Universidade de Aveiro Departamento de Engenharia Mecânica
2017

Filipe Aguiar da Silva

Sistema baseado em ROS distribuído para controlo de uma plataforma skid-steering

Universidade de Aveiro Departamento de Engenharia Mecânica
2017

Filipe Aguiar da Silva

Sistema baseado em ROS distribuído para controlo de uma plataforma skid-steering

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Mecânica, realizada sob orientação científica de Vítor Manuel Ferreira dos Santos, Professor Associado do Departamento de Engenharia Mecânica da Universidade de Aveiro e de Miguel Armando Riem de Oliveira, Professor Auxiliar Convidado do Departamento de Engenharia Mecânica da Universidade de Aveiro.

Apoio financeiro dos projetos UID/EMS/00481/2013-FCT e CENTRO-01-0145-FEDER-022083

O júri / The jury

Presidente / President	**Prof. Doutor José Paulo Oliveira Santos** Professor Auxiliar da Universidade de Aveiro
Vogais / Committee	**Prof. Doutor Manuel Bernardo Salvador Cunha** Professor Auxiliar da Universidade de Aveiro (arguente)
	Prof. Doutor Vítor Manuel Ferreira dos Santos Professor Associado da Universidade de Aveiro (orientador)

Agradecimentos / Acknowledgements

Gostaria de começar por agradecer a todos os meus colegas de laboratório que me acompanharam durante todo este processo. Ao professor Vítor Santos por toda a ajuda, sugestões e conhecimento transmitido.
Não podia também deixar de fora a grande família do BEST Aveiro por me terem acolhido, sem eles não era aquilo que sou hoje.
Aos meus país e irmãos por me terem apoiado durante todo este percurso e nunca terem desistido.
Por último, a todos os colegas que tive o prazer de conhecer durante estes cinco anos de pura aprendizagem.
A todos, muitíssimo obrigado.

Palavras-chave ROS; Skid-Steering; Gazebo; Sistema distruibuído; Plataforma robótica móvel; Raspberry Pi

Resumo

Esta dissertação tem como objetivo a implementação de um sistema de ROS distribuído para o controlo de uma plataforma robótica móvel com uma configuração *skid-steering*. Para o seu desenvolvimento são usados múltiplos microcontroladores de baixo custo, o Raspberry Pi, que interligados em rede permitem a partilha de mensagens ROS e o respetivo controlo da plataforma. A par disto, foram desenvolvidas algumas funcionalidades que permitem melhor facilidade de utilização de todo o sistema, como o controlo remoto.

É ainda criado um ambiente de simulação para uma plataforma skid-steering que interage com todos os processos desenvolvidos de ROS.

Por fim, de forma a testar a solução, foi realizada uma aplicação em ambiente virtual que visa a utilização de todas as funcionalidades desenvolvidas. Esta aplicação consiste no seguimento de uma linha usando uma câmara, sendo também possível a intervenção humana no seu controlo, recorrendo ao uso de um comando *joystick*.

Keywords ROS; Skid-steering; Gazebo; Distributed system; Mobile Robotic Platform, Raspberry Pi

Abstract This dissertation aims to implement a distributed ROS system to control a mobile robotic platform with a skid-steering configuration. For its development are used multiple low-cost microcontrollers, Raspberry Pi, which networked together allow the sharing of ROS messages and the platform control. Alongside this, some features have been developed that allow better usability of the entire system, such as remote control. It also created a simulation environment for a skid-steering platform that interacts with all the processes developed in ROS. Finally, in order to test the solution, an application in virtual environment is realized that aims at the use of all the functionalities developed. This application consists in a line following application using a camera, being also possible the human intervention in its control, resorting to the use of a joystick command.

Conteúdo

Lista de Tabelas

Lista de Figuras

Lista de Acrónimos

AGV *Automated Guided Vehicle*. 1

IDE *Integrated Development Environment*. 14

LAN *Local Area Network*. 25

NFS *Network File System*. 28

PWM *Pulse-width modulation*. 7

ROS *Robot Operating System*. 1–5, 7, 8, 10, 13, 21, 28, 35, 41

STL *Standard Triangle Language*. 39

URDF *Unified Robot Description Format*. 12, 31, 33–35

Capítulo 1

Introdução

1.1 Enquadramento e motivação

O uso de robôs móveis para a execução de tarefas tem vindo a aumentar, sendo diversos os ambientes onde os podemos encontrar. Encontram-se em meio industrial, onde estão presentes os *Automated Guided Vehicle* (AGV) desempenhando tarefas como o transporte de mercadoria, ou até em ambiente agrícola onde, segundo o artigo escrito por Peter Feuilherade, a inclusão de tais veículos poderá aumentar a produtividade em cerca de 30% [1].

Dada tal importância deste tipo de robôs, a empresa Tarento Robotics [2] tem a ambição de desenvolver plataformas móveis para uso em diversas áreas tais como investigação, medicina e agricultura. Tendo como foco a agricultura, a empresa apresenta a plataforma robótica NARDO (figura 1.1) onde é pretendido que se implemente o controlo da mesma usando uma arquitetura *Robot Operating System* (ROS), deixando também a possibilidade de adaptar a plataforma a novas funcionalidades, com relativa facilidade.

Plataformas com o propósito da apresentada não teriam o mesmo impacto caso não tivessem capacidades de navegação autónoma, sendo uma das características mais desafiantes que os robôs móveis têm. Esta característica requer capacidades de percepção, localização, planeamento de trajetória e controlo de movimento, sendo necessário a utilização de vários sensores e componentes [4]. No entanto, todas estas capacidades dão origem a uma grande quantidade de dados que necessitam de ser processados de modo

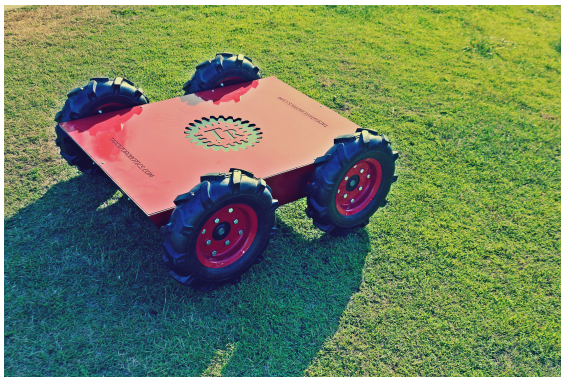


Figura 1.1: Plataforma robótica NARDO [3]

a poder-se tomar decisões, surgindo então a procura por soluções de processamento com alto desempenho.

A facilidade com que um sistema se consegue adaptar a novas condições têm cada vez mais relevância. Por exemplo, no caso da plataforma apresentada, será importante a adaptabilidade da mesma na realização de diferentes tarefas sem que seja necessário uma reconfiguração total, tanto a nível de *hardware* como a nível de *software*, tal como a adição de diversas ferramentas para a realização de tarefas relacionadas com a agricultura.

Posto isto, é importante encontrar soluções que satisfaçam ambos os critérios mencionados anteriormente. Genericamente, como arquitetura de software, propõe-se o uso de abordagens baseadas em ROS que permitam criar um sistema distribuído entre várias máquinas que estejam ligadas em rede, com abstração do hardware utilizado. Por outro lado, será importante o uso de componentes padrão para as unidades computacionais, surgindo assim o uso de Raspberry Pi's como possibilidade. Combinando estas duas soluções, espera-se criar uma uma rede distribuída de unidades computacionais que permite fazer com que cada uma se dedique especificamente ao processamento de um tipo de dados, culminando na realização de uma tarefa final.

No final, importa também fazer o teste da solução desenvolvida e para isto propõe-se o controlo da plataforma móvel em ambiente virtual. Para tal demonstração é importante que se proceda à realização de uma aplicação que demonstre várias unidades de processamento a desenvolverem tarefas diferentes.

1.2 Objetivos

Os objetivos pretendidos com a presente dissertação são os seguintes:

- Criação de uma rede de unidades computacionais para a execução de ROS numa abordagem distribuída;
- Instalação do ROS e desenvolvimento do software necessário à interligação da plataforma com sistemas periféricos;
- Criação de um ambiente de simulação para uma plataforma skid-steering;
- Criação de uma aplicação de demonstração.

1.3 Estado da arte

Seguidamente serão apresentados alguns projetos e plataformas da mesma configuração que aquela em estudo neste trabalho, assim como algumas questões de controlo deste tipo de plataformas. Também são abordados alguns exemplos onde o ROS é usado para o controlo de sistemas distribuídos.

1.3.1 Plataformas *Skid-steering*

Atualmente no mercado existem algumas plataformas com a configuração de *skid-steering*, sendo maioritariamente usadas para operar em ambientes agrícolas, exploração de minas, militar, operações de salvamento, ou em ambientes com solo irregular. Alguns dos exemplos são as plataformas desenvolvidas pela Clearpath [5] sendo criadas essencialmente

para ambientes terrestres, como é o caso da plataforma Husky (figura 1.2). Esta plataforma usa uma arquitetura ROS podendo ser adquirida com diversos tipos de unidades computacionais como por exemplo minicomputadores como os Intel NUC ou então uma *motherboard* mini ITX.

Figura 1.2: Plataforma Husky da Clearpath [6]

CARLoS

Um outro exemplo é o caso da plataforma móvel robótica Guardian da Robotnik, tendo também uma configuração *skid-steering* é criada especialmente para a utilização em projetos relacionados com segurança, inspeção, e investigação [7].Para o controlo, a plataforma usa uma arquitetura ROS, enquanto que como unidade computacional é usado uma *motherboard* com um processador Intel. Uma aplicação onde esta plataforma foi usada é o projeto CARLoS (Figura 1.3) onde o objetivo é desenvolver um robô móvel capaz de executar alguns processos de soldadura durante a construção de navios. Para tal é acoplado um manipulador que contém na sua extremidade a ferramenta de soldadura. A plataforma tem então de ter algumas capacidades tais como a navegação autónoma dentro dos blocos do navio [8].

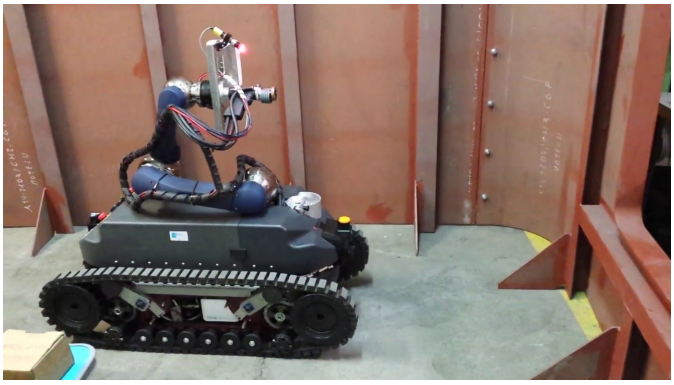


Figura 1.3: Robô CARLoS [8]

VINBOT

Ainda usando uma plataforma da Robotnik, mas com a versão Summit XL na configuração *skid-steering*, foi desenvolvido um robô todo-o-terreno autónomo, denominado

por VINBOT (figura 1.4), tendo como objetivo realizar a análise do estado de vinhas e avaliar o seu rendimento. Para tal são aplicados um conjunto de sensores capazes de captar e analisar, imagens assim como dados 3D. A recolha dos dados é enviada para os viticultores ajudando assim num aumento significativo da produção de vinho. Todo este sistema opera sobre uma arquitetura ROS.

Figura 1.4: Robô VINBOT[9]

1.3.2 Questões de controlo

O uso de plataformas do tipo *skid-steering* tem tido uma utilização crescente principalmente para terrenos arenosos. No entanto, o tipo de cinemática usado traz alguns desafios devido à interação entre as rodas e o chão. O problema mais comum no controlo deste tipo de robôs é o facto de necessitarem que ocorra derrapagem para que possam alterar a sua trajetória, no entanto, tal procedimento é difícil de descrever pois depende de diversos fatores como a ficção entre o chão e as rodas[10].

O funcionamento deste tipo de plataforma, representada na figura 1.5 é realizado pelo acionamento em simultâneo de ambas as rodas laterais. Para a plataforma realizar uma trajetória linear ambos os pares de rodas têm de rodar com a mesma velocidade, enquanto que no caso de uma trajetória curva as rodas têm de rodar com velocidades diferentes.

Em [10] é apresentado uma solução para as equações da cinemática deste tipo de robô (equação 1.1), na quais w_L e w_R representam as velocidades angulares das rodas do lado esquerdo e direito respetivamente, v_x e w a velocidade linear e angular da plataforma respetivamente. O valor das constantes r e c têm que ser calculadas com base em ensaios experimentais pois dependem da interação com o meio.

$$\begin{bmatrix} v_x \\ w \end{bmatrix} = r \begin{bmatrix} \frac{w_L + w_R}{2} \\ \frac{-w_L + w_R}{2c} \end{bmatrix} \tag{1.1}$$

De forma aperfeiçoar as equações de cinemática, são diversas as abordagens utilizadas. Um estudo realizado na universidade de Beihang [11] propõe que para se conseguir chegar a um modelo cinemático se recorra a um laser (Sick LMS400) para traçar a trajetória percorrida, de forma a obter os valores para as constantes de forma a caracterizar da melhor forma o modelo cinemático. Para essa experiência é usada a plataforma móvel robótica Pioneer P3-AT (figura 1.6) que é desenvolvida pela empresa Adept e muito usada

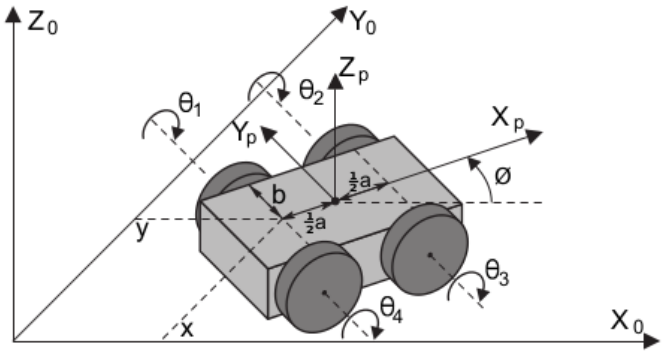


Figura 1.5: Esquema de uma plataforma móvel *skid-steering* [10]

Figura 1.6: Plataforma Pioneer P3-AT[12]

para investigação utilizando um computador a bordo para comunicar com o *hardware* usando uma biblioteca própria da marca.

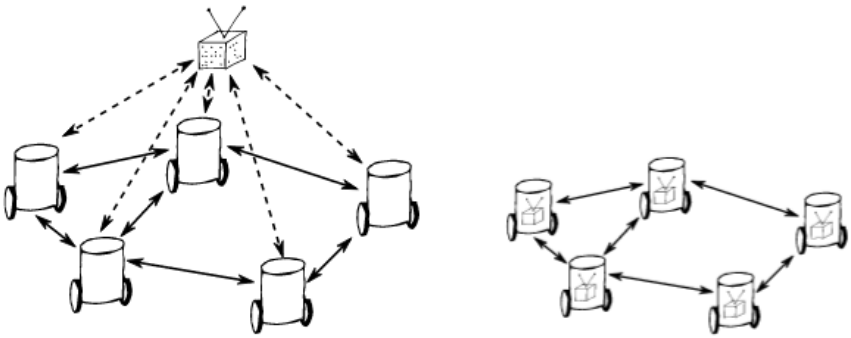
1.3.3 Sistemas distribuídos usando ROS

De uma forma breve, o ROS consiste numa *framework* usada no desenvolvimento de robô que permite ter vários processos a correr ao mesmo tempo denominados por nodos, podendo estes ser desenvolvidos em diversas linguagens e plataformas, permitindo uma abstração do *hardware*. Para realizar a comunicação entre todos eles, existe o nodo *roscore* que gerência toda a passagem de informação. Os nodos ROS podem então ser distribuídos por diversas máquinas, o que permite ter sistemas distribuídos.

Uma aplicação que pode ser realizada é o controlo de vários robôs tendo por base uma única unidade de controlo onde é executado o *roscore* (figura 1.7a) que terá como tarefa contolar os diverssos robôs remotamente e fazer o gerenciamento da comunicação entre os diversos robôs, tendo estes apenas de executar nodos específicos [13]. No trabalho descrito em [14] foi utilizada esta abordagem de onde é feita a simulação de vários robôs para realizarem uma função de mapeamento e de exploração.

Muito semelhante ao exemplo anterior, outra possibilidade é existir um robô que contenha uma unidade computacional que atue como Master onde seja executado o *roscore*, e haver outros robôs que apenas executam os seus nodos específicos comunicando sempre com o Master.

Porém, também existe a possibilidade de se ter uma rede distribuída de ROS onde cada unidade computacional corre o seu *roscore* havendo uma passagem dos tópicos através da utilização de *packages* especiais (figura 1.7b), como por exemplo o `wifi_comm`



(a) (b)

Figura 1.7: (a) Controlo de vários robôs usando uma unidade centralizada executando o ROS Master (b) Controlo de vários robôs cada um correndo o ROS Master [13]

[15]. Uma aplicação desta abordagem pode ser consultada no trabalho de [16], onde o objetivo é fazer o controlo de diversos robôs de modo a que todos executem uma tarefa de mapeamento.

1.4 Estrutura da dissertação

A presente dissertação é constituída pelo presente capítulo e mais seis:

- **Estrutura experimental e ferramentas** - Descrição do *hardware* e *software* utilizado ao longo do trabalho;
- **Controlo da plataforma** - São abordados os procedimentos executados para a realização do controlo da plataforma;
- **Unidade computacional** - É descrito todo o processo para a obtenção da solução proposta para a unidade computacional de todo o sistema;
- **Ambiente de simulação** - Descrição do processo e resultados obtidos para o desenvolvimento do ambiente de simulação;
- **Aplicação ilustrativa** - Descrição da aplicação desenvolvida e seus resultados;
- **Conclusão e trabalho futuro** - São apresentadas as conclusões do trabalho e as propostas de trabalho futuro.

Capítulo 2

Infraestrutura experimental e ferramentas

Neste capítulo são descritos todos os *softwares*, equipamentos e bibliotecas usadas ano desenvolvimento do trabalho, nomeadamente a plataforma, o ROS, o simulador Gazebo, e unidades de processamento e controlo.

2.1 Plataforma robótica móvel

Como referido no capítulo 1, o trabalho desenvolvido terá como objeto de estudo uma plataforma robótica móvel denominada por NARDO, representada na figura 1.1.

A plataforma possui um motor dedicado ao acionamento de cada par de rodas laterais da plataforma. Os motores utilizados são motores de escovas DC de 24V tendo já incorporada uma caixa redutora (figura 2.1).

Para fazer o acionamento dos motores são utilizados os *drivers* de potência YC-MD1301-2.2 tendo uma corrente máxima de 10A DC. Estas unidades permitem a entrada de um sinal *Pulse-width modulation* (PWM) de 5V e de 35VDC para a alimentação dos motores. Permitem ainda fazer a troca de sentido de direção com recurso a uma ponte H.

Como alimentação de todo o sistema, são usadas duas baterias de chumbo de 12V ligadas em série dando assim uma tensão de saída de 24V permitindo a alimentação dos

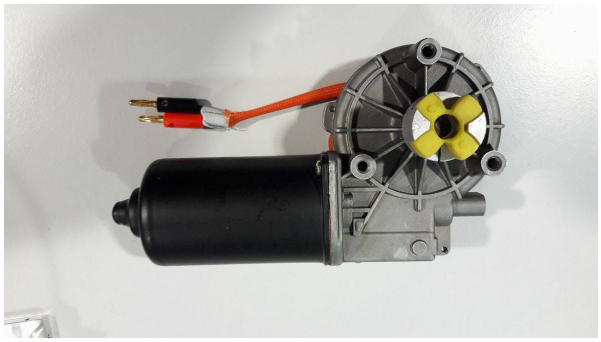


Figura 2.1: Motor utilizado na plataforma

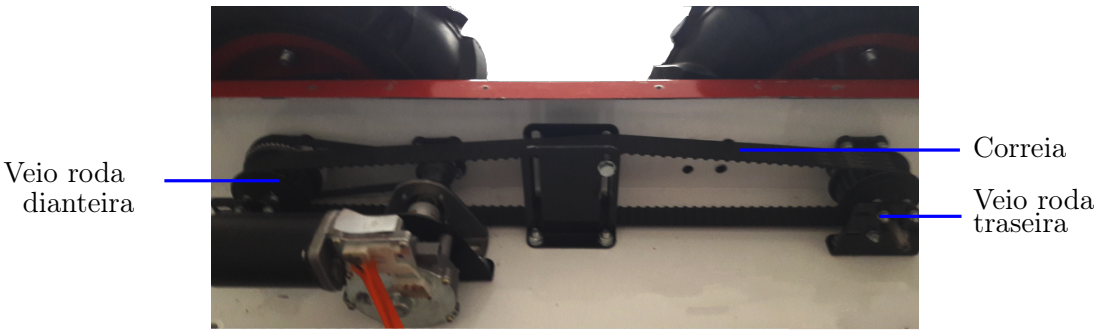


Figura 2.2: Sistema de transmição utilizado na plataforma

motores. Existe ainda um conversor de tensão que converte os 24V das baterias em 20V com o objetivo de fazer a alimentação para os circuitos de controlo.

O controlo remoto da plataforma é feito com um comando de *PlayStation 3* ligado via bluetooth a um controlador Arduino UNO con recurso a um "shield"USB e um adaptador bluetooth.

Como meio de transmissão entre os motores e as rodas é usado um sistema de correias transmitindo a mesma potência para ambas as rodas (figura 2.2). Na figura 2.3 pode ser observada uma ilustração com todos os componentes mencionados anteriormente.

A plataforma tem 800 mm de largura, 1200 mm de comprimento e 350 mm de altura. As suas rodas possuem um rasto com grande relevo, o que torna a plataforma apropriada para realizar tarefas em terrenos arenosos.

2.2 ROS - Robot Operating System

Originalmente criado em 2007 o ROS é um ambiente de desenvolvimento para a realização de software para robôs, oferecendo funcionalidades idênticas a um sistema operativo. O ROS oferece diversas vantagens tais como:

- Grande variedade de ferramentas: juntamente com o ROS temos ao nosso dispor diversas ferramentas que nos permitem fazer *debugging*, visualização e simulação das aplicações criadas;
- Suporte para diversos sensores e atuadores: temos disponível um vasto número de controladores para os mais diversos tipos de sensores tais como lasers e câmaras;
- Modularidade: devido ao facto de em ROS se usarem vários processos dedicados a uma operação, no caso de alguma parte do robô falhar esta não compromete a funcionalidade das restantes caso se use redundância.

No geral, pode-se dizer que o ROS está subdividido em três grupos, o *file system level*, o *Computation Graph level* e o *Community level* (figura 2.4).

Filesystem level

Algo que distingue o ROS de outras soluções é o seu sistema de ficheiros. Tal como um sistema operativo, os ficheiros do ROS também estão guardados em disco com uma configuração específica segundo o diagrama da figura 2.5.

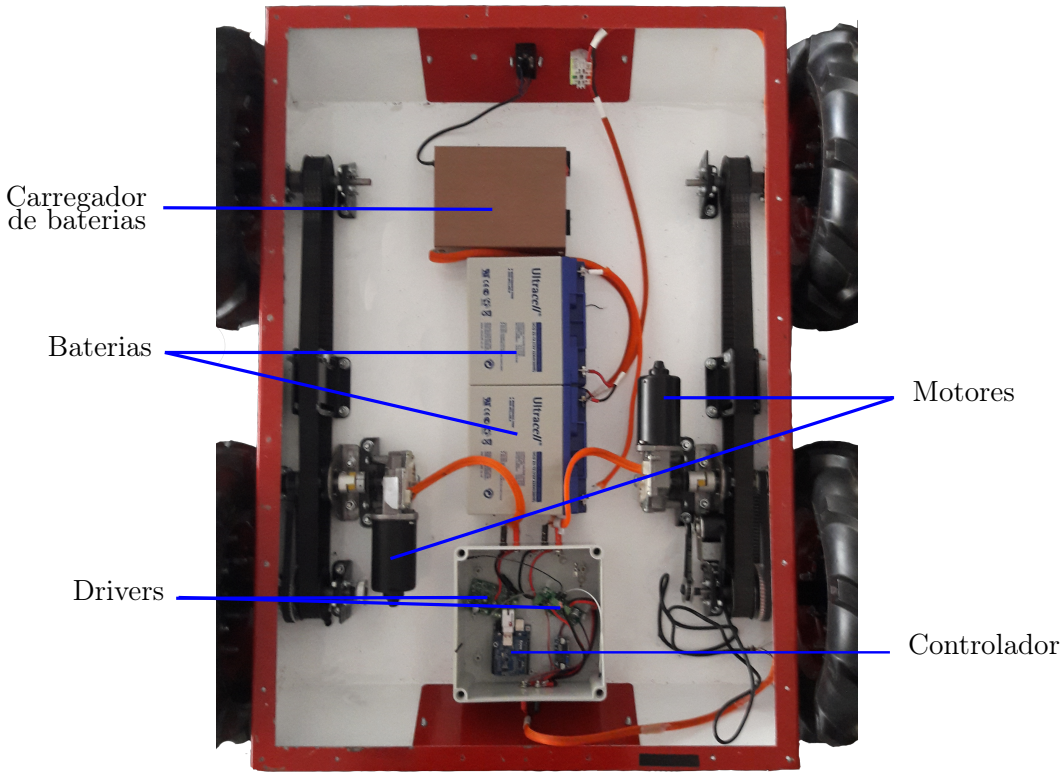


Figura 2.3: Ilustração dos componentes da plataforma

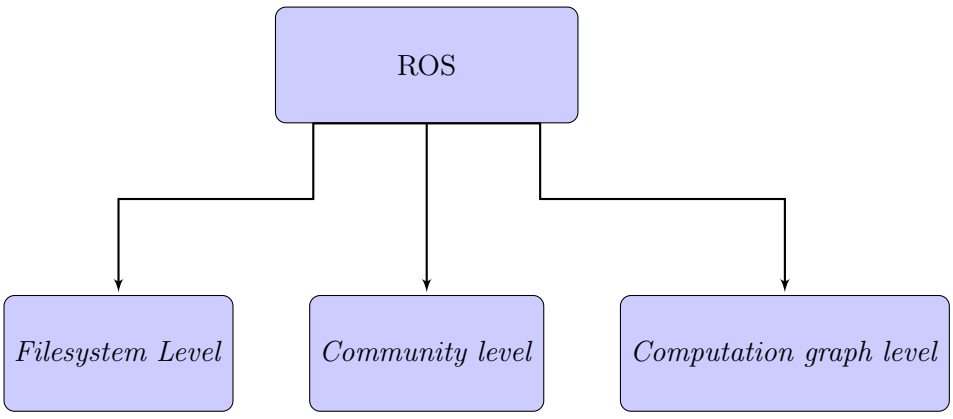


Figura 2.4: Estrutura geral do ROS

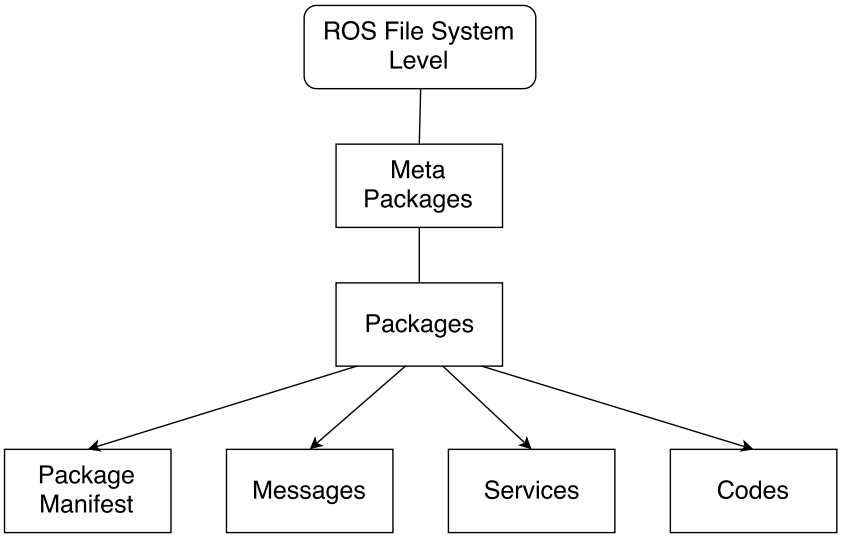


Figura 2.5: Diagrama com a estrutura de ficheiros do ROS

Segue uma explicação de cada uma das partes representadas:

- **Packages**: Os ROS *packages* são a unidade mais básica da arquitetura ROS. Aqui estão contidos todos os processos, bibliotecas, ficheiros de configuração, entre outros.

- **Package manifest** - Trata-se de um ficheiro que contém a informação acerca do *package* tal como o autor, licenças, dependências e regras de compilação.

- **Meta packages** - É usado para agrupar *packages* para um determinado propósito. Um exemplo de um *Meta package* é o Navigation Stack.

- **Meta packages manifest** - Semelhante ao *package manifest*, inclui as dependências das *packages* que lhe estão contidos.

- **Messages (.msg)** - ROS messages são o tipo de informação que o ROS envia entre os vários processos. Pode-se definir um tipo de mensagem dentro de uma pasta denominada de `msg` dentro de um *package*.

- **Services (.srv)** - ROS services é uma espécie de interação pedido/resposta entre processos. Os tipos de mensagem de pedido e resposta são definidos dentro de uma pasta denominada de `srv` dentro de um *package*.

Computation graph level

O *computation graph level* é o grupo onde o ROS processa todos os dados dentro da sua rede. As principais partes deste nível são os nodos, mensagens, tópicos, serviços, *bags*, o *Master* e *Parameter Server*. A figura 2.6 apresenta um esquema com esta informação.

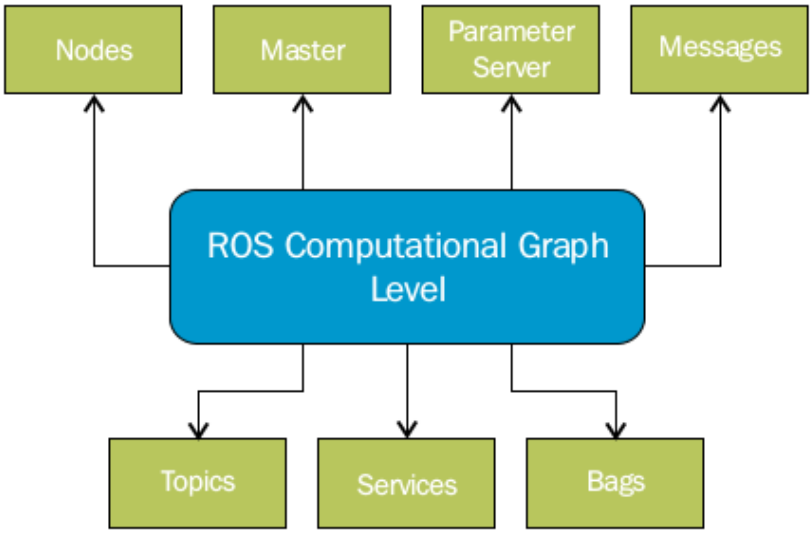


Figura 2.6: Diagrama referente às diferentes partes do *Computation graph level* [17]

Nos seguintes pontos será explicado qual é a funcionalidade de cada parte:

- **Nodos** - Os nodos são os processos que realizam o processamento. Estes estão escritos usando bibliotecas específicas do ROS tais como `roscpp` e `rospy`. Um robô pode ter vários nodos onde cada um é responsável pelo processamento de uma determinada tarefa. Para comunicarem entre si, os nodos podem usar métodos de comunicação do ROS como a publicação e subscrição de tópicos, descrito seguidamente.
- ***Master*** - O ROS Master é responsável pelo registo e administração dos nodos existentes. Assim os nodos podem trocar mensagens, reconhecerem-se entre si, ou invocarem algum serviço. Num sistema distribuído, o ROS Master deverá ser ativado inicialmente e posteriormente os restantes nodos podem comunicar.
- ***Parameter Server*** - Permite que todos os dados estejam guardados num sistema central, possibilitando ao utilizador alterar e configurar os parâmetros dos nodos enquanto estes estão em execução.
- **Mensagens** - São o método de comunicação entre os nodos, estas possuem uma estrutura de dados bem definida e conhecida entre cada nodo ROS. Existem algumas estruturas padrão de mensagens, no entanto é possível desenvolver outras.
- **Tópicos** - É o nome que é dado ao canal por onde são enviadas as mensagens. Quando um nodo publica uma mensagem através de um tópico podemos dizer que o nodo publicou um tópico, o mesmo acontece no caso da receção de uma mensagem onde dizemos que o nodo subscreve o tópico. É possível subscrever tópicos que não estão a ser publicados assim como um tópico pode ser acedido por vários nodos.
- **Serviços** - Por vezes o uso de mensagens não é suficiente sendo necessário existir uma interação pedido/resposta entre processos. Para isto, os serviços permitem que um nodo cliente envie um serviço que posteriormente um nodo servidor irá processar enviando no fim uma resposta informando da conclusão da tarefa.

- ***Bags*** - É um formato de ficheiro que armazena todas as mensagens de tópicos e de serviços, dando a possibilidade de play back de um período de tempo definido.

Community level

O Community level consiste em todas as distribuições ROS, repositórios, a Wiki do ROS e o ROS Answers que permite a todos os investigadores, desenvolvedores e indústrias a partilharem os seus programas, ideias e conhecimento no sentido de melhorar as comunidades de robótica.

Por si só, o ROS não faz a manutenção dos seus repositórios para os pacotes de ROS, no entanto os utilizadores e desenvolvedores são encorajados a hospedar os seus próprios repositórios para os pacotes que eles próprios criaram ou usaram. Os desenvolvedores têm a possibilidade de utilizarem a ROS Wiki para publicitar e criarem tutoriais que demonstram as usabilidade dos seus próprios pacotes. ROS answers é um fórum que ajuda a responder a questões relacionadas com ROS que os seus utilizadores possuam[17].

2.3 URDF

Uma das formas de definirmos um robô, os seus sensores e o seu ambiente de trabalho envolvente, é criar o seu *Unified Robot Description Format* (URDF). Aqui pode-se descrever robôs que tenham uma estrutura do estilo de árvore, ou seja, que é constituído por elos que terão de ser rígidos. Todos estes elos são então conectados através de juntas.

O URDF é composto usando *tags* especiais de XML que posteriormente serão processadas. Exemplos destas *tags* são a `<link>` e `<joint>`, onde na primeira é definido um elo incorporando características como a forma e massa, na segunda podemos definir as juntas indicando o seu tipo e os elos que lhe pertencem.

De forma a simplificar a escrita deste tipo de ficheiro é possível usar xacros que são macros de XML. Ao utilizar xacros pode-se tornar o ficheiro URDF mais curto, aumentar a sua facilidade de leitura, e pode ser usado para construir descrições de robôs complexas. Usando algumas ferramentas do ROS, é possível converter um xacro em URDF.

Na listagem 2.1 encontra-se um exemplo do código URDF de um robô que contem dois elos e uma junta a interligados, estando ilustrado o seu resultado final na figura 2.7

Listagem 2.1: Exemplo da definição de um robô com dois elos e uma junta no formato URDF

```
<?xml version="1.0"?>
    <robot name="robot">

        <link name="first_link">
            <visual>
                <geometry>
                    <cylinder length="0.4" radius="0.04"/>
                </geometry>
                <origin rpy="0 0 0" xyz="0 0 0.09"/>
                <material name="red">
                    <color rgba="0 0 1 1"/>
                </material>
```

```
        </visual>
    </link>
    <joint name="joint" type="revolute">
        <parent link="first_link"/>
        <child link="second_link"/>
        <origin xyz="0 0 0.2"/>
        <axis xyz="0 1 0" />
    </joint>
    <link name="second_link">
        <visual>
            <geometry>
                <cylinder length="0.4" radius="0.04"/>
            </geometry>
            <origin rpy="0 0 0" xyz="0 0 0"/>
            <material name="red">
                <color rgba="1 0 0 1"/>
            </material>
        </visual>
    </link>
</robot>
```

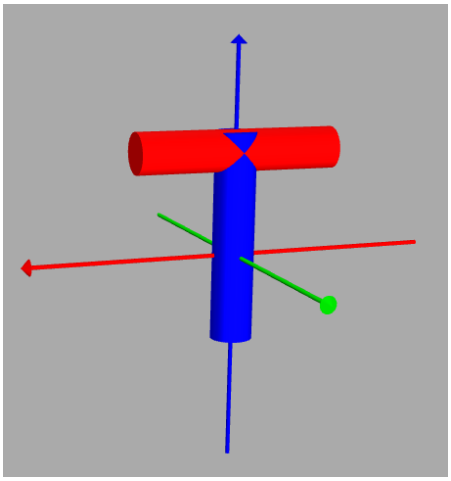


Figura 2.7: Visualização do robô resultante do código presente na listagem 2.1

2.4 Gazebo

Neste trabalho será usado um simulador de robôs denominado por Gazebo onde se pode testar robôs complexos, sensores e uma variedade de objetos. Este simulador possui modelos de simulação para alguns robôs comuns que poderão ser usados sem a necessidade de serem criados desde raiz [18].

O ROS já possui vários *packages* que permitem uma fácil comunicação com o Gazebo, tal como o `gazebo_msgs` que contem estruturas de mensagens e serviços que permitem interagir como Gazebo a partir do ROS. Para além desta comunicação, o Gazebo permite

também a adição de vários sensores aos robôs tais com sensor de visão. Devido à sua vasta lista de *plugins* pode simular muitos dos sensores existentes no mercado atualmente. Na figura 2.8 pode observar-se uma imagem ilustrativa da interface gráfica [19].

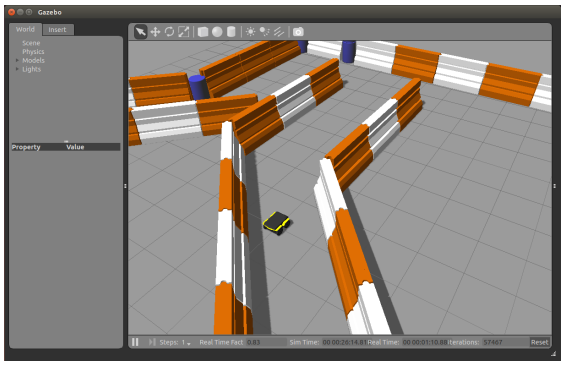


Figura 2.8: Interface gráfica do Gazebo [20]

Uma ferramenta que o Gazebo possui é o *Building Editor* que permite criar ambientes de simulação, onde é possível criar edifícios, por exemplo.

2.5 OpenCV

OpenCV (*Open Source Computer Vision Library*) é uma biblioteca usada para processamento de imagem, podendo ser utilizada livremente tanto em meio académico ou comercial. Para a sua utilização podem ser usadas diversas linguagens de programação tais como C/C++, Python e Java.

No caso do trabalho realizado, a biblioteca é utilizada na realização da aplicação para a demonstração do trabalho proposto, sendo usado o processamento de imagem para a deteção da linha.

2.6 Controladores Arduino

Uma das unidades de processamento usadas neste trabalho é o microcontrolador Arduino. O Arduino é uma plataforma *open-source* e fácil de usar que permite a aquisição de sinais externos e também a ativação de saídas digitais. Devido a toda esta versatilidade, o sistema Arduino é largamente usado em muitos projetos tais como a construção de robôs.

O tipo de programação utilizada é bastante simplificada comparando com a programação de microcontroladores PIC, pois o Arduino já possui bibliotecas internas que ajudam na configuração dos seus pinos. Para realizar a programação do microcontrolador existe um *Integrated Development Environment* (IDE) de fácil utilização onde já é incluído o compilador.

Neste momento existem diversas plataformas com configurações diferentes, no entanto neste trabalho apenas foram usadas as versões UNO e Nano representados nas figuras 2.9 e 2.10 respetivamente. Na tabela 2.1 podem ser consultadas as especificações de cada uma destas plataformas[21].

Figura 2.9: Controlador Arduino Uno[22]

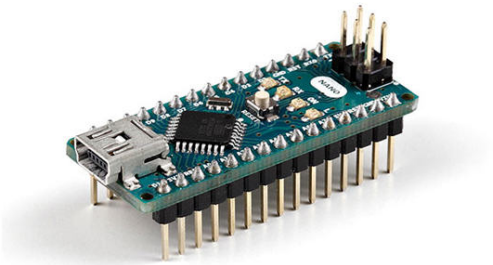


Figura 2.10: Controlador Arduino Nano[23]

Versão	Arduino Uno	Arduino Nano
Microcontrolador	ATmega328	ATmega328
Tensão de trabalho	5V	5V
Flash Memory	32 KB	32 KB
Clock Speed	16 MHz	16 MHz
Pinos saída	14	14
Pinos pwm	4	4
Pinos analógicos	6	8
Tamanho	68.6 mm × 53.4 mm	18 mm × 45 mm
Massa	25 g	7 g

Tabela 2.1: Especificações do arduino Uno e Nano [22, 23]

2.7 Raspberry Pi

Raspberry Pi é um computador com a dimensão de um cartão de crédito e de baixo custo. Foi originalmente criado para melhorar as competências de programação e *hardware* para

crianças e jovens. No entanto devido ao seu tamanho e baixo custo, começou a ser utilizado para outros fins tais como projetos eletrónicos e domótica.

Com a sua capacidade de processamento este sistema consegue ter as capacidades de um computador, desempenhando todas as funcionalidades implicando um baixo consumo energético[24].

Atualmente existem no mercado diversas versões desta placa onde difere o seu formato assim como as suas especificações. Neste trabalho foram usadas a versão Raspberry 2 e o Raspberry Pi 3 (figura 2.11). Na tabela 2.2 são apresentadas algumas das suas características.

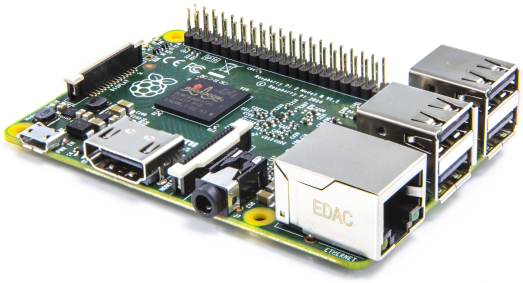


Figura 2.11: Unidade computacional Raspberry Pi 3[25]

	Raspberry Pi 2	**Raspberry Pi 3**
Processador	Cortex-A7	Cortex-A53 64-bit
Número de núcleos	4	4
CPU Clock	900MHz	1.2GHz
RAM	1 GB	1 GB
GPIO pinos	40	40
Porta HDMI	Sim	Sim
Porta Ethernet	Sim	Sim
Portas USB	4	4
SPI	Sim	Sim
I2C	Sim	Sim
Wi-Fi	Não	Sim
Bluetooth	Não	Sim

Tabela 2.2: Lista de específicações dos Raspberry Pi 2 e 3[26]

Capítulo 3

Controlo da plataforma

Neste capítulo abordam-se todos os procedimentos tomados para o controlo da plataforma, mais especificamente a elaboração do controlo em malha fechada dos motores e a comunicação entre a unidade computacional central e os microcontroladores.

3.1 Interveções na plataforma

Para se realizar o controlo da plataforma foi necessário fazer algumas alterações à mesma, mais especificamente a implementação de *encoders* de forma a ser possível ter *feedback* em relação ao estado das rodas, podendo assim ser feito um controlo em malha fechada.

Os *encoders* disponíveis, fornecidos pela empresa que propôs o trabalho, são do tipo incremental e possuem as características apresentadas na tabela 3.1.

Tensão mínima de entrada	5V
Tensão máxima de entrada	24V
Número de canais	2
Rotação máxima	5000 RPM
Pulsos por rotação	100
Diâmetro do veio	6 mm
Diâmetro exterior	40 mm

Tabela 3.1: Características dos *encoders* usados

Para o acoplamento entre os motores e *encoders* é usado um sistema com uma correia que faz a transmissão entre o veio do *encoder* e o veio da roda, existindo uma polia em cada um (figura 3.1). Nesta transmissão acontece uma desmultiplicação de 1:3, multiplicando em três vezes o número de rotações que o *encoder* lê em relação às rotações da roda, aumentando assim a precisão de leitura.

O suporte do encoder é realizado através de uma chapa que é fixada no suporte do veio do motor, tendo também a furação referente ao encaixe do *encoder*.

No total, foram instalados dois *encoders* na plataforma ambos nas rodas dianteiras da plataforma.

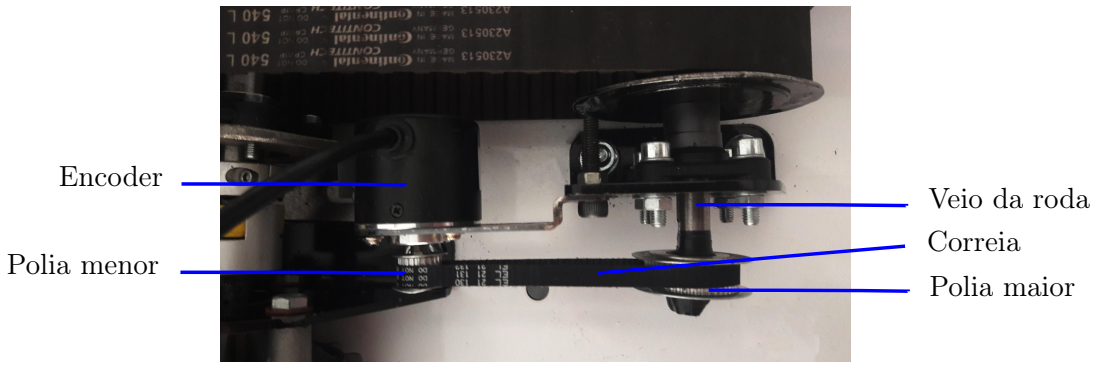


Figura 3.1: Montagem final dos *encoders*

3.2 Leitura de posição

Para a realização da leitura dos *encoders* a solução encontrada baseia-se em colocar um controlador Arduino dedicado para cada *encoder*, sendo cada um destes denominado por Slave. O valor lido por cada encoder é posteriormente comunicado para outro controlador Arduino, Master, utilizando uma comunicação I2C. O Arduino Master é então responsável por fazer o processamento dos dados recebidos. Na figura 3.2 encontra-se um esquema da configuração final utilizada.

A opção pelo uso de controladores Arduino para a leitura dos *encoders* deveu-se principalmente ao facto de estes estarem disponíveis no laboratório evitando assim o gasto de recursos em outras soluções tais como circuitos contadores. Neste trabalho, devido à disponibilidade do *hardware*, foram utilizados um controlador Arduino UNO como Slave Right e Master, enquanto que como Slave Right foi usado um Arduino Nano.

Em relação à leitura dos sinais do *encoder* tirou-se proveito do facto de estarem disponíveis dois pinos de *interrupt* em cada Arduino, podendo-se assim utilizar as variações de ambos os canais para provocar interrupções no Arduino, aumentando a resolução de leitura em quatro vezes pois, conforme demonstrado na figura 3.3, existem quatro combinações diferentes aquando da leitura dos sinais do *encoder*.

Assim, para realizar a leitura do encoder com o Arduino foram utilizados os pinos de *interrupt* do Arduino, pinos 2 e 3. Utilizando a função do Arduino `attachInterrupt()` com a opção `CHANGE` pode-se definir uma função que será executada sempre que o sinal recebido num dos pinos altere. Esta função é responsável por decrementar ou incrementar o valor correspondente aos pulsos do encoder tendo em conta o seu estado e o do outro pino, conforme ilustrado na figura 3.4.

Ainda no Arduino Slave tem de se efetuar a comunicação I2C, para tal recorreu-se ao uso a biblioteca `Wire.h` [28]. Aqui é necessário definir um endereço fixo para o Slave de modo a que o Master possa fazer pedidos. Tem de se definir uma função que será executada quando é feito um pedido, neste caso para o envio do número de pulsos. Para efetuar um melhor tratamento da informação, a mensagem enviada por I2C tem o formato `<número_de_pulsos>/`, sendo do tipo `String`.

No Arduino Master, é então necessário também utilizar a biblioteca `Wire.h` de forma a que seja possível fazer a comunicação I2C com os Arduinos Slave. Para a obtenção das mensagens por parte dos controladores Slave, o Arduino Master tem que fazer um pedido onde é indicado o endereço do Arduino de onde se quer obter a leitura do *encoder*, assim

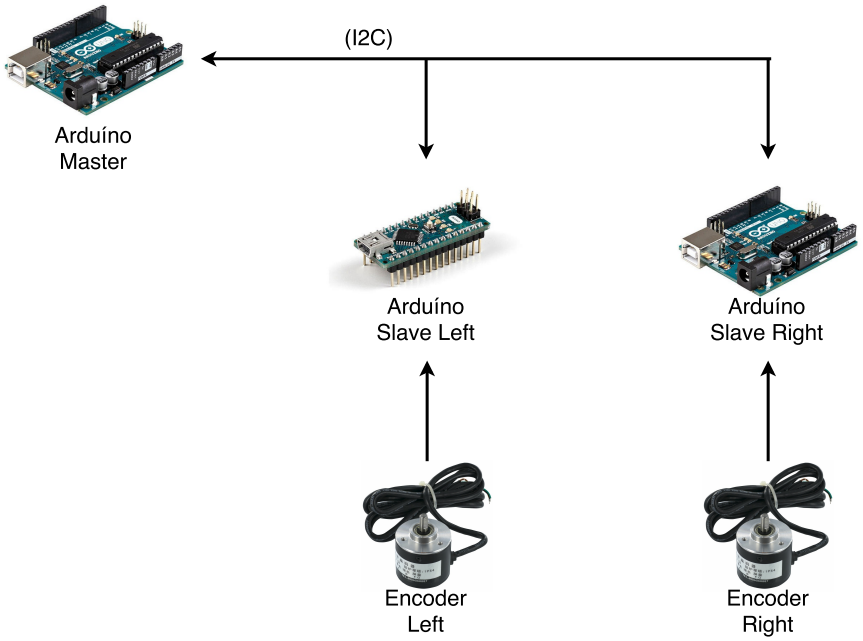


Figura 3.2: Diagrama com a rede de Arduinos utilizada para a leitura dos *encoders*

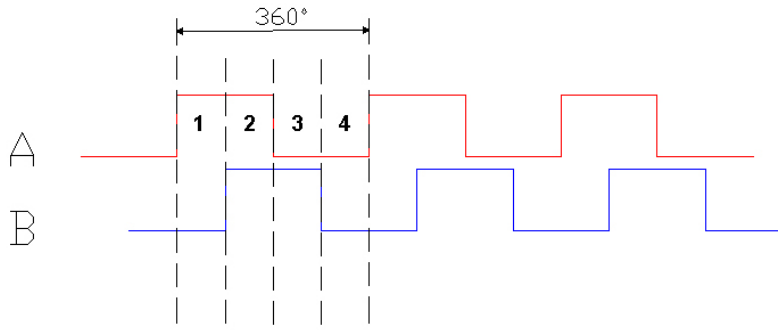


Figura 3.3: Diagrama temporal dos sinais recebidos do *encoder*[27]

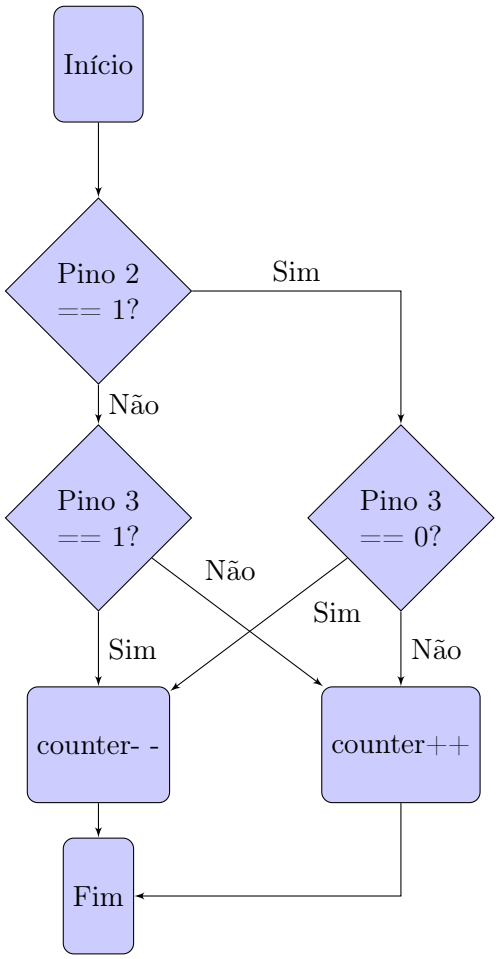


Figura 3.4: Diagrama referente à função executada sempre que é ativada a interrupção 1 num controlador Slave

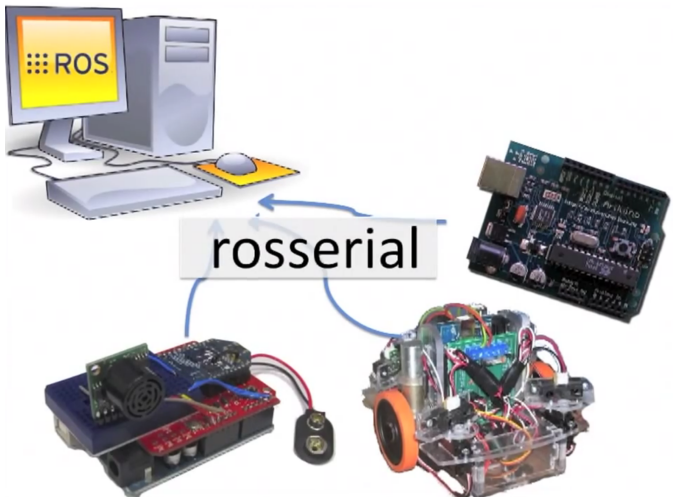


Figura 3.5: Comunicação comunicação entre diversos dispositivos usando rosserial [29]

como o tamanho da mensagem a ler.

Após a receção da mensagem, o Master tem que fazer o seu tratamento, eliminando a "/"e a conversão do valor lido para inteiro.

3.3 Comunicação com microcontrolador

Para realizar a comunicação entre a unidade computacional e o Arduino foi utilizado um *package* de ROS denominado por Rosserial. Este *package* baseia-se numa série de protocolos de comunicação que permitem fazer a comunicação entre o ROS e outros dispositivos através de porta série, *socket* ou vise-versa (figura 3.5). O protocolo Rosserial pode converter entre mensagens e serviços ROS para tipos de mensagens para outros tipos que sejam entendidos pelos dispositivos a usar [29]. Para complementar, existe uma biblioteca para Arduino, `rosserial_arduino`, que permite realizar a leitura das mensagens ROS, podendo assim subscrever e publicar tópicos ROS.

No caso da aplicação em questão, na unidade computacional é publicado um tópico onde a sua mensagem contem a informação da velocidade desejada em cada roda da plataforma. Após o Arduino detetar a receção de uma mensagem é então ativada uma *callback* no microcontrolador que atualizará o valor das velocidades desejadas nas respetivas variáveis.

Por outro lado, de forma a existir um controlo contínuo e uma monitorização por parte da unidade computacional, o Arduino também realiza a publicação de um tópico que contem os dados referentes ao número de pulsos lidos em cada roda.

3.4 Controlador PID

De modo a fazer um controlo em malha fechada desenvolveu-se um controlador PID utilizando um microcontrolador Arduino. Este tipo de controlador baseia-se no cálculo de um erro da posição, através do valor lido momentaneamente e o desejado. Após o erro ser processado o controlador calcula um valor de saída base com base nas constantes de

ganho Proporcional (P), Integral (I) e Derivativo (D).

No trabalho desenvolvido, este tipo de controlador foi implementado no Arduino Master e para isto foi desenvolvida uma *class* que permite a sua implementação. Esta *class* para além de ter o obrigatório construtor que aceita como parâmetros as constantes do PID, tem os seguinte métodos:

- `setSetPoint(double value)` - tem como função fazer a definição do valor a atingir com o PID;
- `addNewSample(double value)` - adiciona uma nova amostra que será usada para realizar o cálculo do novo erro;
- `compute()` - este é o método principal da *class*, que tem como objetivo calcular o valor de saída do controlador, sendo necessário aqui calcular o erro entre a última leitura e o valor pretendido;

Listagem 3.1: *Class* desenvolvida para a implementação do controlador PID

```
class PID{
public:
  double error, sample, lastSample, kP, kI, kD, P, I, D, pid, setPoint;
  long lastProcess;

  PID(double _kP, double _kI, double _kD){
    kP = _kP;
    kI = _kI;
    kD = _kD;
  }

  void addNewSample(double _sample){
    sample = _sample;
  }

  void setSetPoint(double _setPoint){
    setPoint = _setPoint;
  }

  double process(){
    error = setPoint - sample;
    float deltaTime = (millis() - lastProcess) / 1000.0;
    lastProcess = millis();

    P = error * kP;
    I = I + (error * kI) * deltaTime;
    D = (lastSample - sample) * kD / deltaTime;
    lastSample = sample;
    pid = P + I + D;
    return pid;
  }
```

```
};
```

Com a definição desta *class*, pode-se fazer o desenvolvimento de todo o código necessário para o Arduino Master, que está de acordo com o diagrama da figura 3.6. Neste programa é necessário realizar a ativação das saídas do Arduino que ativarão os motores, tanto em velocidade como em direção. A taxa de amostragem utilizada para o PID foi de 100 Hz.

Para o cálculo da velocidade linear de cada roda é utilizada a equação 3.1, onde D representa o diâmetro da roda em metros, P o número de pulsos lidos pelo encoder num determinado intervalo de tempo, N o número de pulsos que o encoder dá por rotação e T o intervalo de tempo da leitura em segundos. No caso do número de pulsos este será igual a 1200, pois existe a desmultiplicação de 1:3 no acopolamento do *encoder*, e a multiplicação por quatro devido ao uso dos dois sinais do *encoder*.

$$v = \frac{\pi \times D \times P}{4 \times N \times T}(m/s) \tag{3.1}$$

Após a implementação do programa no Arduino, é necessário fazer o ajuste dos parâmetros do controlador PID. O método encontrado para a definição do controlador PID foi com base em tentativa e erro. De modo a facilitar todo este processo desenvolveu-se primeiramente um programa de Arduino que permite fazer o ajuste das constantes do PID através do computador, evitando que seja necessário o envio de um novo programa com as novos valores. Quando a resposta dos motores corresponder às expectativas, é então elaborado um programa de Arduino final que contém as constantes do controlador finais.

Ao longo dos testes desenvolvidos na plataforma para a obtenção do PID um dos motores deixou de funcionar. Devido ao facto de novos motores demorarem algum tempo a chegarem optou-se por uma abordagem diferente para o resto do trabalho passando por uma via de simulação, mantendo no entanto toda a compatibilidade com a plataforma real caso ela voltasse a ser considerada.

Pretende-se então simular a plataforma usando o simulador Gazebo e testando da mesma forma a solução proposta para o caso da unidade computacional.

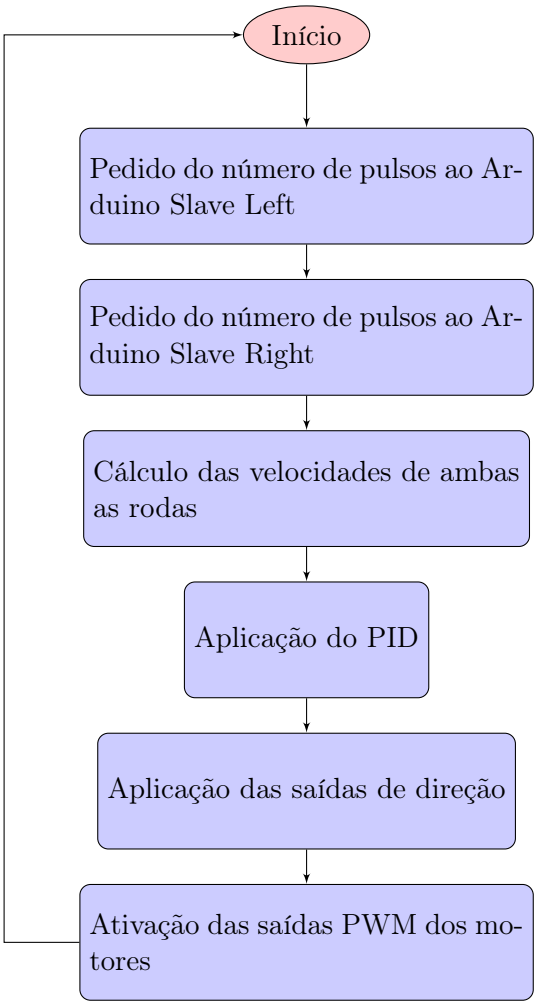


Figura 3.6: Diagrama da estrutura do código utilizado no controlador Master para a implementação do PID

Capítulo 4

Sistema Computacional

Neste capítulo será abordado todo o desenvolvimento e referente ao projeto de uma rede distribuída de Raspberry Pi's operando sobre uma arquitetura ROS. É explicado qual a solução proposta que se pretende implementar, e são apresentadas as etapas para a implementação dessa solução.

4.1 Solução proposta

Como referido no capítulo 1, pretende-se criar uma rede de Raspberry Pi's onde cada um poderá ter várias tarefas a desempenhar. Deste modo pode-se aumentar o poder computacional conforme as necessidades, estando apenas limitados ao poder de processamento de um Raspberry Pi para um nodo ROS.

A utilização dos Raspberry Pi's como unidade computacional foi a selecionada, pois para além de ser uma solução de baixo custo e que se encontravam disponíveis. Estes também são compactos o que permite uma melhor adaptabilidade em futuras aplicações.

Para executar a comunicação entre todas as unidades computacionais é usado o protocolo de comunicação *ethernet*. Como *hardware* utilizado tem-se um *switch* e os Raspberry Pi's e outras unidades computacionais necessárias para a aplicação a desempenhar.

Apesar de neste trabalho ter sido utilizado uma rede *Local Area Network* (LAN) para a comunicação entre as unidades computacionais, esta também poderia ser realizada usando uma rede *wireless*, permitindo assim ter várias unidades de processamento distantes e sem uma ligação física entre elas.

Nesta solução, apenas uma das unidades computacionais irá necessitar de estar a correr o *roscore* estando as restantes dedicadas a executar nodos específicos.

Na figura 4.1 está representado um esquema da rede computacional proposta.

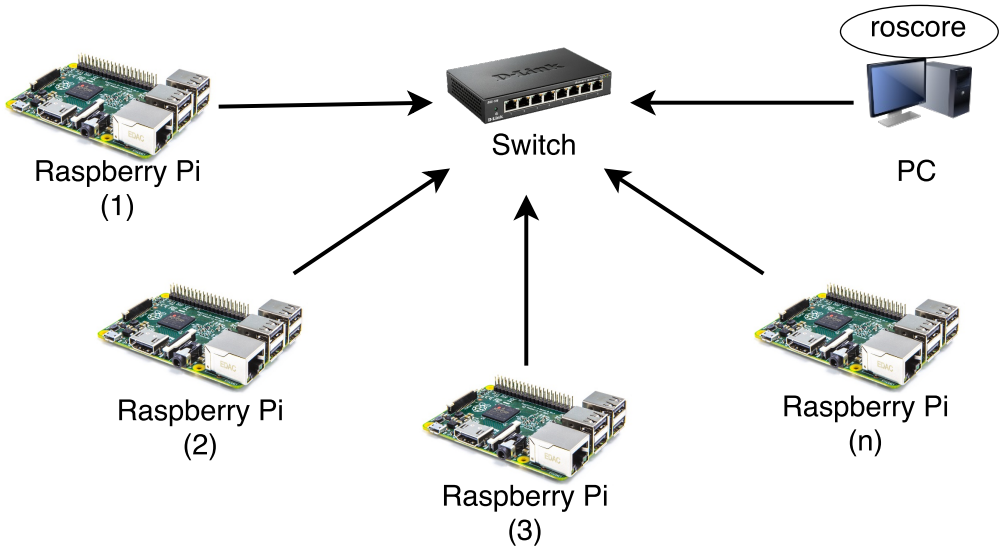


Figura 4.1: Esquema ilustrativo da rede de Raspberry Pi's a implementar

4.2 Instalação do sistema operativo e ROS

Na procura de um sistema operativo que garantisse estabilidade e que satisfaça as exigências que o ROS necessita foi encontrada a opção de instalação do Ubuntu Mate 16.04 LTS. Para se proceder à sua instalação foram feitos os seguintes passos:

- Descarregamento do website oficial a versão do sistema operativo optimizado para Raspberry Pi;
- Montagem de um cartão micro SD com uma capacidade mínima de oito *gigabytes* com a imagem descarregada anteriormente;
- Instalação do sistema operativo no Raspberry Pi.

Fica-se assim em condições de proceder à instalação do ROS. Para tal foi instalado o versão ROS Kinetic seguindo os tutorial de instalação presente em [30].

4.3 Comunicação entre unidades computacionais

De modo a definir a comunicação entre todas as unidades computacionais procedeu-se à definição de um IP fixo em cada máquina, desta forma a inicialização da rede é mais breve.

Para colocarmos cada Raspberry Pi a conseguir comunicar com uma outra máquina onde esteja a correr o *roscore* é necessário definir duas variáveis de ambiente:

- **ROS_MASTER_URI** - A definição desta variável indica aos nodos em que máquina é que o *Master* está a ser executado;
- **ROS_IP** - Tem como objetivo definir o endereço da rede local onde os nodos ROS estão a ser executados.

Para a definição do ROS_MASTER_URI é necessário executar no terminal o seguinte código: `export ROS_MASTER_URI=http://<IP_Master>:11311` onde `<IP_Master>` é referente ao IP da máquina onde o *Master* irá estar em execução.

Quanto à definição do parâmetro ROS_IP é necessário executar o seguinte comando: `export ROS_IP=<IP_local>` onde `<IP_local>` é o IP da máquina onde serão executados os nodos.

De forma a testar a comunicação entre todas as unidades foi feito um teste onde se verificava se um tópico que fosse publicado num Raspberry Pi era visualizado em outro Raspberry Pi usando o comando `rostopic list`. O *setup* utilizado consiste então no uso de um computador onde é executado o *roscore*, e dois Raspberry Pi's onde são publicados tópicos. Na ilustração da figura 4.2 é possível observar a publicação de um tópico num dos Raspberry Pi's, e respetiva visualização no computador e num Raspberry Pi.

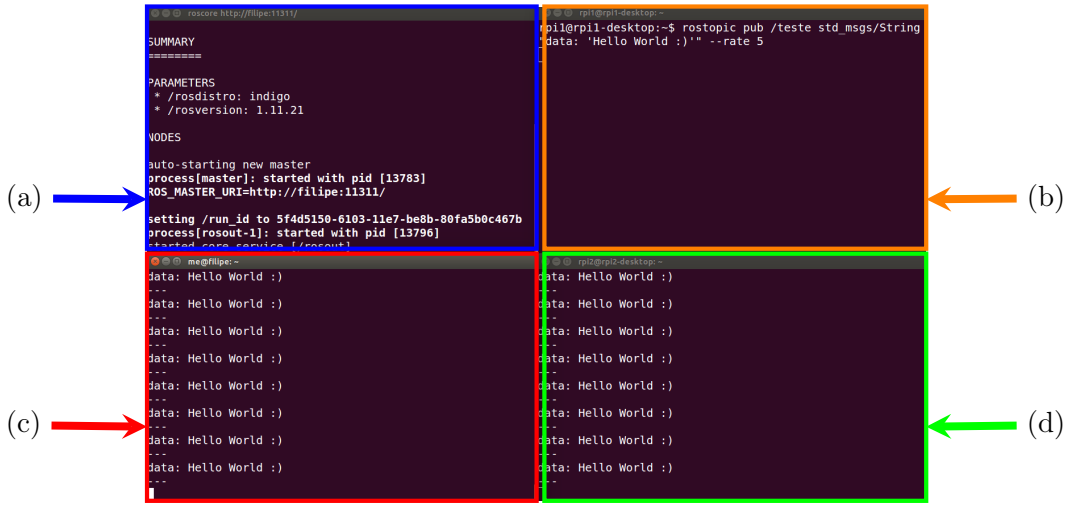


Figura 4.2: Publicação de tópicos e respetiva visualização entre os Raspberry Pi's e o computador (a) Execução do *roscore* no computador (b) Publicação de um tópico num Raspberry Pi (c) Visualização do tópico no computador (d) Visualização do tópico publicado num Raspberry Pi

4.4 Ambiente de Desenvolvimento

De forma a facilitar o trabalho de desenvolvimento envolvendo diferentes máquinas foram criadas determinadas funcionalidades para se obter uma melhor facilidade de uso, descritas a seguir.

4.4.1 Partilha de ficheiros

Como existe a possibilidade de haver um número elevado de máquinas a trabalhar simultaneamente em rede, uma das funcionalidades implementada foi a partilha de ficheiros entre todas. Isto é útil principalmente para se proceder à partilha de *packages* que são desenvolvidos numa máquina principal.

Assim, a solução proposta baseia-se na criação de uma pasta partilhada entre todas as

unidades sendo uma delas o servidor. Para isto, recorreu-se ao uso do *software Network File System* (NFS) que permite criar pasta partilhada dentro de uma rede. Esta pasta encontra-se alojada numa das unidades computacionais que servirá como servidor, conforme ilustrado na figura 4.3. Após feita a configuração do servidor as unidades clientes podem subscrever a pasta partilhada e podem usar o seu conteúdo.

No caso de os ficheiros a serem partilhados corresponderem a *packages* é criar um link simbólico no respectivo diretório do *workspace* ROS referente ao *package* a ser usado. Para isto, pode-se usar o comando `ln -s <real_folder> <link_folder>`, onde `<real_folder>` corresponde ao diretório da pasta real (na pasta partilhada) e `<link_folder>` o diretório onde se quer criar o atalho (ROS *workspace*).

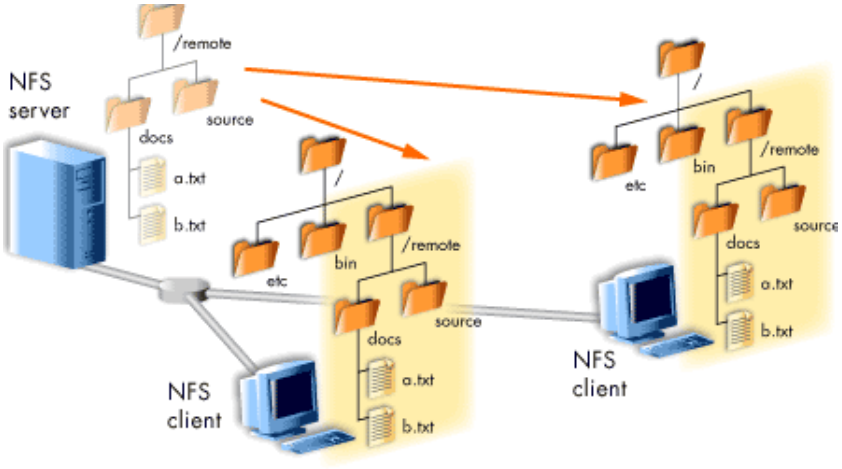


Figura 4.3: Esquema da partilha de ficheiros usando NFS[31]

4.4.2 Controlo remoto

De modo a facilitar o acesso a todas as máquinas presentes, propõe-se o uso de *Secure Shell* (SSH) que é um protocolo de rede criptográfica que permite o acesso remoto a várias máquinas de forma segura [32]. Este controlo remoto tem muita utilidade pois permite fazer o lançamento de um nodo nodo de uma unidade específica sem haver necessidade de se ligar um monitor. No entanto, como o número de unidades computacionais pode ser elevado e de forma a agilizar o processo de *login* desenvolveu-se um *script* que abre um novo terminal referente a cada máquina que estiver definida (listagem 4.1).

Listagem 4.1: *Script* usado para fazer a inicialização das unidades remotas no computuador principal

```
#!/bin/sh
echo Inicializando Raspberry Pi 1
gnome-terminal -x sshpass -p password ssh -Y rpi1@192.168.1.10
echo Inicializando Raspberry Pi 2
gnome-terminal -x sshpass -p password ssh -Y rpi2@192.168.1.11
```

Para facilitar a visualização de alguma interface gráfica referente a algum nodo, a inicialização da comunicação SSH é feita com o parâmetro `-Y`, permitindo assim que as interfaces gráficas possam ser visualizadas num computador remoto.

Ainda para facilitar o *setup* das várias unidades computacionais, foi desenvolvido um *script* específico que faz a inicialização de todos os processos necessários, incluindo a declaração das variáveis de ambiente para a comunicação (listagem 4.2).

Listagem 4.2: Exemplo de *script* usado para na inicialização das unidades remotas

```
#!/bin/sh
echo Definindo as variáveis de ambiente
export ROS_MASTER_URI=http://192.168.1.1:11311
export ROS_IP=192.168.1.11
```

4.4.3 *Launch file* geral

De forma a fazer a inicialização de todos os nodos distribuídos por vários Raspberry Pi's é possível criar um *launch file* que realiza o lançamento de nodos em máquinas remotas. Para isto é necessário criar dois *launch files* na unidade computacional principal:

- Um referente a que nodo tem de ser executado na unidade computacional remota. Esta tem de mencionar o nome do nodo e respetivo *package* a executar.

  Na listagem 4.3 encontra-se o ficheiro `rpi1.launch`, que é uma *launch file* referente a um nodo denominado por `demo_node`.

- O outro ficheiro é também um *launch file* onde é referido qual o *launch file* que tem de ser executado na máquina remota. Na fistagem 4.4 está um exemplo de um *launch file* que pretende executar o `rpi1.launch` na máquina com o utilizador "rpi1".

Listagem 4.3: *Launch file* para a execução do nodo `demo_node`

```
<launch>
  <node name="demo_node" pkg="mybot_demo" type="demo_node" />
</launch>
```

Listagem 4.4: *Launch file* que permite fazer o lançamento do ficheir `rpi1.launch` numa unidade remota

```
<launch>
    <machine name = "me" address="192.168.1.10" user="rpi1"
             env-loader="/opt/ros/kinetic/env.sh"/>
    <include file="rpi1.launch"/>
</launch>
```

Capítulo 5

Ambiente de simulação

Neste capítulo descrevem-se os passos executados para a obtenção do ambiente de simulação, desde a definição do URDF do robô até à sua visualização no simulador.

5.1 Definição do robô

Para a criação do ambiente de simulação a primeira etapa a ser executada é a realização de algo que consiga descrever todo o robô. Deste modo, foi criado o modelo em URDF que permite definir o robô, e onde se encontra a sua forma geométrica bem como o tipo de juntas que o compõem. Para o robô em questão, pode ser encontrado na figura 5.1 um esquema das diversas partes que o compõem assim como as ligações entre elas.

Os diversos componentes que compõem a plataforma tiveram de ser criados/adaptados recorrendo à modelação 3D, neste caso usando o *software* SolidWorks (figura 5.2). Nesta definição do robô é possível definir quais as propriedades dos diversos componentes, tais como a massa e propriedades inerciais, permitindo assim ao simulador realizar uma simulação o mais realista possível.

Ainda, para a definição do robô é necessário implementar o tipo de controlador que a plataforma terá de ter. Neste sentido, o Gazebo possui um *plugin* que permite integrar o controlo de uma plataforma *skid-steering*. Para a sua integração é necessário identificar quais os nomes das juntas correspondentes a cada roda, qual a distância entre rodas e o seu diâmetro, o torque máximo, nome dos tópicos onde o Gazebo irá publicar certas informações como odometria, e qual o nome do tópico que terá as informações para o robô ser comandado. Na listagem 5.1 pode ser encontrado um excerto de código referente à implementação do *plugin*.

No total, para a definição do robô foram elaborados quatro ficheiros descritos seguidamente:

- `macros.xacro` - Neste ficheiro estão definidas as macros usadas ao longo da definição do robô;
- `materials.xacro` - Aqui são definidas macros relacionadas com as propriedades dos materiais, como por exemplo a cor;
- `mybot.gazebo` - Neste ficheiro são definidos os *plugins* do Gazebo utilizados;

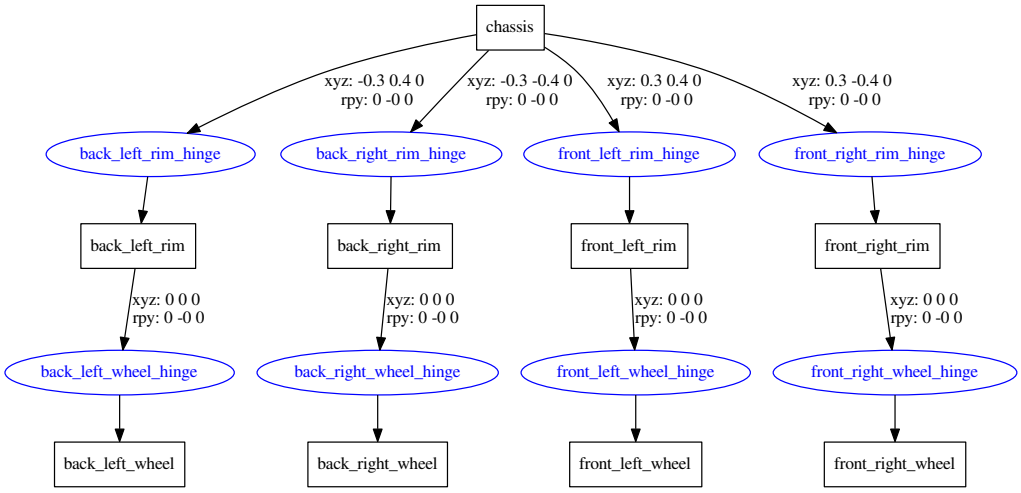
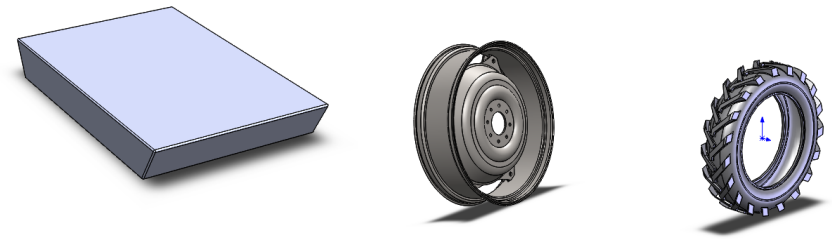


Figura 5.1: Diagrama do URDF do robô gerado usando a ferramenta `urdf_to_graphiz`



(a) (b) (c)

Figura 5.2: Componentes criados para em SoliWorks (a) Chassis (b) Jante (c) Pneu

- `mybot.xacro` - Aqui são definidas todas as partes do robô assim como o tipo de juntas que existe entre elas.

Listagem 5.1: Definição do *plugin* referente a plataformas *Skid-steering*

```
<gazebo>
  <plugin filename="libgazebo_ros_skid_steer_drive.so"
          name="skid_steer_drive_controller">
    <updateRate>100.0</updateRate>
    <leftFrontJoint>front_left_rim_hinge</leftFrontJoint>
    <rightFrontJoint>front_right_rim_hinge</rightFrontJoint>
    <leftRearJoint>back_left_rim_hinge</leftRearJoint>
    <rightRearJoint>back_right_rim_hinge</rightRearJoint>
    <wheelSeparation>0.7</wheelSeparation>
    <wheelDiameter>0.175</wheelDiameter>
    <robotBaseFrame>chassis</robotBaseFrame>
    <MaxForce>5.0</MaxForce>
    <torque>20</torque>
    <commandTopic>cmd_vel</commandTopic>
    <odometryTopic>odom</odometryTopic>
    <odometryFrame>odom</odometryFrame>
    <broadcastTF>1</broadcastTF>
  </plugin>
</gazebo>
```

5.2 Integração no Gazebo

Para se proceder à visualização do robô no Gazebo é necessário a criação de um ficheiro `.launch` (listagem 5.2). Este ficheiro irá definir qual será a configuração do ambiente envolvente através da definição de um ficheiro apropriado, e inclui também a localização do ficheiro com o URDF assim como algumas configurações iniciais para o Gazebo.

Listagem 5.2: *Launch file* para a execução do Gazebo com o modelo da plataforma

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
  <arg name="world" default="empty"/>
  <arg name="paused" default="false"/>
  <arg name="use_sim_time" default="true"/>
  <arg name="gui" default="true"/>
  <arg name="headless" default="false"/>
  <arg name="debug" default="false"/>
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find mybot_gazebo)/worlds/mybot.world"/>
    <arg name="paused" value="$(arg paused)"/>
    <arg name="use_sim_time" value="$(arg use_sim_time)"/>
    <arg name="gui" value="$(arg gui)"/>
    <arg name="headless" value="$(arg headless)"/>
```

```
    <arg name="debug" value="$(arg debug)"/>
  </include>
  <param name="robot_description" command="$(find xacro)/xacro.py
              '$(find mybot_description)/urdf/mybot.xacro'"/>
  <node name="mybot_spawn" pkg="gazebo_ros" type="spawn_model"
              output="screen"
        args="-urdf -param robot_description -model mybot" />
</launch>
```

Para a definição de todo a ambiente envolvente, com o qual o robô pode interagir, tem de se criar primeiramente utilizando o Gazebo um ficheiro que terá todas as informações acerca do mesmo, desde os objetos existentes e fontes de luz. Posteriormente, pode ser gravado um ficheiro `.world` que será chamado pelo *launch file*.

Após a execução do ficheiro `.launch` o Gazebo é executado, obtendo-se o resultado visualizado na figura 5.3.

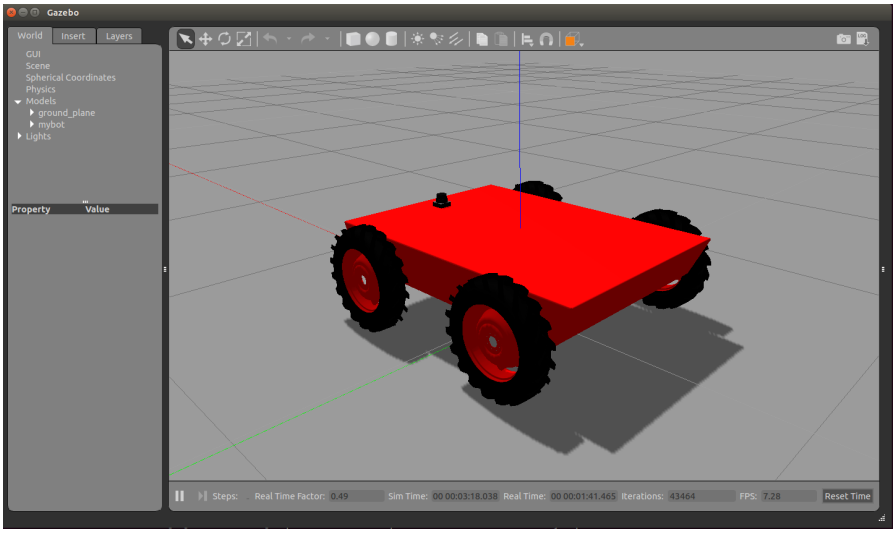


Figura 5.3: Visualização do robô no Gazebo

De forma a testar se o modelo do robô está operacional para a comunicação com o ROS realizou-se a publicação de uma mensagem de velocidade no tópico `/cmd_vel`, correspondendo então ao movimento do robô virtual.

5.3 Integração de sensores

Para fins de teste, foram adicionados alguns sensores à plataforma de modo a observar os diversos resultados.

Para tal, a sua adição passou por dois passos. Primeiramente a adição da parte geométrica do sensor no ficheiro `mybot.xacro`, onde se indica as suas dimensões, forma, inércia e espaço de colisão, e qual a sua posição relativamente ao robô. Posteriormente é necessário indicar qual o tipo de sensor que se pretende simular, sendo então necessário integrar o respetivo *plugin* no ficheiro `mybot.gazebo`, onde se indica os tópicos onde os dados serão publicados assim como as suas características.

5.4 Interação com Rviz

Para complementar todo o ambiente de simulação é importante existir um visualizador onde seja possível observar o resultado da simulação. No caso em questão recorreu-se ao Rviz.

Assim, criou-se um *launch file* que permite a inicialização do visualizador para o caso do robô utilizado. Posteriormente, na interface do Rviz é necessário proceder à adição do robô e dos restantes dados que se quer visualizar, como por exemplo os dados de um laser.

Ainda no *launch file*, são executados dois nodos, o `joint_state_publisher` e o `robot_state_publisher`. Em relação ao `joint_state_publisher` este irá ver todo o ficheiro URDF encontrando todas as juntas existentes e para aquelas que não são fixas publicará valores de juntas. Por outro lado, o `robot_state_publisher` é responsável por fazer uma leitura do estado das juntas do robô e fará a publicação do seu estado da forma de ROS `tf`, que poderá posteriormente ser usada para ver o estado as juntas do robô no Rviz.[17]

Na listagem 5.3 pode ser observada a *launch file* utilizada.

Listagem 5.3: Ficheiro `mybot_rviz.launch` usado para a execução do Rviz

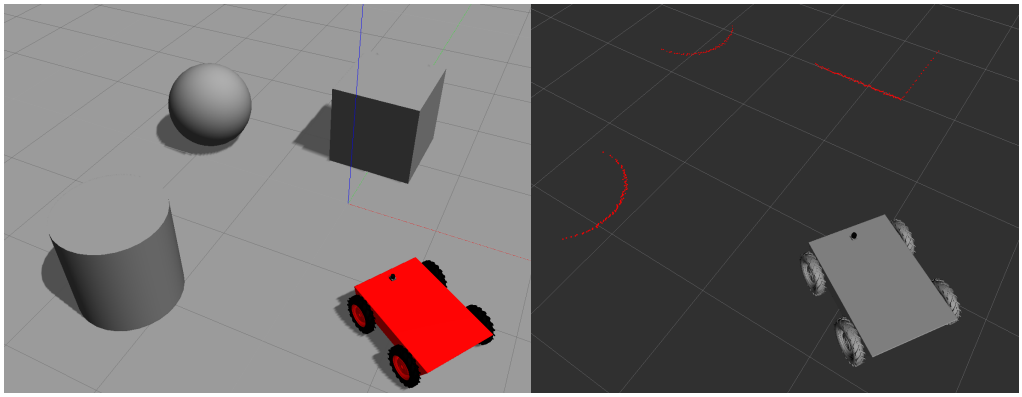
```
<?xml version="1.0"?>
<launch>
  <param name="robot_description" command="$(find xacro)/xacro.py
           '$(find mybot_description)/urdf/mybot.xacro'"/>
  <node name="joint_state_publisher" pkg="joint_state_publisher"
        type="joint_state_publisher">
    <param name="use_gui" value="True"/>
  </node>
  <node name="robot_state_publisher" pkg="robot_state_publisher"
        type="state_publisher"/>
  <!-- Show in Rviz -->
  <node name="rviz" pkg="rviz" type="rviz"/>
</launch>
```

Na figura 5.4 pode observar-se o resultado dos valores lidos por um laser simulado no Rviz.

5.5 Resultado final

Para a obtenção do ambiente de simulação foi necessário criar alguns *packages* ROS. Os packages criados são os seguintes:

- `Description` - Neste *package* estam presentes os ficheiros que descrevem o robô (URDF), e ficheiros necessários para a definição da forma de partes do robô. A launch file que é usada para a visualização no Rviz também se encontra neste *package*.
- `Gazebo` - Aqui encontram-se os *launch files* que permitem correr a simulação em Gazebo, assim como os ficheiros com a configuração do mundo.



(a) (b)

Figura 5.4: (a) Simulação em Gazebo da plataforma na presença de alguns objetos (b) Visualização em Rviz dos dados lidos pelo laser[13]

No esquema 5.5 pode ser visualizado todo o esquema de ficheiros utilizado para a simulação.

No final da implementação deste ambiente de simulação obtém-se um esquema dos nodos e tópicos final como o representado na figura 5.6. Aqui podem ser observados os tópicos publicados pelo laser (`/mybot/laser/scan`), tópicos publicados pelo Gazebo (`/tf`), assim como dos nodos do `joint_state_publisher` e `robot_state_publisher` descritos na secção 5.4.

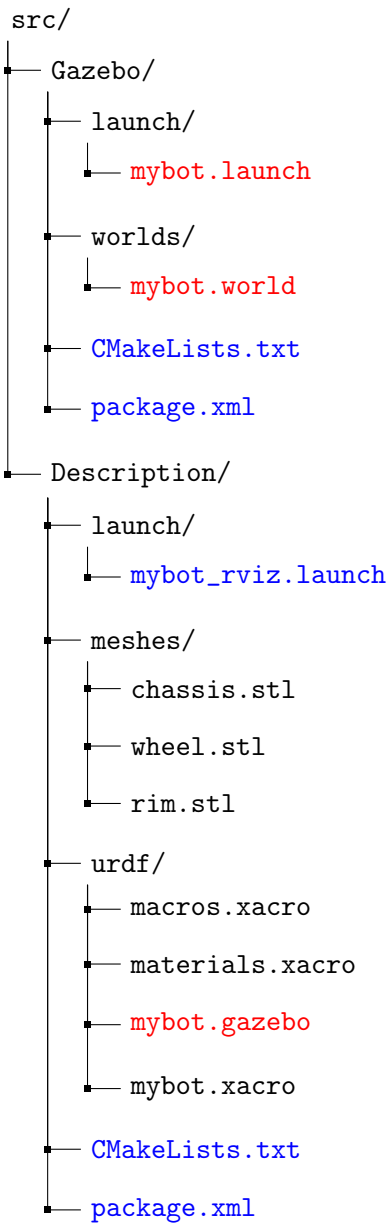


Figura 5.5: Esquema de ficheiros final para o ambiente de simulação virtual, estando a azul os ficheiros somente utilizados pelo ROS, a vermelhos os ficheiros somente usados pelo Gazebo, e os que são usados por ambos a preto

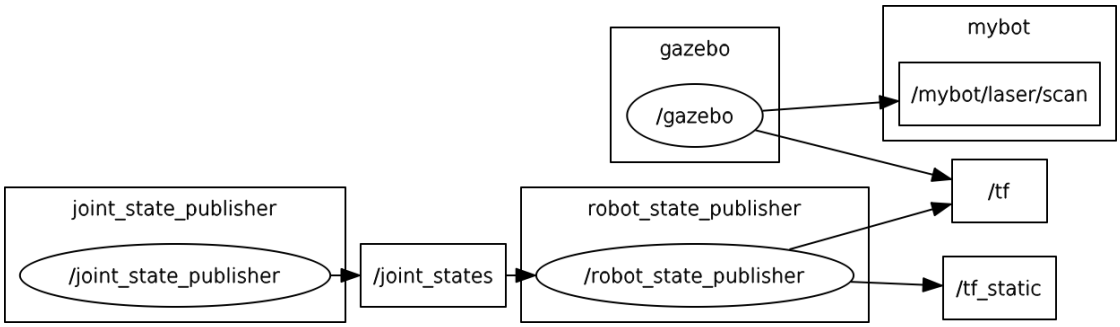


Figura 5.6: Diagrama dos nodos e tópicos para o caso do ambiente de simulação

Capítulo 6

Uma aplicação ilustrativa

Este capítulo descreve a aplicação realizada de forma a testar as soluções propostas. Essa aplicação assenta na simulação do seguimento de uma linha com uma câmara virtual em Gazebo.

6.1 Aplicação proposta

De forma a testar o sistema distribuído de ROS, propõe-se o desenvolvimento de uma aplicação usando os Raspberry Pi's e o simulador desenvolvido.

A aplicação proposta é baseada no seguimento de uma linha que será detectada com o recurso a uma câmara. No entanto, é adicionado o fator humano à aplicação por intermédio de um comando *joystick*, onde as ordens vindas deste terão prioridade perante as anteriores. Quando as mensagens oriundas do comando deixarem de existir o robô iniciará a procura da linha e posterior guiamento através da mesma.

Deste modo é possível ter diversas unidades computacionais a funcionar ao mesmo tempo, estando uma dedicada a fazer o processamento da imagem da câmara, outro dedicado ao comando que é manipulado por uma pessoa, e por fim outra unidade onde será realizada a simulação.

6.2 Desenvolvimento

6.2.1 Criação do ambiente com a pista

Para a realização da aplicação proposta teve-se de realizar a modelação da pista, tendo sido desenvolvido um ficheiro do tipo COLLADA utilizando o *software* SketchUp (figura 6.1). Utilizando este tipo de ficheiro, propriedades como a cor mantém-se em comparação com um ficheiro *Standard Triangle Language* (STL), permitindo que o ficheiro inclua o desenho da linha a seguir.

Para a criação da trajetória a seguir pela plataforma optou-se por algo realizado de forma genérica, mas onde se incluem várias formas como retas e curvas com diferentes curvaturas. O resultado final pode ser observado na figura 6.2. Nos limites da pista é colocada uma parede de forma a evitar que a plataforma os passe.

De forma a visualizarmos a pista no simulador, procedeu-se à criação do ficheiro `.world` e o *launch file* respetivo.

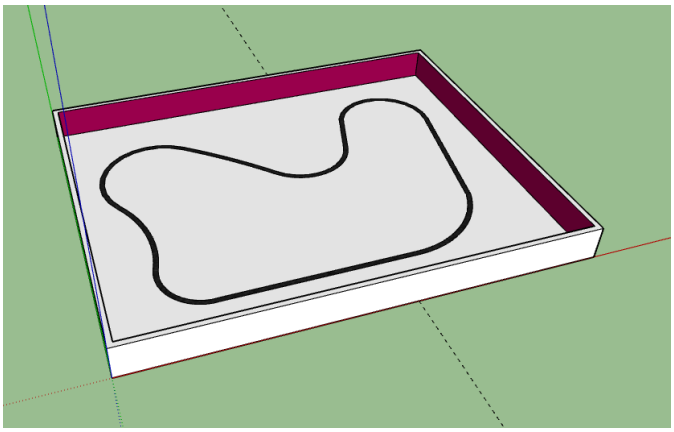


Figura 6.1: Pisda criada no *software* SketchUp

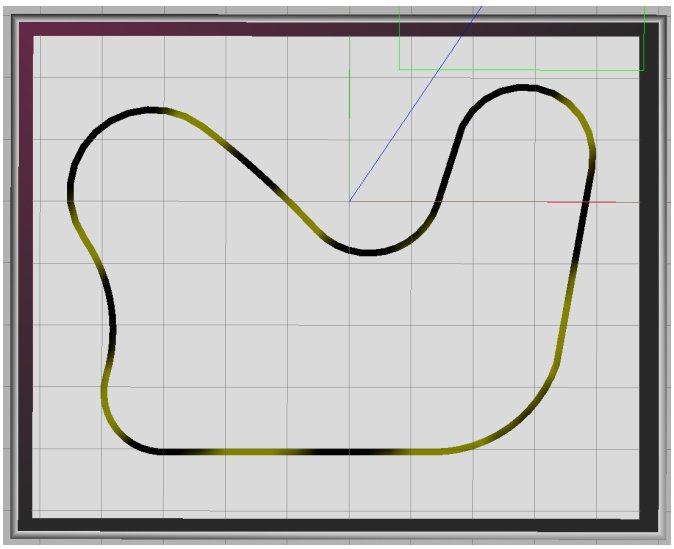


Figura 6.2: Visualização da pista visualizada no Gazebo

6.2.2 Alterações efetuadas à plataforma

Para a realização da aplicação proposta foi necessário adicionar novos sensores à descrição da plataforma. Para tal adicionou-se uma câmara para fazer a detecção da linha, e um scanner laser para detetar os limites da pista criada.

Para a adição da câmara fez-se a edição de alguns dos ficheiros mencionados na secção 5.1. Primeiramente, no ficheiro `mybot.xacro` é necessário definir o seu aspeto visual, assim como propriedades inerciais e colisões. Posteriormente, no ficheiro `mybot.gazebo` é feita a definição do *plugin* do Gazebo que permite realizar a captura da imagem e respetiva publicação num tópico ROS, neste caso o `libgazebo_ros_camera.so`. Para aumentar o nível de semelhança entre o cenário virtual e a realidade foi adicionado um ruído à imagem extraída através da câmara.

Para o caso do scanner laser, o procedimento efetuado foi semelhante ao da câmara, tendo-se usado o *plugin* `libgazebo_ros_gpu_laser.so` para fazer a simulação do laser, no qual se definem propriedades como os limites de alcance e resolução do laser.

6.2.3 Arquitetura da aplicação

Para o desenvolvimento da aplicação foram desenvolvidos quatro nodos distintos.

line_node

A função principal deste nodo é realizar o tratamento da imagem proveniente da câmara e dar as respetivas ordens de movimento à plataforma.

A primeira operação a realizar neste nodo é realizar o tratamento da imagem que é publicada num tópico. Para isto recorre-se ao package `cv_bridge` que permite a conversão entre imagens ROS e imagens OpenCV. Ao fim de se ter a imagem no formato OpenCV realiza-se o seu tratamento de forma a que no final se obtenha uma imagem binarizada e onde apenas a linha a seguir seja reconhecida.

Posteriormente ao tratamento de imagem admite-se que a plataforma pode estar entre dois estados, numa situação em que a linha é perfeitamente visualizada pela plataforma e outro onde o robô não reconhece/visualiza a linha.

No primeiro caso a estratégia adotada para fazer o seguimento da linha baseia-se em encontrar um ponto central e superior da linha e verificar qual a sua distância para o centro da imagem. Com base nesta distância recorreu-se à aplicação de um controlador PID que faz o ajuste da velocidade angular a aplicar à plataforma consoante a distância em *pixels* entre o centróide e o centro da imagem.

No caso de a plataforma se encontrar perdida, esta começa a andar aleatoriamente e quando deteta na imagem uma parte correspondente à linha a plataforma começa a aproximar-se da mesma e quando está sobre a linha a plataforma começa a rodar sobre si própria de forma a alinhar-se com a linha. Quando se encontra totalmente alinhada a plataforma entra no estado descrito anteriormente.

Por fim, os valores de velocidades calculados para os diferentes estados são publicados no tópico `/nardo/line_vel`.

Todos os procedimentos do código podem ser observados no algoritmo 1.

Algoritmo 1: Algoritmo para o seguimento da linha com base na imagem recebida pela câmara

```
1: Tratamento de imagem com base em binarizações
2: Análise da imagem binarizada
   if imagem vazia then
|  state = 1
   end
   if state = 1 then
   if imagem vazia then
   |  Andar em frente;
   else
   |  if Primeira e última coluna da imagem ≠ 0 then
   |  |  if Centróide do objeto em y menor que metade da altura da imagem
   |  |  then
   |  |  |  Plataforma anda em frente
   |  |  else
   |  |  |  Plataforma roda sobre si;
   |  |  |  if Primeira e última linha da imagem ≠ 0 then
   |  |  |  |  state = 0
   |  |  |  end
   |  |  end
   |  else
   |  |  Plataforma anda em direção ao objeto detetado na imagem;
   |  end
   end
   else
   Encontrar centróide na linha numa parte superior;
   Cálculo da diferença em x para o centro da imagem;
   Aplicação do PID para cálculo da velocidade angular;
   Publicação da velocidade final no respetivo tópico;
   end
```

No algoritmo 1 está descrito o procedimento para a análise da imagem e respetivas ordens de controlo.

joy_node

Como descrito na secção 6.1, para realizar o controlo da plataforma manualmente é usando um *gamepad*. Deste modo é necessário criar um nodo que seja responsável pela aquisição dos seus dados e posterior tratamento.

Para proceder à aquisição dos dados é usado o *package* `joy` [33] que publica o estado de todas as componentes do comando no tópico `/joy`. De modo a converter os dados recebidos em uma mensagem para o controlo da plataforma, é então criado o nodo `joy_node` que fará a leitura dos dados publicados no tópico `/joy`. Para a definição da velocidade linear é usado o *joystick* esquerdo, enquanto que para a velocidade angular é usado o *joystick* direito (figura 6.4). Para ambas as velocidades, o seu valor é proporcional ao deslocamento do joystick dado pelo utilizador, sendo as mensagens publicadas no tópico `/nardo/joy_vel`.

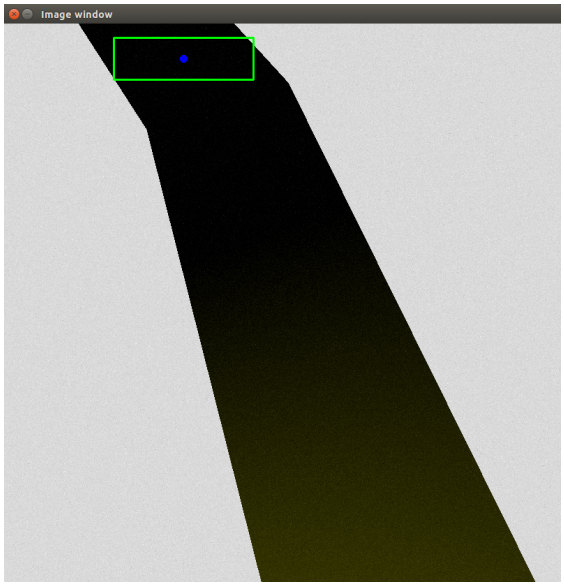


Figura 6.3: Visualização da linha e deteção do centróide

No diagrama da figura 6.5 estão exemplificados as etapas executadas por este nodo sempre que é pressionado um botão no *gamepad*.

laser_node

De forma a precaver a existência de colisões foi então criado um nodo que faz a leitura dos dados obtidos pelo scanner laser dando um alerta quando a plataforma se encontra numa situação de emergência, ou seja quando a plataforma se está a aproximar dos limites da pista.

Esta situação de emergência é atuada quando a distância mínima que o laser reconhece é menor que 40 centímetros, resultando na publicação de uma mensagem segundo o formato booleano no tópico `/nardo/laser`, correspondendo o valor `0` ao estado de emergência (algoritmo 2).

Algoritmo 2: Algoritmo usado no tratamento dos dados lidos pelo laser

1: Deteção do menor valor lido pelo laser
2: Análise do menor valor
if *Menor valordlido pelo laser* $\leq$ *0.4* **then**
  Estado de emergência;
else
  Estado normal;
end

decision_node

Como descrito anteriormente, são publicadas várias mensagens que visam o controlo da plataforma, no entanto é necessário decidir quando é que cada uma o deve fazer. Para

Figura 6.4: Publicação de tópicos e respetiva visualização entre os Raspberry Pi's e o computador

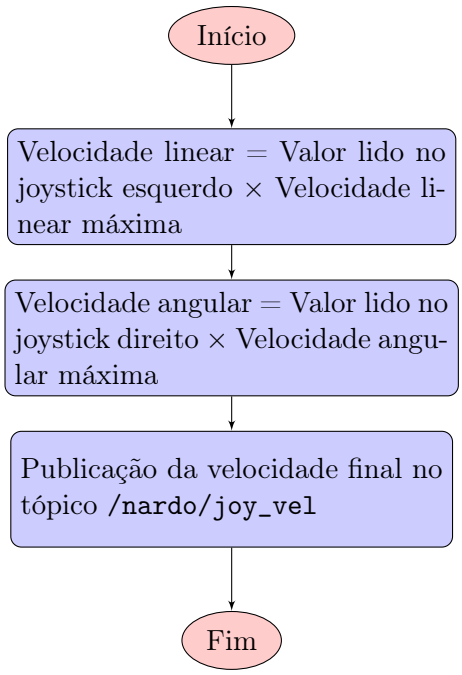


Figura 6.5: Diagrama da estrutura da *callback* usada quando são recebidos dados do *gamepad*

isto, é criado o nodo `decision_node` que tem como objetivo interpretar as mensagens publicadas pelos três nodos descritos anteriormente, resultando numa mensagem final que controla a plataforma.

Assim, este nodo pode estar em três estados. O estado padrão é quando a plataforma na rotina referente ao seguimento da linha, onde estão os dados recebidos no tópico `/nardo/line_vel` são transpostos para a mensagem de controlo da plataforma, `/nardo/cmd_vel`.

O seguinte estado possível é quando a plataforma está a ser comandada pelo *gamepad* e então as mensagens de controlo a enviar para a plataforma são iguais às enviadas pelo nodo `joy_node`.

Por último, o terceiro estado é ativado quando a plataforma encontra-se em estado de emergência, tendo este estado prioridade sobre qualquer outro. Quando este estado é ativado são dadas ordens à plataforma para rodar 180 graus sobre si própria e andar em frente. Para a realização destes movimentos são enviados comandos de movimento para a plataforma até percorrer o movimento desejado, usando os dados recebidos pelo tópico `/odom` para verificar o estado do robô.

Interligando todos os nodos mencionados anteriormente, obtêm-se os diagramas demonstrados na figura 6.6, onde estão representadas as várias *callbacks* e a estrutura do programa principal.

A estrutura final da aplicação pode ser consultada na figura 6.7, onde é possível observar todos os nodos e tópicos utlizados.

6.3 Resultados finais

Após o desenvolvimento da aplicação, é realizado o teste de todo o sistema. Para tal, distribuem-se os diferentes nodos pelos Raspberry Pi's a utilizar. Para os testes realizados recorre-se a um computador pessoal onde é executado o simulador, o `decision_node` e o `joy_node`. São usados dois Raspberry Pi's, um da versão 3 executa o `laser_node`, e um da versão 2 que processa o `line_node`. Esta disposição pode ser consultada na figura 6.8 e 6.9.

De forma a poder-se visualizar se o controlo da plataforma é realizado convenientemente, fez-se a visualização da trajetória percorrida no `Rviz`. Para o caso do percurso de toda a linha no mapa, obtem-se o resultado obtido na figura 6.10, que comparando com a trajetória da pista demonstrada na figura 6.2 pode-se aferir que se encontram bastante semelhantes, estando também presente na mesma imagem a trajetória que a plataforma executa quando encontra a linha. O gráfico da figura 6.11 demonstra qual o módulo da variação em *pixles* entre um ponto no centro da linha e o centro da imagem da câmara segundo a direção horizontal, ao longo do percurso da plataforma pela linha.

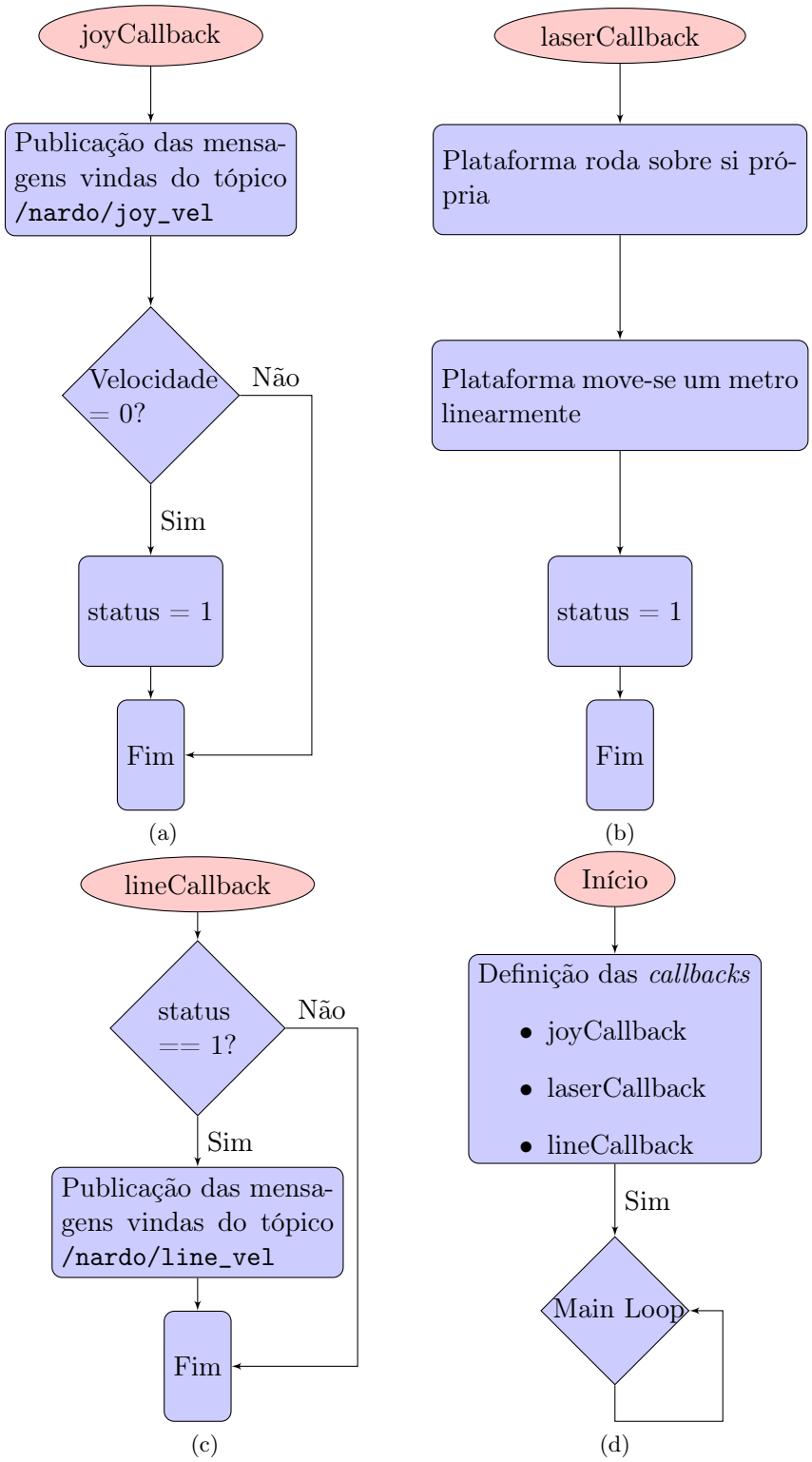


Figura 6.6: (a) Callback acionada quando é recebido uma mensagem no tópico `/nardo/joy_vel` (b) Callback acionada quando é recebido uma mensagem relativa a uma situação de emergência (c) Callback acionada quando a é recebido uma mensagem no tópico `/nardo/line_vel` (d) Estrutura geral do programa

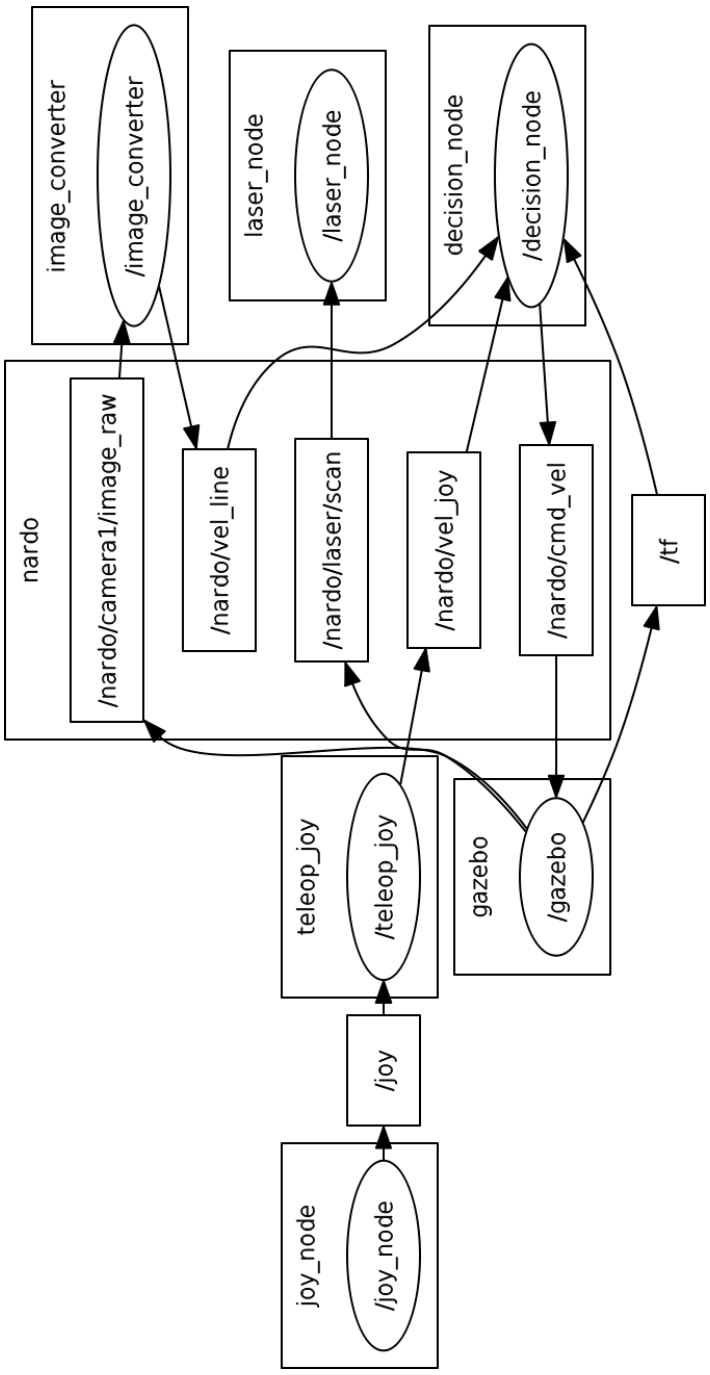


Figura 6.7: Diagrama do nodos e tópicos da aplicação final

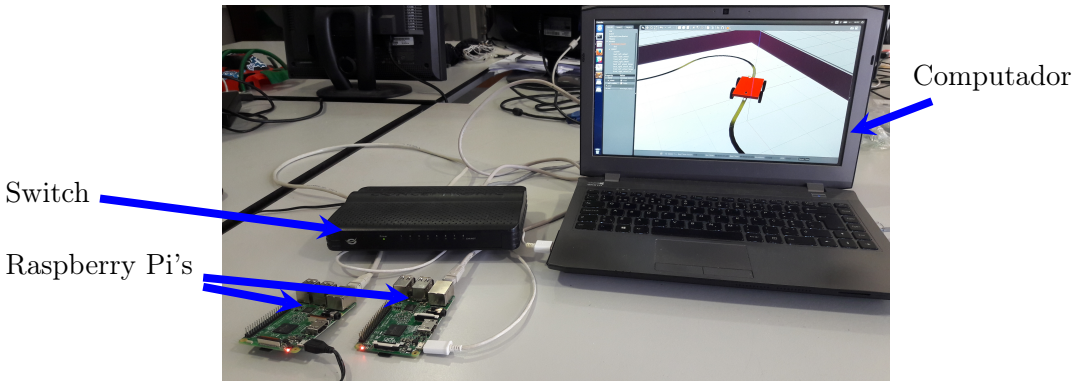


Figura 6.8: Ilustração com o sistema todo

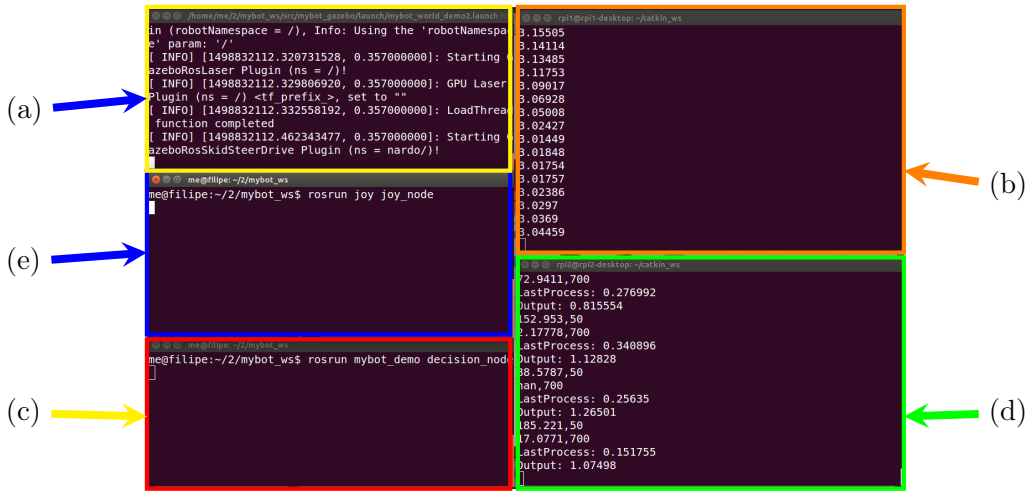


Figura 6.9: Publicação de tópicos e respetiva visualização entre os Raspberry Pi's e o computador; (a)Execução do simulador no Gazebo; (b) Execução do `laser_node` num Raspberry Pi; (c) Execução do `joy_node` no computador; (d) Execução do `line_node` num Raspberry Pi; (e) Execução do `decision_node` no computador

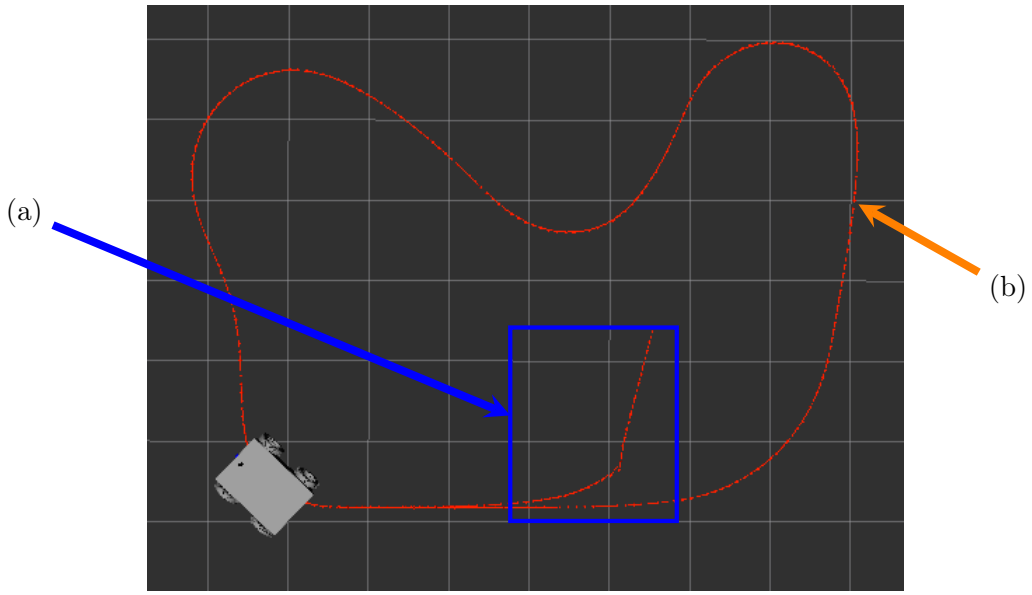


Figura 6.10: (a) Trajetória de aproximação à linha (b) Trajetória percorrida pelo robô no seguimento da linha

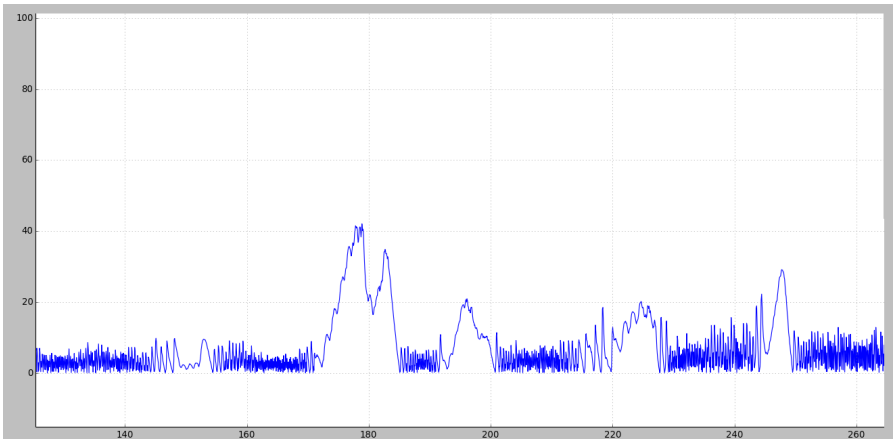


Figura 6.11: Gráfico gerado usando a ferramenta `rqt_plot` com a variação da distância em módulo entre o centro da imagem e o centróide da linha

Capítulo 7

Conclusões e trabalho futuro

7.1 Conclusões

No âmbito desta dissertação foram realizadas diversas tarefas. Começou-se inicialmente por fazer o controlo em malha fechada dos motores da plataforma NARDO, sendo para tal necessário a instalação de *encoders* e a realização da sua leitura. Para a leitura dos *encoders* foi desenvolvido um sistema com um microcontrolador dedicado a esta tarefa que posteriormente transmite os dados a outro microcontrolador principal para o seu processamento. Em relação ao controlo em malha fechada foi implementado um controlador PID que recebe as velocidade desejadas da plataforma. Importa referir que apesar de se ter chegado a umas constantes para o controlador estas foram testadas com a plataforma em suspensão, pelo que será necessário realizar novamente a afinação dos parâmetros quando a plataforma for colocada em contacto com o solo. O uso da solução apresentada verificou-se adequada.

O sistema distribuído de ROS foi desenvolvido com sucesso verificando-se que é possível fazer a comunicação entre as várias unidades computacionais. A par deste sistema, foram ainda desenvolvidas algumas ferramentas que permitem uma melhor facilidade de desenvolvimento tais como a partilha de ficheiros entre as diversas unidades e também uma solução para o controlo remoto das diversas unidades.

Posteriormente, devido à ocorrência de problemas num dos motores da plataforma que impediu o seu uso, optou-se por criar um ambiente de simulação para realizar testes. Este ambiente foi desenvolvido usando o simulador Gazebo que permite a interação com o ROS. Assim, criou-se o modelo completo da plataforma com a adição de sensores. Pode-se concluir que o simulador utilizado satisfez as necessidades requeridas permitindo a interação com o robô e meio envolvente.

Por último, foi realizada uma aplicação que permite o teste de todas as componentes desenvolvidas. Nesta aplicação foram colocados vários nodos distribuídos por Raspberry Pi's, sendo então possível verificar que é possível ter um sistema de ROS distribuído a realizar o controlo de uma plataforma *skid-steering*, embora apenas tenha sido realizado no ambiente virtual.

Em suma, com a solução desenvolvida é possível prever que esta pode ser útil para diversas aplicações tais como o controlo de robôs. É de salientar a modularidade que este tipo de solução pode trazer pois permite com facilidade que seja adicionado um novo sensor ou funcionalidade a um robô.

7.2 Trabalho futuro

Após a realização deste trabalho ficam ainda abertas outras opções de trabalho a realizar no futuro:

1. **Compilação cruzada**

   Uma das funcionalidades que se sugere que seja adicionada a todo o sistema distribuído de ROS é a possibilidade de se fazer compilação cruzada do código desenvolvido num computador pessoal. Desta forma permitimos que não seja necessário fazer a compilação de código no Raspberry PI, reduzindo assim o tempo de compilação.

2. **Teste do sistema distribuído num robô real**

   Após a validação do sistema distribuído de ROS usando o ambiente de simulação, sugere-se que este seja consequentemente testado numa plataforma real, permitindo assim provar que a comunicação com o hardware é realizada com sucesso. Isto permite ainda verificar se o simulador desenvolvido consegue corresponder ao que realmente acontece na realidade.

3. **Implementação de novos motores e respetivo controlo da plataforma**

   Como um dos motores da plataforma se encontra inoperacional é necessário proceder à sua substituição. Posteriormente será necessário adaptar os valores dos controladores PID para ambos os motores.

   Posteriormente ao controlo dos motores será necessário realizar o controlo por parte da unidade computacional, ou seja, criar um nodo ROS que permita receber comandos de velocidade e converter para uma mensagem que seja compreendida pelo controlador dos motores, por exemplo de uma mensagem do tipo Twist para as velocidades que cada roda tem que ter. Para isto, será necessário aplicar as equações cinemáticas da plataforma.

Bibliografia

[1] *Farming (r)evolution | Robohub*. URL: http://robohub.org/farming-revolution/ (acedido em 30/05/2017).

[2] *Our Mission*. URL: http://tarentorobotics.com/ (acedido em 30/05/2017).

[3] URL: https://www.google.pt/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&ved=0ahUKEwjD0NeGjJjUAhUDDxoKHaZwA-gQjhwIBQ&url=https%3A%2F%2Fangel.co%2Ftarento-robotics&psig=AFQjCNGPNyGuaUnatZ2yWmdmgRlRW9dKdg&ust=1496250080441709&cad=rjt (acedido em 30/05/2017).

[4] Joshua S. Lum. "Utilizing Robot Operating System (ROS) in robot vision and control". Thesis. Monterey, California: Naval Postgraduate School, set. de 2015. URL: https://calhoun.nps.edu/handle/10945/47300 (acedido em 30/05/2017).

[5] *Clearpath Robotics: Autonomous Mobile Robots*. URL: https://www.clearpathrobotics.com/ (acedido em 30/05/2017).

[6] *Husky UGV - Outdoor Field Research Robot by Clearpath*. URL: https://www.clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/ (acedido em 30/05/2017).

[7] *Mobile robot GUARDIAN*. URL: http://www.robotnik.eu/mobile-robots/guardian/ (acedido em 30/05/2017).

[8] *The CARLoS Project*. URL: http://carlosproject.eu/who (acedido em 30/05/2017).

[9] *VINBOT*. Abr. de 2014. URL: http://vinbot.eu/ (acedido em 30/05/2017).

[10] W. Domski, A. Mazur e M. Cholewinski. "Factitious force method in control of skid-steering platforms with rare constraints in motion". Em: *2016 21st International Conference on Methods and Models in Automation and Robotics (MMAR)*. Ago. de 2016, pp. 664–669. DOI: 10.1109/MMAR.2016.7575215.

[11] Tianmiao Wang et al. "Analysis and Experimental Kinematics of a Skid-Steering Wheeled Robot Based on a Laser Scanner Sensor". Em: *Sensors (Basel, Switzerland)* 15.5 (abr. de 2015), pp. 9681–9702. ISSN: 1424-8220. DOI: 10.3390/s150509681. URL: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC4481911/ (acedido em 30/05/2017).

[12] *Webots documentation: Using the Pioneer 3-AT and Pioneer 3-DX robots*. URL: https://www.cyberbotics.com/doc/guide/using-the-pioneer-3-at-and-pioneer-3-dx-robots (acedido em 30/05/2017).

[13] *Coordinated multi-robot exploration: Out of the box packages for ROS - IEEE Xplore Document*. URL: http://ieeexplore.ieee.org/document/7063639/ (acedido em 30/05/2017).

[14] Zhi Yan et al. "Team Size Optimization for Multi-robot Exploration". en. Em: *Simulation, Modeling, and Programming for Autonomous Robots*. Springer, Cham, out. de 2014, pp. 438–449. DOI: 10.1007/978-3-319-11900-7_37. URL: https://link.springer.com/chapter/10.1007/978-3-319-11900-7_37 (acedido em 30/05/2017).

[15] *wifi_comm - ROS Wiki*. URL: http://wiki.ros.org/wifi_comm (acedido em 05/07/2017).

[16] R. Reid et al. "Cooperative multi-robot navigation, exploration, mapping and object detection with ROS". Em: (2013), pp. 1083–1088. ISSN: 1931-0587. DOI: 10.1109/IVS.2013.6629610.

[17] L. Joseph. *Mastering ROS for Robotics Programming*. Packt Publishing, 2015.

[18] Vanessa Mazzari. *Robotic simulation scenarios with Gazebo and ROS*. Fev. de 2015. URL: http://www.generationrobots.com/blog/en/2015/02/robotic-simulation-scenarios-with-gazebo-and-ros/ (acedido em 03/07/2017).

[19] *Gazebo*. URL: http://gazebosim.org/ (acedido em 30/05/2017).

[20] *Simulating Jackal*. URL: http://docs.ros.org/indigo/api/jackal_tutorials/html/simulation.html (acedido em 30/05/2017).

[21] *Arduino - Introduction*. URL: https://www.arduino.cc/en/Guide/Introduction (acedido em 30/05/2017).

[22] *Arduino UNO*. URL: http://www.arduino.org/products/boards/arduino-uno (acedido em 30/05/2017).

[23] *Arduino NANO*. URL: http://www.arduino.org/products/boards/arduino-nano (acedido em 30/05/2017).

[24] *Raspberry Pi - Teach, Learn, and Make with Raspberry Pi*. URL: https://www.raspberrypi.org/ (acedido em 30/05/2017).

[25] *Products - Raspberry Pi*. URL: https://www.raspberrypi.org/products/ (acedido em 17/06/2017).

[26] *RaspberryPI models comparison | Comparison tables - SocialCompare*. URL: http://socialcompare.com/en/comparison/raspberrypi-models-comparison (acedido em 30/05/2017).

[27] *Rotary Encoder G40B-6-400-2-24 | Elektrologi*. URL: http://elektrologi.kabarkita.org/rotary-encoder-g40b-6-400-2-24/ (acedido em 30/05/2017).

[28] *Arduino - Wire*. URL: https://www.arduino.cc/en/Reference/Wire (acedido em 30/06/2017).

[29] *rosserial - ROS Wiki*. URL: http://wiki.ros.org/rosserial (acedido em 06/07/2017).

[30] *kinetic/Installation - ROS Wiki*. URL: http://wiki.ros.org/kinetic/Installation (acedido em 29/06/2017).

[31] *nfs.gif*. URL: http://www.redhatlinuxsysadmin.com/redhat-linux-system-administration/module5/images/nfs.gif (acedido em 04/07/2017).

[32] *SSH Protocol*. URL: https://www.ssh.com/ssh/protocol/ (acedido em 29/06/2017).

[33] *joy - ROS Wiki*. URL: http://wiki.ros.org/joy (acedido em 30/06/2017).

[34] Donald Knuth. *Knuth: Computers and Typesetting*. URL: http://www-cs-faculty.stanford.edu/\~{}uno/abcde.html.

[35] *Gazebo : Tutorial : Building Editor*. URL: http://gazebosim.org/tutorials?tut=building_editor (acedido em 30/05/2017).