultrassónicos os sensores infravermelhos não utilizam o método de medir o tempo propagação da energia emitida, por sua vez o método utilizado para estimar a distância de um objeto é a triangulação.

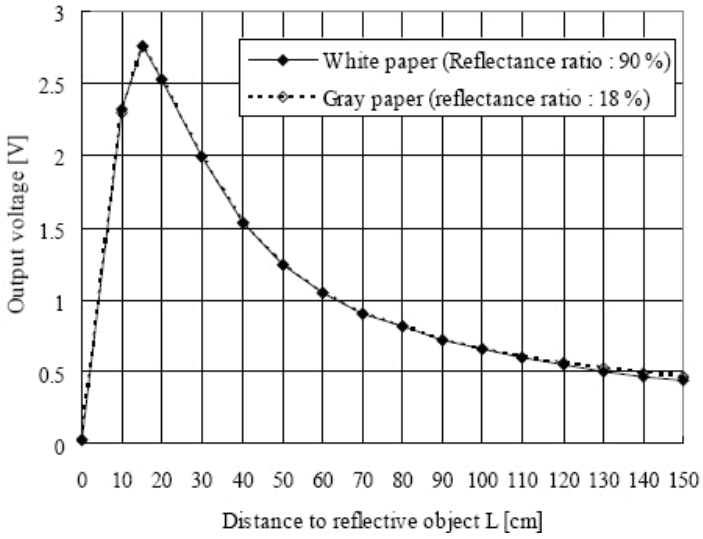
Figura 12 - Representação do método utilizado para estimar a distância de objetos.



Fonte: https://acroname.com/articles/sharp-infrared-ranger-comparison

Como apresentado na figura 12 a angulação do feixe recebido pelo detector varia em função da distância e assim sabendo o ângulo é possível estimar a distância de um objeto. Devido a trigonometria envolvida no processo de computar a distância de um objeto, a saída desses detectores não é linear quando se diz respeito a distância medida, como podemos observar na figura a seguir.

Figura 13 - Curva de reflexão de tensão x distância refletida do sensor GP2Y0A02YK0F



Fonte:https://www.sparkfun.com/datasheets/Sensors/Infrared/gp2y0a02yk_e.pdf

Os sensores infravermelhos fornecem medições com um grau de precisão aceitável quando se tratando de mapeamento de ambientes fechados. Porém, existem desvantagens relacionadas ao uso deste tipo de sensor. A primeira desvantagem é que a qualidade das medições está relacionada às características do ambiente e do objeto detectado, pois como o princípio de funcionamento do sensor utiliza a reflexão de luz infravermelha e a intensidade da luz refletida no objeto é influenciada por fatores como cor e material do objeto assim como a iluminação do ambiente pode influenciar nas medições.

A segunda desvantagem é o campo de visão extremamente limitado visto que o sensor emite apenas um feixe direcional (ACRONAME, 2018). Uma solução para este problema seria girar um ou mais sensores para realizar uma varredura do ambiente ao redor do robô e isso nos leva ao próximo tipo de sensor estudado o LiDAR.

3.1.1.2.2. LiDAR

Segundo Giongo, et al.(2010), LiDAR é uma sigla para *Light Detection and Ranging* que é um termo utilizado para denominar este novo sistema de sensoriamento, o termo faz analogia a sigla RADAR, porém seu funcionamento é extremamente diferente. O LiDAR utiliza raios de luz infravermelha para detectar as

distancias de seus alvos, mas diferentemente dos outros sensores infravermelhos o LiDAR utiliza o tempo de voo do sinal emitido para determinar essas distâncias, utilizando circuitos de altíssima velocidade o LiDAR é capaz de contar o tempo que um feixe de luz demora para atingir um alvo e ser refletido de volta. Esse tipo de método possibilita o uso deste tipo de sensor em plataformas giratórias, fazendo com que o LiDAR seja capaz de realizar varreduras com grandes campos de visão.

O LiDAR em um primeiro momento foi utilizado para criar mapas topográficos, o sensor era montado em satélites ou aeronaves que sobrevoavam seus alvos e realizavam o mapeamento, como resultado temos mapas como o mostrado na figura 14.

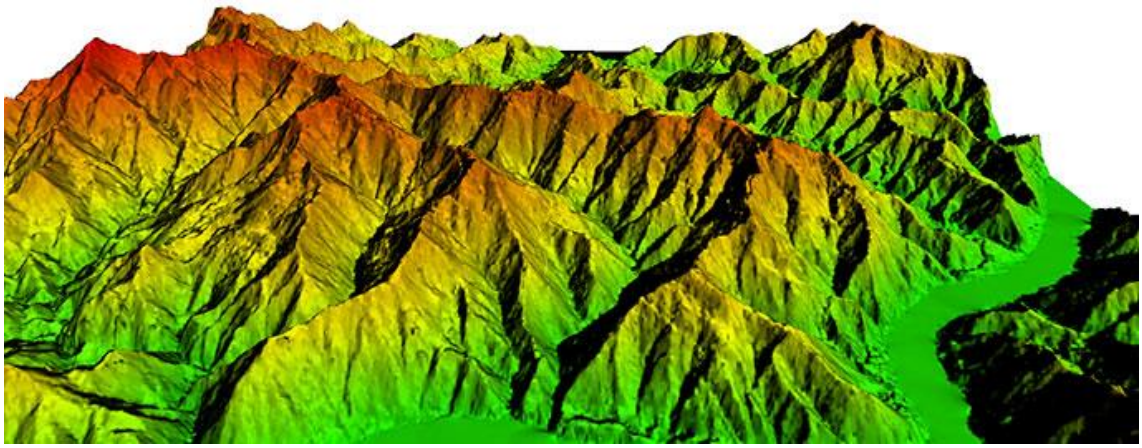


Figura 34 - Resultado da utilização de sensores LiDAR para mapeamento topográfico. Fonte:http://www.aamgroup.com/services-and-technology/aerial-survey

A medida que esta tecnologia se tornava mais popular, diversas outras aplicações foram encontradas, pois como se tratava de um sensor de alta precisão, de longo alcance, e principalmente por se tratar de um sensor que não precisava de luz externa, rapidamente chamou a atenção da comunidade da robótica. E assim surgiram os chamados *mobile LiDAR* que são os principais tipos de sensores utilizados na robótica autônoma.

Os sensores denominados *mobile LiDAR* são utilizados amplamente na robótica autônoma, principalmente em sistemas que utilizam *SLAM*. Estes sensores atualmente chegam a utilizar mais de 64 canais, ou seja, 64 pares de emissores e receptores infravermelho girando a grandes velocidades para detectar o ambiente ao

Figura 15 - Exemplos de LiDARs.

Fonte: http://robotglobe.org/e-stores/sensors/2d-lidar-sensors/

seu redor. Isso possibilita o mapeamento de ambientes extremamente dinâmicos, possibilitando a detecção e monitoramento de quaisquer entidades próximas ao sensor.

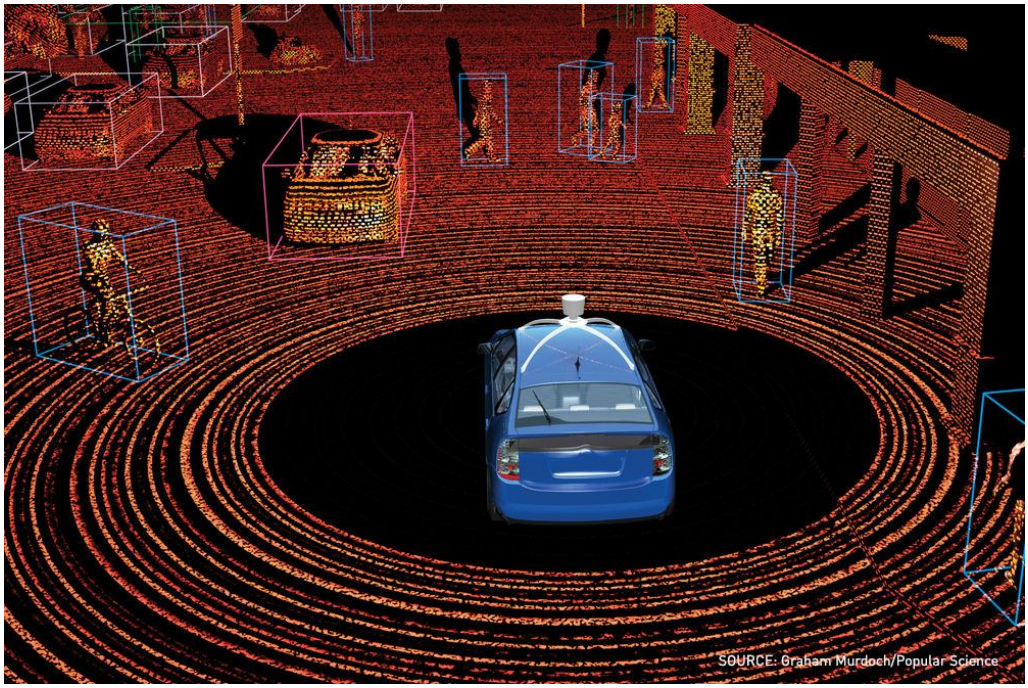


Figura 16 - Representação de um Carro Autônomo utilizado LiDAR para detectar o ambiente. Fonte: https://www.clearpathrobotics.com/2017/01/3d-lidar-true-3d-sensing-spinning-2d-alternatives/

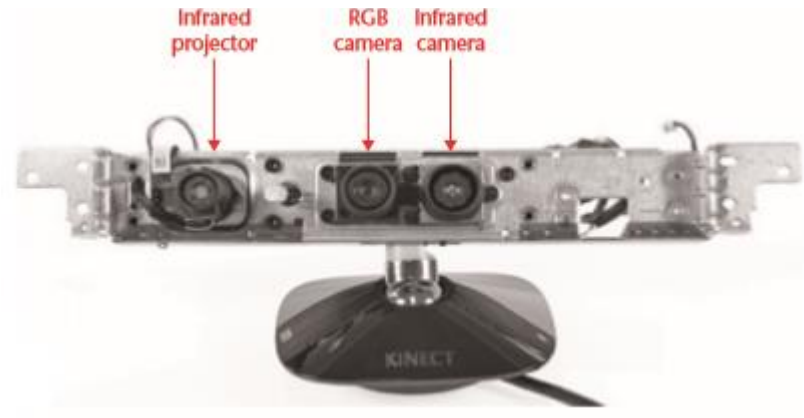
Kinect V1

Figura 17 – Foto do Sensor Kinect V1.

Fonte: (ZHANG, 2012)

Em 4 de novembro de 2010 a Microsoft lançou o Kinect, uma câmera RGBD de baixo custo. O termo RGBD é usado para denominar sistemas de câmeras capazes de além de captar as cores de uma imagem como no caso de câmeras convencionais, mas também para detectar distâncias dos objetos. Para isso o Kinect conta com uma câmera convencional e um sensor de profundidade. O sensor de profundidade é composto por um projetor infravermelho combinado com uma câmera infravermelho.

Figura 18 – Foto ilustrando a disposição dos componentes do sensor Kinect.



Fonte: (ZHANG, 2012)

O projetor infravermelho não é nada mais que um laser que passa por um difrator transformando o feixe único em diversos pontos como apresentado na figura 19.

Figura 19 - Pontos IR Kinect.

Fonte: https://www.engadget.com/2010/11/08/visualized-kinect-night-vision-lots-and-lots-and-lots-of-do/

Cada ponto projetado é distinguível dos outros. É usada uma imagem de referência onde as distâncias de cada ponto são conhecidas, os ângulos assim como a relação para com os pontos vizinhos, de um determinado mesmo ponto são comparados entra as duas imagens e a distância é estimada como na figura 20.

Figura 20 - Princípio de Luz Estruturada.

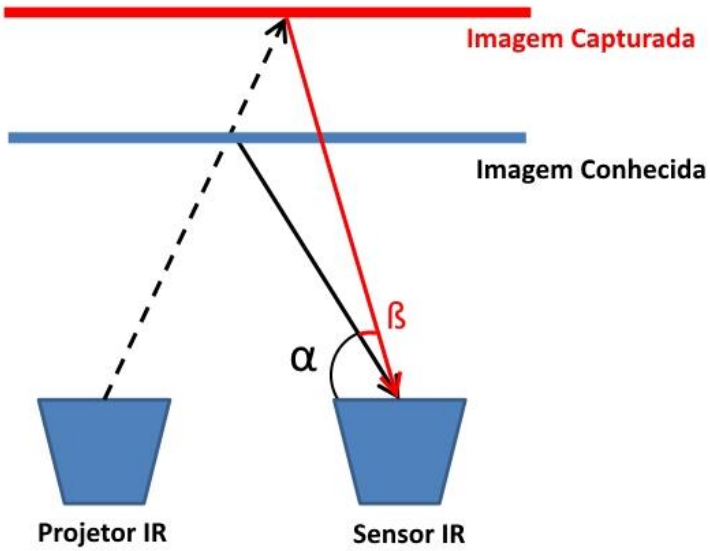


Imagem Adaptada de: https://courses.engr.illinois.edu/cs498dh/fa2011/lectures/Lecture%2025%20%20How%20the%20Kinect%20Works%20-%20CP%20Fall%202011.pdf

Através deste método conhecido como princípio de luz estruturada a Microsoft conseguiu criar um sensor robusto para suas aplicações em ambientes fechados a um preço acessível (ZHANG, 2012). Isso fez com que o Kinect se tornasse um dos sensores mais populares para praticantes da robótica, inclusive projetos de implementação de *SLAM* foram criados utilizando o Kinect, as famílias 1 e 2 do TurtleBot, o robô comercializado pelo instituto ROS utiliza o Kinect como um de seus sensores.

Características do Kinect

- Alcance: 1,8m a 4m
- Campo de Visão: 43° Verticalmente e 57° Horizontalmente
- Taxa de Quadros: 30 Quadros por segundo
- Resolução de Imagem Máxima: 640x480

Por conta de sua precisão, custo e pela gama de projetos semelhantes, e também por que a equipe já o possuí o Kinect, ele foi escolhido como o sensor utilizado para mapeamento neste projeto. Além do mapeamento, o Kinect se tornou extremamente importante em outro aspecto do projeto, a localização.

3.1.1.3. Sensores de Localização

Como explicado anteriormente a técnica mais utilizada para estimar o deslocamento na robótica é a odometria, que se baseia na utilização de *encoders* que são sensores capazes de medir o deslocamento das rodas.

Mas neste projeto não foi utilizada a odometria tradicional ao invés disso foi usado a odometria visual que utiliza as imagens captadas por câmeras para estimar a localização do robô. E para isso foi utilizado o sensor Kinect. Com o sensor definido foi possível planejar qual seriam os outros componentes do *hardware*.

3.1.2. Arquitetura de Hardware

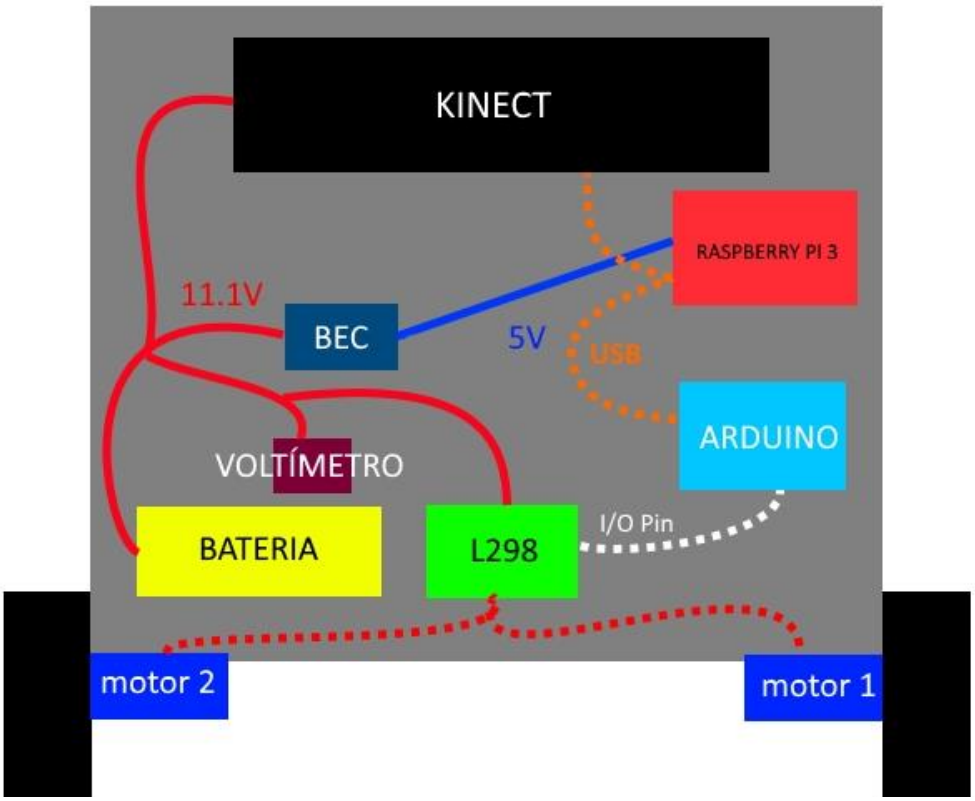


Figura 41 - Representação do robô. Fonte: Autores

Com o tipo de sensor definido foi possível determinar os outros componentes do robô.

3.1.2.1. Controladores

Uma vez que o sensor foi definido, foi preciso determinar com o que seria feito o tratamento desses dados. Para recebimento e tratamento dos dados do Kinect foi utilizado o *Raspberry PI* 3 Model B que como retratado anteriormente se destaca pela facilidade de uso, baixo custo e capacidade computacional. Em conjunto com o *Raspberry PI* foi utilizado o Arduino Uno por conta da extensa familiaridade dos autores com a plataforma de prototipagem, e pela placa fornecer uma alternativa mais prática e dinâmica quando em questão a interação com outros dispositivos de hardware como por exemplo motores, em relação ao que poderia ser alcançado pelo *Raspberry PI*.

3.1.2.2. Motores

Com o sensor e os controladores definidos, neste projeto restam 2 pontos para a construção do hardware, os conjuntos motores e a alimentação elétrica. O robô foi

pensado para ser um robô que usa direção diferencial, que como explicado anteriormente consiste em uma disposição de duas rodas motoras e uma movida, cuja a variação de velocidade entre as rodas motoras proporcionam ao robô a capacidade de virar. Esse modelo foi escolhido pela simplicidade e pelo baixo custo que ele proporciona. Visto isso foi escolhido o motor cuja especificações são:

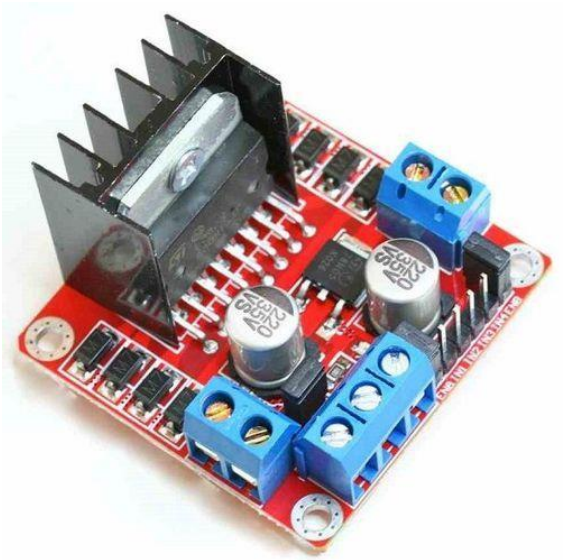
- Tensão Nominal: 6V
- Redução: 1:34
- Rotação Nominal: 210RPM
- Torque Máximo: 5,2kg.cm
- Encoder Integrado: 2 Canais, 341.2 Pulsos por Volta
- Corrente em regime de torque máximo: 1,10A

Figura 22 - Motor DC 6V utilizado no projeto.

Fonte: https://goo.gl/3k9shB

Como driver para o acionamento dos motores foi utilizado uma placa desenvolvida para trabalhar com o Arduino, o SHIELD L298, sua função além de proporcionar uma interface que possibilita controlar o sentido de rotação dos motores e suas velocidades, mas também isola o circuito de acionamento dos motores com o circuito de comando, no caso o Arduino.

Figura 23 - Imagem da placa Shield L298.



Fonte:https://www.indiamart.com/proddetail/l298-shield-16016570191.html

3.1.2.3. Alimentação Elétrica

Na área da robótica móvel outro fator importante é garantir uma autonomia aceitável para sua aplicação e esse fator necessita ser pensado com grande cuidado pois autonomia muitas vezes implica em peso, custo e espaço físico. Pois quanto maior o consumo elétrico, maior a bateria ou a quantidade de baterias necessárias, que implica em maior custo, maior peso e maior necessidade de espaço. Com os outros principais componentes já definidos já é possível escolher a bateria. Em um projeto de um robô duas principais variáveis se destacam na escolha da bateria, que são: Tensão necessária para os componentes operarem; Carga da Bateria. Todo o resto do dimensionamento da alimentação de um robô parte dessas variáveis, segue uma lista com os componentes e suas especificações:

- Sensor Kinect: 12V, 12 Watts
- *Raspberry PI* 3 Model B: 5V, 2 Watts
- Arduino Uno: 5V, 1,6 Watts
- Motor: 12 V, 6,6 Watts

Com estes dados foi possível calcular um consumo de 2820 mAh. Porém, neste primeiro momento de desenvolvimento os motores que são responsáveis por quase 40% do consumo irão atuar em apenas uma pequena parcela do tempo e esse valor para o objetivo proposto na prática é menor.

Apesar de haver uma grande variedade de tipos de baterias, a bateria de Lítio-Polímero foi escolhida. Este tipo de bateria se destaca por ser uma bateria com uma maior densidade energética comparada as outras e uma maior taxa de descarga que é o fator relacionado a quantidade de corrente elétrica que a bateria consegue fornecer.

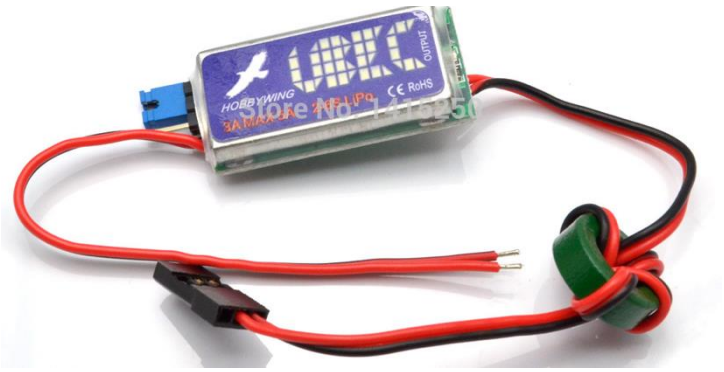
Figura 24 - Imagem da Bateria de LiPo utilizada no projeto.

Fonte: https://goo.gl/hrNRTq 1

A bateria escolhida conta com 3 células, em sua especificação é informado que cada célula da bateria fornece em média 3,7 V e ao todo 11,1 V, mas realizando medições constantes durante o descarregar da bateria foi observado que o valor de tensão de cada célula com a bateria completamente carregada fica em torno de 4,2 V totalizando no caso dessa bateria em torno de 12,6 V, e ao longo do descarregamento esse valor diminui se aproximando do valor especificado somente ao final da carga da bateria. É especificado que a bateria possui uma carga de 2200mAh, esse valor atende ao projeto para fins de desenvolvimento, posteriormente é possível utilizar uma bateria com maior capacidade de carga.

Esta bateria alimenta de forma direta o sensor Kinect, os motores e o *SHIELD* L298, uma vez que tais componentes trabalham com 12 V, além disso o robô conta com um voltímetro embarcado para monitoramento dos níveis de tensão das células da bateria. O *Raspberry PI* por sua vez necessita de 5V de alimentação, para isso foi utilizado um regulador de tensão chamado *BEC* (*battery eliminator circuit*) que rebaixa o nível de tensão de 12 V para 5 V e uma corrente de saída máxima de 3 A.

Figura 25 - Imagem do Regulador de Tensão utilizado no projeto



Fonte: https://goo.gl/HdxY7d

E por fim temos o Arduino Uno que é alimentado pelo *Raspberry PI* através do cabo USB que também é utilizado para transmissão de dados.

3.2. Projeto do Software

Esta etapa descreve:

- Quais passos foram seguidos para obter o *software* funcionando;
- Quais conjuntos de algoritmos foram utilizados;
- Quais resultados foram obtidos;
- Qual a experiência obtida;

Todo o sistema do robô foi desenvolvido baseado no sistema ROS que como explicado anteriormente é um conjunto de *frameworks* e bibliotecas com o foco em facilitar o desenvolvimento da robótica. Essa plataforma prove soluções modulares nos mais diversos campos da robótica, com soluções customizáveis e muitas vezes muito próximas do conceito de *plug and play*. Essas soluções são muitas vezes agrupadas e então tais grupos são denominados *stacks*. Os *stacks* consistem de nodes, *launchers*, bibliotecas, arquivos de configuração(.yaml) e conjuntos de dados(.bag), tudo o que é preciso para o algoritmo ser executado.

3.2.1. Kinect

Na robótica, um dos primeiros desafios de um sistema de controle de um robô consiste em como adquirir e ter acesso aos dados dos sensores. Como o sensor se comunica? Essa é uma das primeiras perguntas, no caso do único sensor utilizado neste trabalho, o Kinect, é utilizado o padrão físico *USB* (*Universal Serial Bus*) para se conectar. Em relação ao protocolo de comunicação, a Microsoft disponibilizou SDKs (*Software Development Kits*) tanto para comunicação com o Kinect como para fornecimento de ferramentas de desenvolvimento de aplicações.

O SDK utilizado neste projeto é o *libfreenect*, também conhecido como *OpenKinect*, ele é um driver para realizar a interface com o Kinect, com ele é possível ter acesso as imagens *RGB* e *Depth*, Motores, Acelerômetros, *LED* e o Áudio do Kinect. O *freenect* é disponibilizado em versão *standalone* tanto para Windows, Mac e Linux, e a versão integrada ao ROS.

Quando o *freenect* é instalado no ROS, ele vem com alguns algoritmos, como por exemplo o *freenect.launch*, um arquivo que inicializa a execução e parâmetros de diversos nodes do *freenect_stack*.

O *freenect.launch* nesta aplicação é executado em um computador, onde o Kinect está conectado, e realiza tais tarefas:

- Inicia e gerencia os Drivers de comunicação com o Kinect, o que resulta na aquisição de duas imagens, uma imagem RGB e uma imagem de profundidade.
- Realiza *depth registration* que basicamente é o alinhamento dos pixels da imagem *RGB* com suas respectivas profundidades.
- Gera um point cloud de cada frame que é basicamente um conjuntos de pontos no espaço e publica no tópico */camera/depth_registered/image_raw* assim como publica outros tópicos informando estado do sensor entre outras coisas.

3.2.2. RTAB-MAP

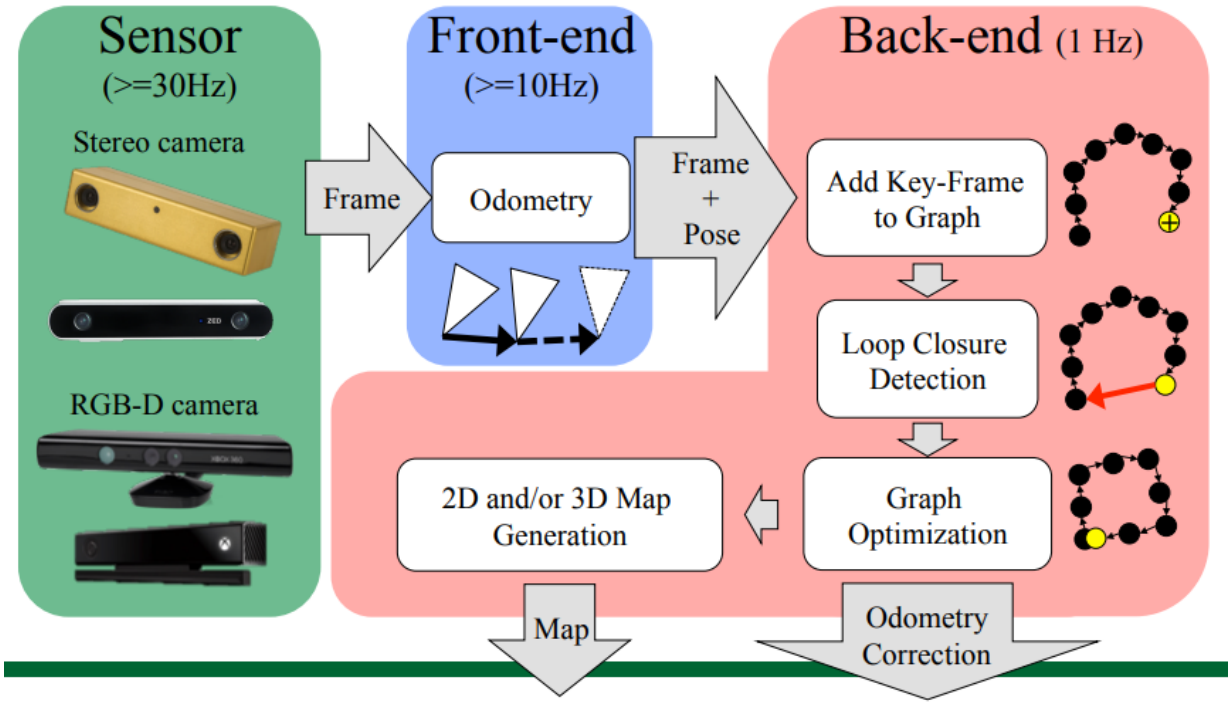
RTAB-MAP ou *Real-Time Appearance-Based Mapping* que significa Mapeamento em tempo real baseado em aparência, é um framework que se baseia em uma abordagem chamada incremental *appearance-based loop closure detector* que é usada para determinar o quão provável é de uma nova imagem adquirida vir de uma localização anterior ou de uma localização nova. Este *framework* é o coração

deste projeto, ele é o algoritmo que realiza o *SLAM* e está sendo executado no *Raspberry PI*.

Feito para trabalhar com câmeras RGB-D o RTAB-MAP fornece uma solução efetiva e prática para o problema do *SLAM*. Este *framework* é disponibilizado para diversos sistemas operacionais, dentre eles estão Windows, OSX, Linux. Além disso o RTAB-MAP é disponibilizado para o ROS.

Segundo Labbé (2015) o funcionamento do RTAB-MAP é dividido em duas etapas, *front-end e o back-end* como representado na figura a seguir.

Figura 26 - Imagem representando o funcionamento do rtab-map



Fonte: https://introlab.3it.usherbrooke.ca/mediawiki-introlab/images/3/31/Labbe2015ULaval.pdf

3.2.2.1. Front-End

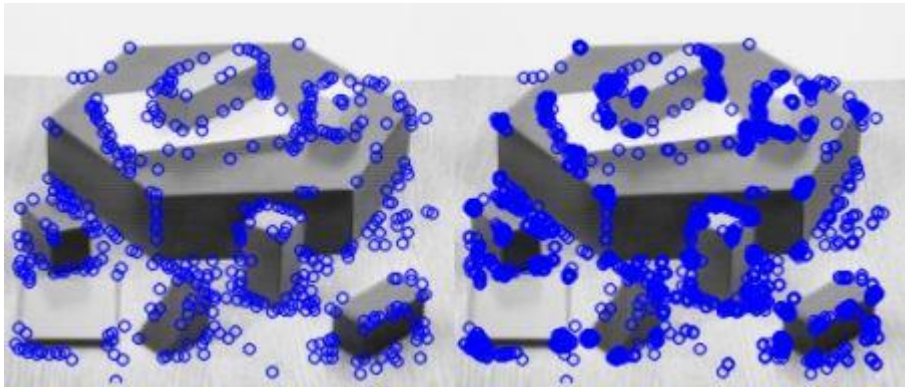
Uma das principais características do RTAB-MAP é que ele não requer uma fonte externa de odometria, ele gera a própria odometria a partir das imagens recebidas através de sensores como câmeras RGB-D ou câmera estéreo. Para isso ele executa algumas etapas:

- *Feature Extraction*
- Comparação de Características entre múltiplos quadros
- *Optical Flow*

A odometria visual observa o deslocamento de pontos específicos nas imagens para estimar o deslocamento do observador. Mas como tais pontos são identificados nas imagens? Utilizando *Feature Extraction*, ou Extração de Características. *Feature Extraction* é uma área de visão computacional que estuda como extrair características únicas de imagens ou regiões de uma imagem, uma vez que tais características são extraídas é possível identifica-las em outras imagens. Antes dessas características serem extraídas é necessário detecta-las, é nessa parte em que entram os *Feature Detectors*.

O tipo de característica que se procura pode variar dependendo da aplicação, no caso do *RTAB-MAP* é utilizado o *FAST* (*Features from accelerated segment test*) um algoritmo para a detecção de características, que basicamente procura por bordas e cantos de objetos nas imagens.

Figura 27 – Imagem representando as features detectadas pelo algoritmo FAST



Fonte:https://docs.opencv.org/3.0beta/doc/py_tutorials/py_feature2d/py_fast/py_fast.html

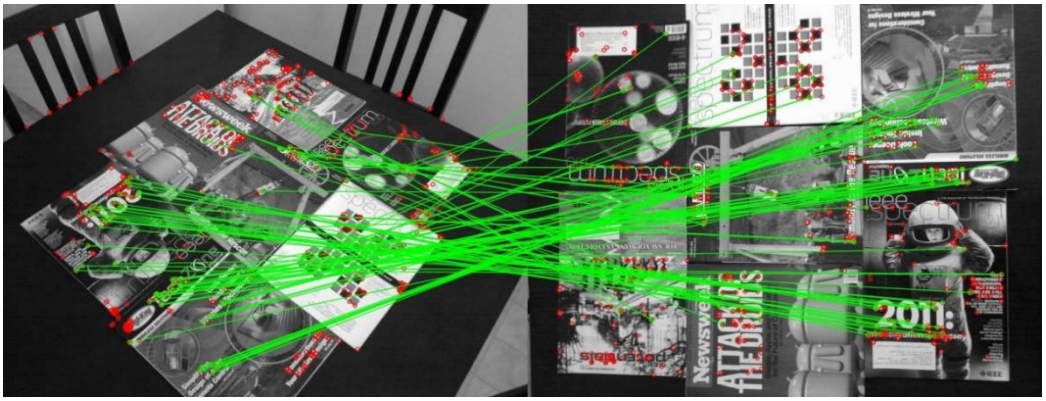
Após características serem encontradas em uma imagem cada característica é submetida a um algoritmo descritor, que basicamente gera uma descrição da região da imagem, uma descrição que é invariante de iluminação e perspectiva. Essa descrição é então armazenada para ser comparada com a descrição dos próximos *frames*.

Existem dois principais tipos de descritores os baseados em histogramas de gradientes orientados e os descritores binários. O *RTAB-MAP* disponibiliza a possibilidade de trabalhar com os principais descritores das duas famílias, mas por padrão utiliza o descritor binário *ORB*.

Se uma descrição de um ponto de uma imagem é comparada e é constatado que representam um mesmo objeto ou ponto no espaço, então é executada uma estimativa de *OpticalFlow* para estimar o deslocamento que o robô sofreu com base no deslocamento das *features* entre as imagens.

Com isso a posição do robô é obtida e a imagem e as *features* detectadas e extraídas são armazenadas para serem utilizadas no *back-end*.

Figura 28 – Imagem representando a detecção das mesmas features de um mesmo objeto em perspectivas diferentes.



Fonte: http://www.willowgarage.com/sites/default/files/orb_final.pdf

3.2.2.2. Back-End

Outra importante característica do *RTAB-MAP* é que ele trabalha com *Loop Clousure Detection*. *Loop Clousure Detection* é uma técnica para detectar se uma

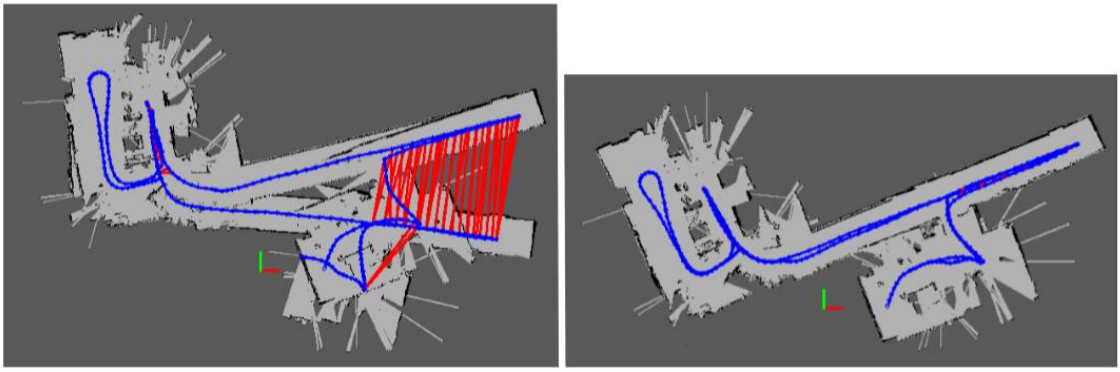
Figura 29 – Imagem mostrando o resultado obtido do algoritmo de loop clousure Detection diante de um frame.

Fonte: https://github.com/introlab/rtabmap/wiki/Loop-closure-detection

localização já foi visitada anteriormente. Todas as *features* das imagens extraídas pela odometria são armazenadas na forma conhecida como *bag of words*, que é basicamente uma classificação de uma imagem armazenada na forma de um documento de texto contendo histogramas de forma esparsa.

Feito isso a cada nova imagem fornecida pela odometria é realizada uma comparação usando filtro de Bayes para detectar se a nova imagem é proveniente de uma localização passada, se é validado que o novo frame representa uma localização já visitada então é utilizado um dos algoritmos de otimização de gráficos já implementados no *RTAB-MAP* como o *TORO* ou *GTSAM* para corrigir erros de localização acumulados durante o mapeamento como mostrado na figura 30.

Figura 30 – Imagem mostrando um mapa sem e com otimização



Fonte: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.666.1195&rep=rep1&type=pdf

Após esta otimização a odometria é corrigida e o mapa é atualizado, e o sistema está pronto para processar outra imagem. Para maior compreensão do framework RTAB-MAP e dos diversos algoritmos que o compõem, consultar: http://introlab.github.io/rtabmap/

3.2.3. Movimentação

A movimentação do robô é controlada através de controle remoto. O controle utilizado é o controle do console Xbox 360, para permitir a comunicação foi utilizado um receptor sem fio, disponibilizado pela Microsoft.

O receptor está ligado em um computador que está rodando o *driver* do controle e uma aplicação do ROS chamada *joynode* a qual converte os comandos recebidos pelo drive do controle e converte no formato de mensagem *joy*, que é o formato padrão do ROS para envio de dados de controles.

No *Raspberry Pi* há uma aplicação que se inscreve no tópico do *joynode* e formata as informações em forma de um vetor, então pública esse vetor em um tópico chamado */joycmd*.

Figura 31 – Controle do Xbox 360 e receptor sem fio.

Fonte:https://www.amazon.in/Xbox-Wireless-Controller-Receiver-Black/dp/B00HJQZS3M

No Arduino é utilizado uma biblioteca chamada *rosserial* cujo objetivo é integrar o Arduino ao sistema ROS, com essa biblioteca o Arduino é capaz de se inscrever em qualquer tópico do sistema e criar e publicar em tópicos que ficam disponíveis para todo o sistema. Com isso o Arduino se inscreve no tópico */joycmd* e uso seus dados para acionar os motores.

Na figura 32 é apresentado a configuração que o sistema está disposto, na figura os principais processadores são representados assim como seus principais processos.

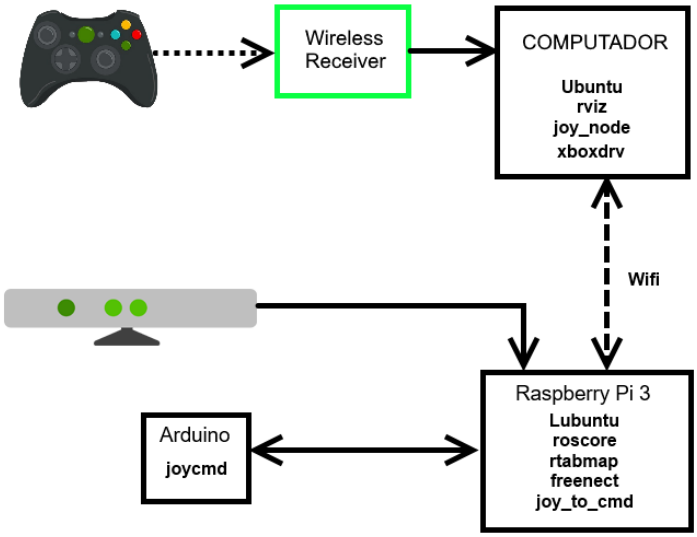


Figura 32 - Representação da arquitetura de software do robô.
Fonte:Autores

4. Testes e Análise dos Resultados

Sensor Ultrassónico HC-SR04

Testes foram realizados para identificar a eficiência do sensor na detecção de objetos domésticos.

1º Teste – Um objeto foi disposto a uma distância conhecida e foram realizadas 200 leituras de 10 em 10 cm até 1 metro de distância do sensor, para identificar a distância cujo erro era menor e a distância cujas leituras eram mais constantes.

Tabela 3 - Tabela com os resultados dos testes de distância e variação das medições

Distância Real(cm)	Erro Médio (cm)	Variância	Média(cm)
10 cm	0,258756	0,00338	9,741244
20 cm	0,65846	0,046598	20,65846
30 cm	0,13950	0,025391	30,1395
40 cm	0,94542	0,022107	40,94542
50 cm	1,02612	0,039598	51,02612
60 cm	1,88284	0,146041	61,88284
70 cm	2,37299	0,114983	72,37299
80 cm	2,77244	0,131519	82,77244
90 cm	2,53816	0,10371	92,53816
100 cm	2,44230	0,068527	102,4423

A tabela 3 mostra os resultados obtidos durante os testes, e com esses dados podemos concluir que a distância com as medições de menor variação é aproximadamente 40cm e a distância com menor erro é a qual o objeto está a uma distância de 30cm do sensor.

2º Teste – Este teste foi realizado para verificar qual o maior ângulo tolerável de inclinação em relação ao eixo acústico que o objeto pode ter (como mostrado na figura 9). Então para este teste um objeto de superfície lisa e plana foi disposto a uma distância de 40cm com os seguintes ângulos de inclinação: 0º, 10º ,20º, 30º, 40º, 50º, 60º. Foram realizadas 8 medições para cada ângulo e os seguintes resultados foram obtidos em centímetros:

Tabela 4 - Resultados obtidos através dos testes de inclinação.

0°	10°	20°	30°	40°	50°	60°
41,07	40,42	41,99	42,77	113,15	153,91	156,81
40,91	40,74	42,12	43,13	3326,1	155,14	157,24
41,02	40,85	41,99	43,24	7,04	153,78	157,37
40,96	40,42	42,1	42,66	113,94	153,89	156,46
41,03	40,85	41,99	43,26	112,69	153,78	156,35
40,91	40,82	42,12	43,24	110,43	153,48	156,81
41,02	40,42	41,99	42,77	109,53	153	157,78
40,64	40,87	42,1	43,68	110,88	153,31	156,91

Com estes resultados podemos concluir que o maior ângulo de inclinação do objeto em relação ao eixo acústico aceitável é de 20º. Esse resultado parece condizer com os resultados obtidos por Borenstein e Koren (1986). Outros testes foram realizados para identificar possíveis variações nas medições com a variação no formato da superfície, estes testes procuraram obter qual o tamanho mínimo, a uma distância de 40 cm do sensor, para que um objeto seja detectado tanto tendo uma superfície plana e uma superfície circular, e com esses testes foi possível determinar que o tamanho mínimo de um objeto de superfície plana é de 4cm de largura e que com superfície circular é de diâmetro mínimo de 7 cm.

Com estes testes foi possível concluir que mesmo o sensor HC-SR04 sendo de fácil acesso e com o preço relativamente baixo, ele é um sensor de baixa precisão, e que o tempo necessário para implementar técnicas de minimização de erro nas medições seria alto e ao mesmo tempo não haveria garantia de alcançar resultados favoráveis, estes problemas poderiam ser reduzidos com o uso de sensores ultrassónicos com precisão e alcance maiores porém o custo iria aumentar muito e pelo fato da necessidade de se usar diversos sensores para ter um campo de visão aceitável para esta aplicação, os sensores ultrassónicos foram descartados como sensores principais de mapeamento.

Testes de Mapeamento Protótipo Final

Testes foram realizados em ambientes domésticos e acadêmicos e as limitações do sistema foram detectadas.

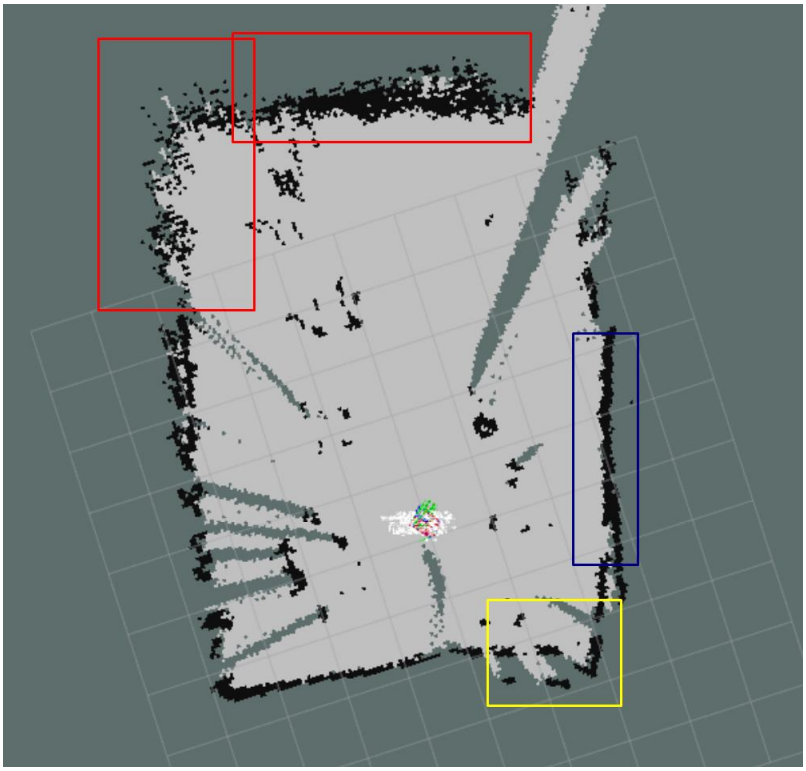


Figura 33 – Visualização do mapa criado pelo protótipo. Fonte:Autores

Na figura acima temos um mapa, resultado de uma varredura onde o robô apenas gira em cima do próprio eixo. Nesta figura foram desenhados retângulos com 3 cores diferentes para chamar a atenção de comportamentos diferentes do mapa. Os retângulos vermelhos implicam pontos no mapa onde a leitura da profundidade dos pontos foi comprometida pelo fato de tais pontos estarem além do alcance do sensor de aproximadamente 4 metros. Os pontos dentro do retângulo azul estão melhores agrupados, de forma uniforme e é possível nesse contexto interpretar estes pontos no mapa como uma parede por exemplo esse resultado é obtido quando os objetos estão sendo detectados dentro do alcance do sensor Kinect. Na área identificada pelo retângulo amarelo podemos visualizar a consequência dos erros de localização acumulados ao longo da odometria.

Foi possível também detectar uma limitação no quesito velocidade do protótipo, a odometria visual ser perder de forma muito frequente mesmo em ambientes estáticos, e por isso é necessário movimentar o robô de forma lenta, o que inviabiliza o uso deste sistema em aplicações onde são exigidas velocidade e agilidade.

5. Conclusão

Neste projeto foi possível analisar o funcionamento de um robô mapeador desde sua concepção até a construção do protótipo. Foi estudado a forma mais viável de se fazer a construção levando em consideração a falta de recursos para realizar o projeto.

Para escolher qual sensor seria utilizado para detectar o ambiente foram realizados testes com os sensores infravermelho, ultrassônico e um Microsoft Kinect de primeira versão, sendo escolhido este último por conta da vastidão de bibliotecas para tornar possível a leitura de um espaço qualquer com uma diminuição de erros razoável, além de contribuir para a localização do robô com bibliotecas com odometria visual.

A implementação inicial com o *Raspberry PI* 3 e o Arduino Uno tiveram os esperados, uma vez que os autores do projeto já tinham certa familiaridade com os produtos. Já na parte de *software*, a biblioteca libfreenetic e o *framework* RTAB-MAP tiveram resultados parcialmente satisfatórios, tendo em vista que foi possível mapear um ambiente interno. Porém, a apenas a odometria visual não é suficiente precisa para aplicações onde se requer velocidade. Para resolver esse problema seria necessário a odometria sendo feita por dados obtidos por *encoders* e magnetômetros em combinação com a odometria visual.

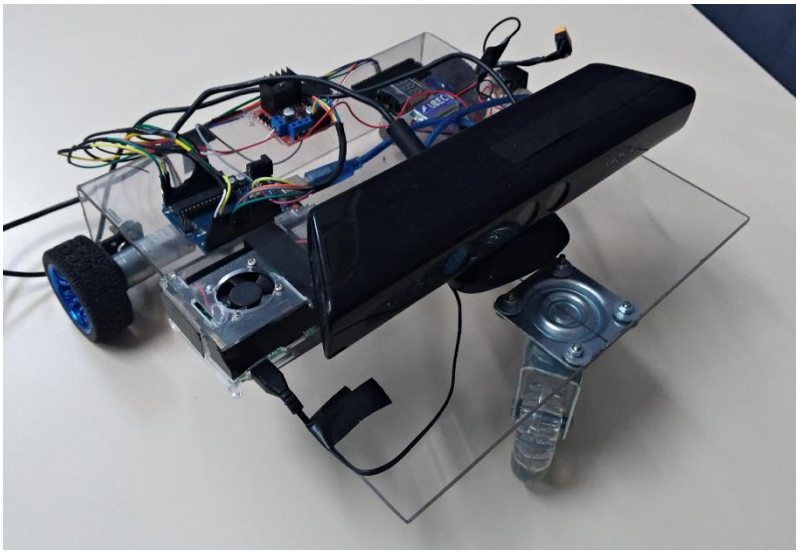


Figura 34 – Foto do protótipo finalizado. Fonte:Autores

5.1. Propostas Futuras

Para o aprimoramento deste projeto seria interessante implementar a utilização de:

- *Encoders* e magnetômetros para a odometria
- *LiDAR* por conta de sua maior velocidade e precisão
- Aprofundar o estudo sobre o *SLAM* e fazer o uso de outros tipos de algoritmos
- Definir e implementar uma aplicação de busca autônoma para o robô
- Estimular a participação de uma Equipe da Fatec nos eventos de robótica de resgate

6. Bibliografia

ACRONAME. Acroname. *Acroname*. Disponivel em: <https://acroname.com/articles/sharp-infrared-ranger-comparison>. Acesso em: 18 Junho 2018.

AULINAS, J. et al. The *SLAM* problem: a survey. *Institute of Informatics and Applications, University of Girona*, Girona, 2008.

BETKE, M.; GURVITS, L. Mobile Robot Localization Using Landmarks. *IEEE TRANSACTIONS ON ROBOTICS AND AUTOMATION*, Abril 1997.

CADENA, C. et al. Past, Present, and Future of Simultaneous Localization And Mapping: Towards the Robust-Perception Age. *IEEE Transactions on Robotics*, 2016.

DISSANAYAKE, G. A Review of Recent Developments in Simultaneous Localization and Mapping. *International Conference on Industrial and Information Systems*, 2011.

DUDEK, G.; JENKIN, M. Computational Principles of Mobile Robotics. *Cambridge University Press*, 2000.

DURRANT-WHYTE, ; BAILEY,. Simultaneous Localization: Part I. *IEEE Robotics & Automation Magazine*, 2006. 10.

NEGUS, C. *Linux Bible*. Indiana: John Wiley & Sons, Inc., 2015.

QUIGLEY, M. et al. ROS: an open-source Robot Operating System, 2009.

RICHARDSON, M.; WALLACE, S. *Make:* Getting Started With *Raspberry PI*. California: Maker Media, 2016.

STALLMAN, R. GNU Operating System, 1998. Disponivel em: <http://www.gnu.org/gnu/thegnuproject.html>. Acesso em: 22 jun. 2017.

WANG, Z.; DISSANAYAKE, G. Observability Analysis of *SLAM* Using Fisher. *Intl. Conf. on Control, Automation, Robotics and Vision*, 17–20 Dezembro 2008. 6.

YANG, Dezhong. Ultrasonic range finder. US7330398B2. 12 de fev. de 2008. [S.l.], p. 6

BORENSTEIN, Johann; KOREN, Yoram. Obstacle Avoidance with Ultrasonic Sensors. Ieee Journal Of Robotics And Automation. Pretória, p. 213-218. 7 jan. 1986.

GIONGO, Marcos et al. LiDAR: princípios e aplicações florestais. Pesquisa Florestal Brasileira, [s.l.], v. 30, n. 63, p.231-244, 28 out. 2010. Embrapa Florestas. http://dx.doi.org/10.4336/2010.pfb.30.63.231.

ZHANG, Zhengyou. Microsoft Kinect Sensor and Its Effect. Ieee Multimedia, [s.l.], v. 19, n. 2, p.4-10, fev. 2012. Institute of Electrical and Electronics Engineers (IEEE). http://dx.doi.org/10.1109/mmul.2012.24.

SANTOS, Guilherme Leal. Localização de robôs móveis autônomos utilizando fusão sensorial de odometria e visão monocular. 2010. 66 f. Dissertação (Mestrado em Automação e Sistemas; Engenharia de Computação; Telecomunicações) - Universidade Federal do Rio Grande do Norte, Natal, 2010.

RICHARDSON, Matt; WALLACE, Shawn. Getting Started with Raspberry PI. Sebastopol: Make:, 2013. 177 p. Disponível em: <https://www.blackmagicboxes.com/wp-content/uploads/2016/12/makestartingpibook.pdf>. Acesso em: 22 jun. 2018.

LABBÉ, Mathieu. Simultaneous Localization and Mapping (SLAM) with RTAB-Map. Sherbrooke: Introlab, 2015. Color. Disponível em: <https://introlab.3it.usherbrooke.ca/mediawiki-introlab/images/3/31/Labbe2015ULaval.pdf>. Acesso em: 20 jun. 2018.

Apêndice A – Código Fonte do Arduino

```
#include <stdint.h>
#include <stdlib.h>
#include <ros.h>
#include <sensor_msgs/Joy.h>
#include <std_msgs/MultiArrayLayout.h>
#include <std_msgs/MultiArrayDimension.h>
#include <std_msgs/Int32MultiArray.h>


ros::NodeHandle nh;

int encA_pin_L = 13;
int encA_pin_R = 10;
int encB_pin_L = 11;
int encB_pin_R = 12;

int inR1 = 4;
int inR2 = 7;
int inE1 = 2;
int inE2 = 3;
int LT = 0;
int RT = 0;
int PWMpinA = 5;
int PWMpinB = 6;

void joydata ( const std_msgs::Int32MultiArray& array){
  //
LT = array.data[0];
RT = array.data[1];

if(RT>0){
RT = map(RT,0,2,0,255);
 digitalWrite(2,HIGH);
 digitalWrite(3,LOW);
analogWrite(6, RT);
}
else{
   analogWrite(6, 0);
}

if(LT>0){
LT = map(LT,0,2,0,255);

 digitalWrite(7,HIGH);
 digitalWrite(4,LOW);
analogWrite(5, LT);
```

```
}
else{
    analogWrite(5, 0);
}




if (array.data[2] == 1){
 digitalWrite(3,HIGH);
 digitalWrite(2,LOW);
 analogWrite(6, 255);
}

if (array.data[3] == 1){
 digitalWrite(4,HIGH);
 digitalWrite(7,LOW);
  analogWrite(5, 255);
}

}
  ros::Subscriber<std_msgs::Int32MultiArray> sub("/joycmd", joydata);


  void setup(){
  pinMode(10,INPUT);
  pinMode(11,INPUT);
  pinMode(12,INPUT);
  pinMode(13,INPUT);
  pinMode(2,OUTPUT);
  pinMode(3,OUTPUT);
  pinMode(4,OUTPUT);
  pinMode(5,OUTPUT);
  pinMode(6,OUTPUT);
  pinMode(7,OUTPUT);
  nh.initNode();
  nh.subscribe(sub);
  }

  void loop(){
  nh.spinOnce();
  delay(1);
  }
```

Apêndice B – Código Fonte joy_to_cmd

```
#include "ros/ros.h"
#include "ros/console.h"
#include <geometry_msgs/Twist.h>
#include <sensor_msgs/Joy.h>
#include "std_msgs/MultiArrayLayout.h"
#include "std_msgs/MultiArrayDimension.h"
#include "std_msgs/Int32MultiArray.h"
#include "std_msgs/String.h"

#include <sstream>

float RT = 0;
float LT = 0;
int RRT = 0;
int RLT = 0;
int iRT = 0;
int iLT = 0;


void Callback(const sensor_msgs::Joy::ConstPtr& joy)
{
  RRT = joy->buttons[4];
  RLT = joy->buttons[5];
  RT = ((joy->axes[4])*(-1)) + 1;
  LT = ((joy->axes[5])*(-1)) + 1;

}

int main(int argc, char **argv)
{

  ros::init(argc, argv, "joy_to_cmd");

  ros::NodeHandle n;

  ros::Subscriber sub = n.subscribe<sensor_msgs::Joy>("/joy", 10, Callback);

  ros::Publisher pub = n.advertise<std_msgs::Int32MultiArray>("joycmd", 100);
  ros::Rate loop_rate(10);

  while (ros::ok())
  {
    std_msgs::Int32MultiArray msg;

    msg.data.push_back(RT);
    msg.data.push_back(LT);
```

```
    msg.data.push_back(RRT);
    msg.data.push_back(RLT);

    char data [10];

    pub.publish(msg);

    ros::spinOnce();

    loop_rate.sleep();
  }



  return 0;
}
```

Apêndice C – Arquivo .bash de inicialização da aplicação no computador

```
#!/bin/bash
cd ~/catkin_ws/
source /opt/ros/kinetic/setup.bash
source devel/setup.bash
export ROS_MASTER_URI=http://10.42.0.130:11311
export ROS_IP=10.42.0.130
xterm -hold -e "roscore" &
sleep 5
export TURTLEBOT_3D_SENSOR=kinect
xterm -hold -e "roslaunch turtlebot_bringup minimal.launch" &
sleep 5
xterm -hold -e "roslaunch rtabmap_ros demo_turtlebot_mapping.launch args:="--
delete_db_on_start" rgbd_odometry:=true" &
sleep 2
xterm -hold -e "sudo xboxdrv -i 0" &
sleep 2
xterm -hold -e "rosrun joy joy_node" &
sleep 2
xterm -hold -e "rosrun teleop_twist_joy teleop_node" &
sleep 5
xterm -hold -e "roslaunch rtabmap_ros demo_turtlebot_rviz.launch" &
sshpass -p "ubuntu" ssh -o StrictHostKeyChecking=no ubuntu@10.42.0.1 'bash -s'
< /home/guilherme/Desktop/rasp.sh

$SHELL
```

Apêndice D – Arquivo .bash de inicialização da aplicação para o Raspberry Pi

```
#!/bin/bash
cd ~/catkin_ws/
source /opt/ros/kinetic/setup.bash
source devel/setup.bash
export ROS_IP=10.42.0.1
export ROS_MASTER_URI=http://10.42.0.130:11311
roslaunch ros_arduino_python arduino.launch
```

Apêndice E – Links com os tutoriais de instalação das ferramentas de programação do projeto

- ROS Link com tutorial de instalação para Ubuntu

http://wiki.ros.org/kinetic/Installation/Ubuntu

- RTAB-MAP Link com tutorial de instalação

https://github.com/introlab/rtabmap_ros#installation

- Link de download da imagem para o Raspberry Pi com ROS e Lubuntu

https://downloads.ubiquityrobotics.com/pi.html