ISEL – Instituto Superior de Engenharia de Lisboa
ADEETC – Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

LEIM
Licenciatura em Engenharia Informática e Multimédia
Unidade Curricular de Projeto

Android BoatCOM

Nuno Brás (A43952)

Carlos Viegas (A45092)

Orientadores

Professor Doutor Carlos Gonçalves (Instituto Superior de Engenharia de Lisboa)

Professor Doutor Mário Assunção (Escola Superior Náutica Infante D. Henrique)

Setembro, 2020

Resumo

Sea2Future é um projeto da Escola Superior Náutica Infante D. Henrique, o qual tem sido desenvolvido em torno da construção de uma Embarcação de Superfície não Tripulada (na língua anglo-saxofónica *Unmanned Surface Vehicle - USV*) com o nome *USV-enautica1*. A execução do mesmo visa a aquisição de competências e o desenvolvimento de futuros trabalhos inseridos nas áreas de *design* e construção de cascos de pequena dimensão, em materiais compósitos, propulsão elétrica, instrumentação e robótica, comunicações de dados Terra-Mar, etc.

Estando integrada no projeto *Sea2Future*, foi desenvolvida uma aplicação para o sistema operativo *Android* que permite controlar um barco remotamente. A execução da mesma foi dividida em 3 principais constituintes: a interface gráfica, a comunicação com o barco e o controlo do mesmo.

A interface gráfica é constituída por *widgets* que permitem ao utilizador fazer uma monitorização dos dados da telemetria da embarcação, juntamente com o controlo da mesma.

O *Robot Opertaing System* foi a plataforma utilizada para estabelecer uma comunicação com a embarcação, sendo esta a responsável pela receção e interpretação da informação captada pelos seus sensores assim como da que é enviada a partir de um *joystick* integrado na interface gráfica da aplicação.

Ao nível da comunicação, tendo já sido exploradas as diferentes opções que possibilitam o envio de dados bidirecionalmente no projeto precedente, seguiu-se o mesmo caminho e optou-se por utilizar uma ligação *Wi-Fi* de forma a facilitar a implementação da mesma e de maneira a que exista coesão entre os dois pontos da comunicação.

Em suma, mantendo a embarcação e o dispositivo móvel dentro do alcance máximo do protocolo *802.11* de 100 metros, assume-se que seria possível manter uma responsividade aceitável do barco, em função dos comandos recebidos, e que a comunicação se manteria satisfatória.

Palavras-chave: `Android`, `Robot Operating System`, Embarcação de Superfície não Tribulada.

Abstract

Sea2Future is a project by Escola Superior Náutica Infante D. Henrique, which is being developed around the construction of a unmanned surface vehicle with the name *USV-enautica1*. Its execution is aimed at acquisition of skills and at the development of future works inserted in related areas.

Integrated in the *Sea2Future* project, it was developed an Android application that allows the user to control the vessel remotely. Its execution was divided into 3 main parts: the graphic interface, communication and control. The graphic interface allows the user to monitor and control the vessel through the data that is shared between the two.

The `Robot Operating System` was the platform used to maintain communication with the vessel, which is responsible for receiving and interpreting the information captured by its sensors as well as the information sent from the application.

In a related project, there were different option taken into account that would enable bidirectional data transfer, but ultimately only one was chosen, therefore the same path was followed as it was decided to use a Wi-Fi connection in order to facilitate the implementation of the communication system and maintain cohesion between communication points.

In short, keeping the vehicle and the mobile device within the maximum reach of the 100 meters enabled in the 802.11 protocol, we assume that it would be possible to maintain an acceptable responsiveness of the boat, when being controlled, and that the communication would remain satisfactory during the transfer of data.

Keywords: `Android`, `Robot Operating System`, Unmanned Surface vehicle.

Agradecimentos

Embora este projeto esteja apenas sob os nomes de Carlos Viegas e Nuno Brás, o seu desenvolvimento e conclusão foi uma conquista bastante importante para nós em qual foi essencial o envolvimento e apoio de todos os que estiveram diretamente ou inderetamente ligados a ele.

Em primeiro lugar, agradecemos aos nossos orientadores Prof. Doutor Carlos Gonçalves e Prof. Doutor Mário Assunção pelo seu tempo, paciência e por estarem ativamente disponíveis para ajudar com quaisquer problemas que foram ocorrendo, tanto durante o período de confinamento, via *Zoom*, como durante o período de desconfinamento, em reuniões presenciais na Escola Superior Náutica Infante D. Henrique.

Ao colega Frederico Cardoso, pela ajuda na compreensão e utilização de temas que foram necessários para o desenvolvimento da aplicação, numa fase onde ainda nos encontrávamos sem rumo.

Eu, Nuno Brás, agradeço aos meus pais, Paula e José, assim como aos meus tios, João e Salomé, pelo apoio que me deram durante todo o meu percurso académico e por me oferecerem condições para a conclusão deste projeto.

Eu, Carlos Viegas, agradeço aos meus pais, Amadeu e Berta, e à minha namorada, Mariana Mendes, por todo o apoio, atenção, preocupação e paciência durante o desenvolvimento do projeto e durante todo o meu percurso académico, tendo sido uma grande fonte de ajuda e motivação para a conclusão da licenciatura.

Índice

Lista de Figuras

Lista de Tabelas

Lista de Código

Lista de Acrónimos

AIS *Automatic Identification System*

API *Application Programming Interface*

ENIDH Escola Superior Náutica Infante D. Henrique

GPS Sistema de Posicionamento Global

IP *Internet Protocol*

ISEL Instituto Superior de Engenharia de Lisboa

JSON *JavaScript Object Notation*

ROS *Robot Operating System*

SDK *Software Development Kit*

SSH *Secure Shell*

UML *Unified Modeling Language*

USV *Unmaned Surface Vehicle*

Capítulo 1

Introdução

A *USV-enautica1*, *Unmanned Surface Vehicle - enautica1*, é uma embarcação da Escola Superior Naútica Infante D. Henrique, ENIDH, que está em fase de construção. Através do projeto *Sea2Future*, em parceria com o Instituto Superior de Engenharia de Lisboa, ISEL, foram apresentadas propostas para a Unidade Curricular de Projeto desde 2018 com intuito de, juntamente com os ideiais e objetivos da ENIDH, enriquecer investigações científicas, entre outras aplicações futuras.

O projeto *Android BoatCOM* dá seguimento ao projeto *Boat Communication - BoatCOM* desenvolvido por Frederico Cardoso no ano letivo de 2018/2019, sendo desenvolvido em torno da embarcação *USV-enautica1*. Esta, considerada uma embarcação autónoma, tem presente um conjunto de módulos que contribuem para o correto funcionamento da mesma, constituído por um Computador principal, um Sistema de comunicação, um Sistema de controlo remoto, um Sistema de navegação e orientação, um Sistema de gestão de energia, um Sistema de propulsão, um Sistema de Deteção de colisões e um Sistema de Controlo Dinâmico, representados na figura 1.1.

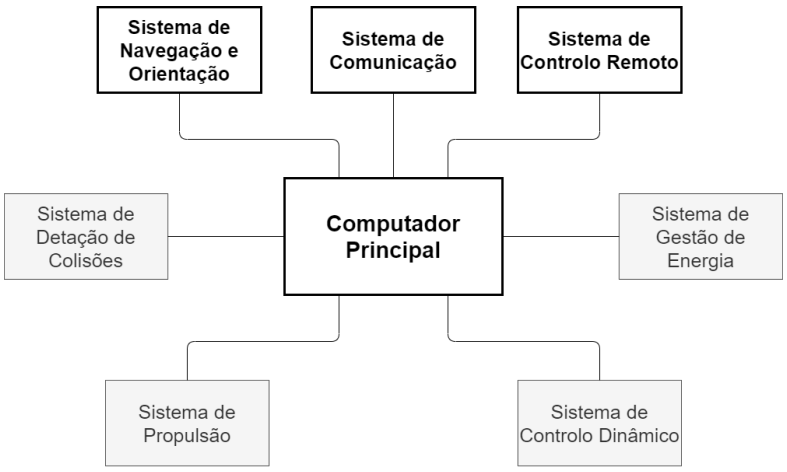

Figura 1.1: Módulos da embarcação *USV-enautica1*

1.1 Objetivos

Neste projeto iremos dar foco ao Sistema de Comunicação, Sistema de Navegação e Orientação e ao Sistema de Controlo Remoto de forma a implementar uma solução leve e de fácil acesso aos utilizadores. A solução consiste numa aplicação móvel para dispositivos `Android`, de forma a existir uma leitura de dados da telemetria recebidos da embarcação, tal como o controlo da mesma através de um *joystick* virtual.

Tendo em conta que a implementação da comunicação entre sistemas do barco é feita através do computador principal a bordo, sendo este um `Raspberry Pi 3 Model B+` que implementa o sistema `Robot Operating System`, no decorrer deste projeto vamos utilizar o mesmo sistema operativo, o que nos permite comunicar com o computador principal de forma versátil e eficaz.

Na interface gráfica iremos implementar componentes de telemetria que nos ajudam a visualizar valores de maneira mais prática. É também de referir que no mapa irá ser possível representar a totalidade do trajeto da embarcação.

1.2 Organização do Relatório

O objectivo deste relatório é apresentar detalhadamente o desenvolvimento do projeto *Android BoatCOM* no que toca ao estudo e pesquisa da informação

que foi necessária para a sua realização, o processo de implementação com base no conhecimento obtido e o relato das validações e testes após a sua conclusão.

O resto do relatório está organizado em seis partes. As quatro primeiras partes dizem respeito ao resumo dos objetivos e contextualização do projeto, capítulo de Introdução 1, à relação do mesmo com outros trabalhos, capítulo de Trabalho Relacionado 2, a visualização abstrata das arquiteturas propostas, capítulo de Modelo Proposto 3 e, finalmente, a sua implementação, capítulo de Implementação do Modelo 4. As duas partes seguintes, numa visão em que o projeto já se encontra desenvolvido, referem-se à colocação da aplicação à prova, capítulo de Validação e Testes 5, conclusões e a sua visão futura com base no que já foi alcançado,, capítulo de Conclusões e Trabalho Futuro 6.

O código desenvolvido para este projeto foi adicionado a um repositório [1] da plataforma `GitHub`[3], que se encontra no seguinte link.

Capítulo 2

Trabalho Relacionado

O conceito para a aplicação a ser criada surgiu como uma melhoria, em vários aspectos, de outro já existente. Ao longo do desenvolvimento do projeto *Android BoatCOM*, foram utilizadas várias ferramentas e bibliotecas externas. Neste capítulo iremos detalhar o projeto posterior a este, secção 2.1, o sistema operativo para estabelecer a comunicação entre a embarcação *USV-enautica1* e o sistema `Android`, denominado de `ROS`[5], secção 2.2, algumas bibliotecas que ajudaram na implementação desta comunicação desejada, secção 2.3, tal como todas as *API*, Interface de Programação de Aplicações, e *widgets* utilizados para a implementação da interface gráfica, secção 2.4.

2.1 Desktop BoatCom

O projeto *Android BoatCOM* surgiu no seguimento do projeto *BoatCom*, que foi desenvolvido anteriormente[2] para ambiente *desktop*, figura 2.1. Por apenas ser executado em contexto de *desktop* e com a utilização de um *joystick* para se realizar o controlo remoto da embarcação, este projeto está a ser desenvolvido como um *upgrade* em relação ao projeto anterior por providenciar ao utilizador uma portabilidade mais facilitada, um melhor acesso em caso de distribuição, através de plataformas como o *Google Play*, e por não ser necessário qualquer outro dispostivo para além do dispositivo móvel `Android`.

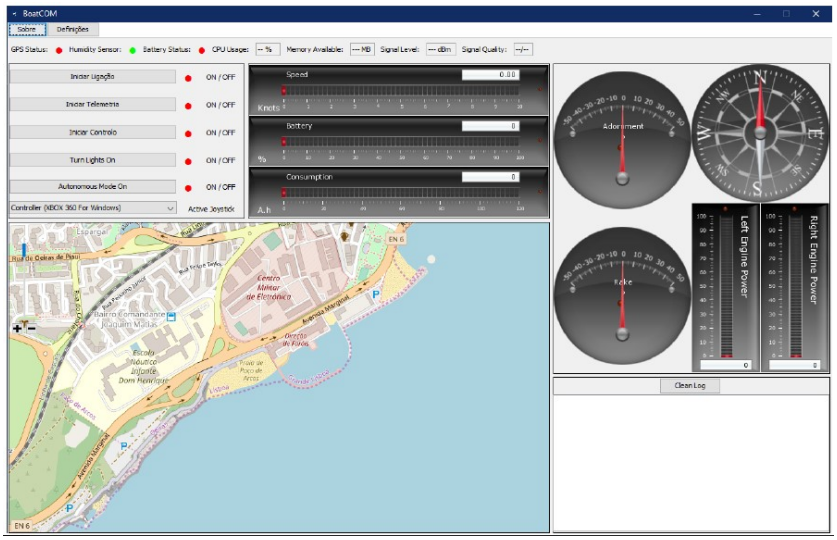

Figura 2.1: *Layout* do *Desktop BoatCom* [2]

Como podemos observar na figura 2.1, no canto superior esquerdo é apresentado um conjunto de botões que permitem a ativação e a desativação das funcionalidades da aplicação, ao lado são apresentados diversos mostradores, denominados de *gauges*, onde serão representados valores da bateria, inclinação e consumo, por exemplo, a aplicação é ocupada maioritariamente pelo mapa onde será indicada a posição atual da embarcação e, finalmente, no canto inferior direita são descritos os eventos que ocorrem durante a utilização desta aplicação.

2.2 Robot Operating System

Visto que a plataforma utilizada para estabelecer a comunicação com a embarcação foi o Robot Operating System (ROS [5]), segue-se uma breve explicação do funcionamento do mesmo e de como é possível estabelecer um canal de comunicação entre dois elementos do sistema.

2.2.1 Arquitetura do Robot Operating System

O ROS [5] é um conjunto de bibliotecas *open source* que ajudam na criação de aplicações para robôs nomeadamente passagem de mensagens entre processos e controlo de dispositivos de baixo nível. Atendendo à forma de como é

estabelecida a comunicação, o ROS [5] tem uma arquitectura própria que permite a troca de mensagens de forma simples através do protocolo TCP/IP 802.11. Esta arquitectura é formada por nós, onde cada nó poderá publicar e/ou subscrever a vários tópicos, como podemos observar na figura 2.2.

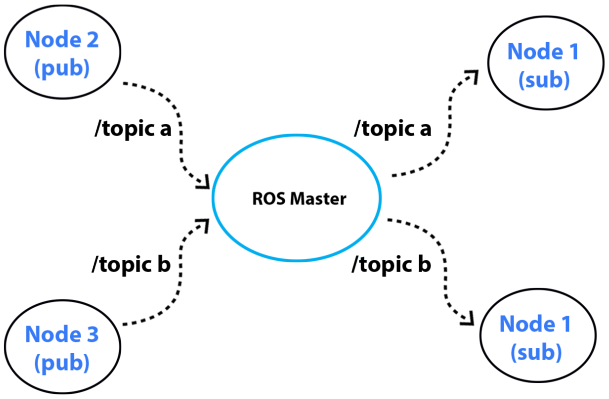

Figura 2.2: Diagrama da arquitectura de comunicação do ROS

Podemos, de forma a simplificar, dizer que um nó representa uma localização remota na rede local e um tópico representa uma variável a qual qualquer nó poderá aceder. Denomina-se por "**subscrição**"o acesso a um tópico de um nó em modo leitura, e de "**publicação**"o acesso a um tópico de um nó em modo escrita. Para existir uma maior organização, encontra-se no meio desta estrutura um nó principal denominado de ROS_MASTER que gere todas as localizações dos nós e tópicos a serem publicados e subscritos. A comunicação inicia-se então por criar o nó principal, ROS_MASTER, e todas as ligações subsequentes irão estabelecer um canal de comunicação com este nó.t Um nó que queira publicar um tópico estabelece um canal de comunicação com o ROS_MASTER, o qual guarda a informação de qual o tópico a ser publicado e a localização do nó que o publica. Por sua vez, um nó que queira subscrever a um tópico estabelece um canal de comunicação com o ROS_MASTER, informando-o de qual o tópico subscrito. O ROS_MASTER por si responde com a localização de todos os nós que publicam o tópico subscrito.

2.3 Android Core

Dada uma procura de como integrar o ROS na aplicação a desenvolver neste projeto, foi estudada a biblioteca Android Core[7]. Esta biblioteca proporciona uma colecção de aplicações que demonstram diversas implementações de comunicação com um sistema ROS e que são de grande utilidade para estabelecer a comunicação desejada.

2.3.1 Vantagens do Android Core sobre Rosbridge

Atendendo que já existia uma aplicação em contexto *Desktop*[2] do projeto *Android BoatCOM*, é de notar que a aplicação *Desktop* dava uso a outra biblioteca para estabelecer um canal de comunicação, denominada de rosbridge_suite[6]. Considerando as duas bibliotecas, colocou-se em questão qual seria a mais vantajosa para o desenvolvimento da aplicação. Analisando a documentação de cada biblioteca podemos focar pontos de interesse que nos puderam ajudar a decidir qual o caminho a tomar.

Relativamente à biblioteca rosbridge_suite[6], esta dá uso a uma API JSON[22] para integrar funcionalidades do ROS[5] em programas que não aplicam um sistema ROS[5].

> *"Rosbridge provides a JSON API to ROS functionality for non-ROS programs. There are a variety of front ends that interface with rosbridge, including a WebSocket server for web browsers to interact with."* [6]

É também de notar que esta biblioteca utiliza o seu próprio protocolo e implementação para estabelecer a comunicação, nomeadamente a criação de um WebSocket.

Relativamente à biblioteca Android Core, esta proporciona uma colecção de componentes e exemplos úteis para o desenvolvimento de uma aplicação que dá uso ao sistema ROS [5], sendo que é baseada na biblioteca rosjava [9] que serve como uma implementação do ROS [5] em Java [8], com suporte para Android [4].

> *"android_core is a collection of components and examples that are useful for developing ROS applications on Android."* [7]

Assim, foi utilizada a biblioteca `Android Core`[7] visto que nos oferece uma colecção de exemplos e ferramentas para estabelecer a comunicação desejada e por não depender de um `WebSocket` para a comunicação, ao contrário da biblioteca `rosbridge_suite`[6].

2.3.2 Uso do `Android Core`

Entre os exemplos da colecção de projetos da biblioteca `Android core`[7], existem dois aos quais foi dado mais uso, sendo estes o `android_tutorial_pubsub` e o `android_tutorial_teleop`. O primeiro disponibiliza um exemplo simples do uso de algumas classes que permitem a publicação e subscrição de tópicos e o segundo disponibiliza um exemplo de uma publicação de um tópico conforme a movimentação de um *joystick* virtual. Para testar os dois projectos apenas foi necessário alterar o nome dos tópicos a serem publicados e subscritos, visto que estes tinham nomes pré-definidos.

2.4 Interface Gráfica

A interface gráfica é o conceito da forma de interação entre o utilizador e aplicação. De forma a que esta interação seja o mais simples, rápida e intuitiva, foram agrupados um conjunto de plataformas e ferramentas que permitem a adição de elementos gráficos e outros indicadores visuais que facilitem a utilização da aplicação.

Na secções seguintes irão ser apresentadas de forma sucinta as bibliotecas que constituem a interface gráfica da aplicação. Na subsecção 2.4.1 será apresentada a plataforma utilizada para a autenticação dos utilizadores e armazenamento dos seus dados e, na subsecção 2.4.2, serão apresentadas as ferramentas utilizadas para a Monitorização, Localização e Controlo da embarcação.

2.4.1 Autenticação

Neste projeto é utilizado um sistema de autenticação somente para evitar conflitos quando múltiplos utilizadores se conectam à mesma embarcação. Se mais ninguém estiver ligado, tem-se controlo total sobre a embarcação, isto é, monitorização, localização e controlo, mas ao ocorrerem novas conexões,

estes utilizadores terão o acesso negado ao *joystick* para evitar que múltiplos comandos sejam enviados para a embarcação em simultâneo.

Firebase

`Firebase`[14] é uma plataforma desenvolvida pela *Google* para a criação de aplicações móveis e para a *web*. Esta plataforma tem uma funcionalidade que permite ligar um projeto de `Firebase`[14] a um projeto de `Android Studio`[16], fazendo a adição do *SDK* básico do `Firebase`[14]. Dentro dos múltiplos tipos de autenticação disponíveis escolhemos a da *Google* para tornar o início de sessão mais flexível e rápido.

2.4.2 *Widgets*

Um *widget* trata-se de uma pequena aplicação inserida na interface gráfica com funcionalidades a ela associada. Neste projeto foram utilizados 4 principais *widgets* provenientes de bibliotecas externas ao *Android Studio*.

Gauge

`Gauge` [10] é uma bilioteca de *GitHub* que permite a criação de um mostrador personalizado ao nível de aspecto, escala e tipos de valores, o que a torna numa ferramenta flexível para a apresentação de qualquer tipo de grandeza.

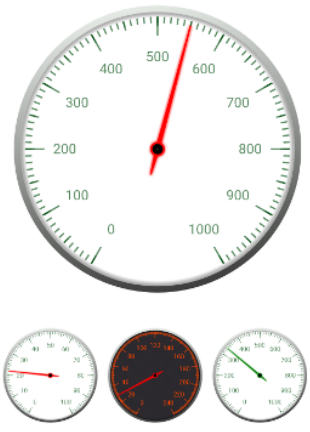

Figura 2.3: Exemplos da biblioteca `Gauge` [10]

MapView (GoogleMaps)

`MapView` [11] é um componente disponibilizado pelo `Android Studio` que permite apresentar um mapa com a informação obtida pelos serviços do *Google Maps*, figura 2.4. Entre os dados que são enviados pela embarcação, estão a sua latitude e longitude. Assim, utilizando esses mesmos valores, é possível atualizar a aplicação em tempo real com a localização da embarcação.

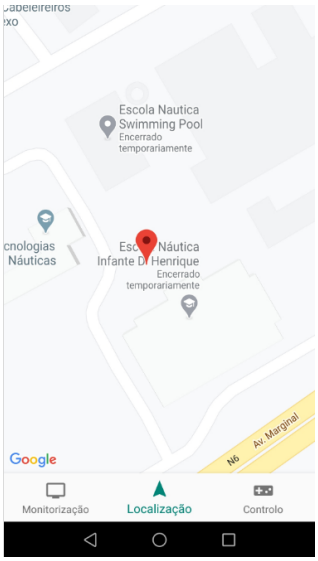

Figura 2.4: Exemplo do `MapView` [11]

JoystickView

Tendo como referência o projeto *BoatCom* original mencionado na secção 2.1 em que se utilizou um *joystick* físico para o controlo da *USV-enautica1*, foi utilizada a biblioteca `JoystickView` [12] do *GitHub* que permite a simulação de *joystick* virtual para `Android`, figura 2.5.

Uma das vantagens deste *widget* é a facilidade de personalização do aspecto do mesmo, o que permite uma fácil adequação a qualquer tema/design da aplicação.

Os movimentos são classificados pelo **ângulo**, obtido pela regra de um transferidor no sentido anti-horário, e pela **força**, obtida pela distância da zona pressionada ao ponto central do *joystick*.

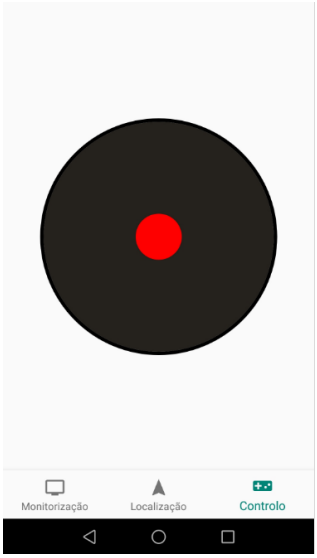

Figura 2.5: Exemplo do `JoystickView` [12]

SeekBar

O *widget* `SeekBar` [13] é uma extensão de uma barra de progresso do `Android Studio` que se destaca por permitir aos utilizadores controlarem o seu valor atual, fazendo com que funcione como um *slider*, figura 2.6. Esta ferramenta é utilizada para que o utilizador possa facilmente alterar a velocidade da embarcação em tempo real, sendo então capaz de fazer mudanças bruscas no movimento, sendo que o mesmo não seria possível utilizando outras alternativas como botões de aumento/redução, por exemplo. Esta alternativa foi priorizada em relação a outras também pelo facto de se assemelhar a uma alavanca real.

Figura 2.6: Exemplo do `Seekbar` [13]

Capítulo 3

Modelo Proposto

De forma a conseguir desenvolver um modelo capaz de satisfazer os requisitos, é necessário apontá-los e analisá-los previamente. Sendo assim, segue-se uma análise dos requisitos funcionais e não-funcionais da aplicação, secção 3.1 juntamente com os casos de utilização, secção 3.2. Nas ultimas duas secções, temos a maneira de como o `ROS`[5] foi implementado na aplicação, secção 3.3, e os modelos iniciais, ou `Moqups` [17], da interface gráfica, secção 4.3.

3.1 Requisitos

Os requisitos são condições que a aplicação deve preencher, sendo estes funcionais, subsecção 3.1.1, ou não funcionais, subsecção 3.1.2.

3.1.1 Requisitos Funcionais

Requisitos funcionais traduzem eventos, da perspectiva do utilizador, que o sistema faz, demonstrados na tabela 3.1

Tabela 3.1: Requisitos funcionais

Ref.#	Função	Categoria
R1.1	Apresentação em tempo real da localização da embarcação e trajeto efetuado	Evidente
R1.2	Apresentação de dados referentes à velocidade, nível de bateria, consumos de bateria, dos motores e valores de inclinação nos eixos X e Y	Evidente
R2.1	Controlo remoto da embarcação	Evidente

3.1.2 Requisitos Não Funcionais

Uma vez que os requisitos funcionais indicam o que o sistema faz, os requisitos não funcionais indicam como o sistema os faz, geralmente sendo ocultos ao utilizador. Estes encontram-se na tabela 3.2

Tabela 3.2: Requisitos não funcionais

Ref.#	Função	Categoria
R1	A interface gráfica deve ser implementada em Android	Evidente
R2	Comunicação bidirecional com a embarcação	Invisível
R3	A comunicação deve ser feita com Wi-Fi e a ligação feita diretamente ao `ROS`	Invisível

3.2 Casos de Utilização

Casos de utilização são narrativas que descrevem uma sequência de possíveis eventos entre dois actores na utilização de um sistema. Sendo assim, os casos de utilização da aplicação são, o pedido dos dados á embarcação pelo utilizador e o controlo remoto da embarcação pelo utilizador, ilustrados na

figura 3.1.

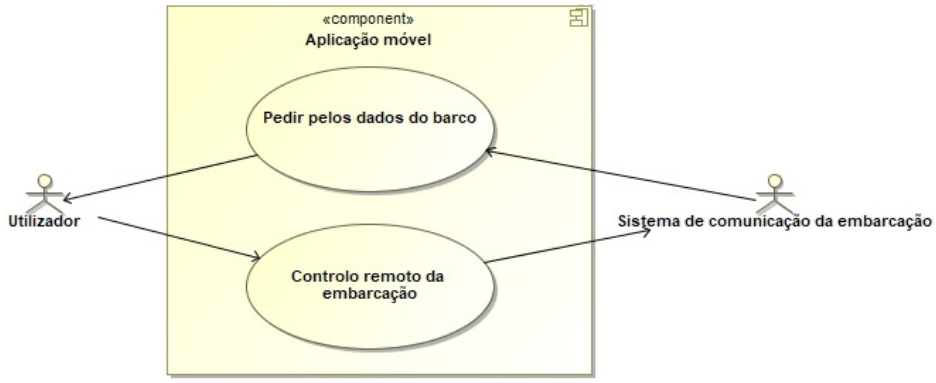

Figura 3.1: Casos de utilização da aplicação

Como podemos observar, os casos de utilização são apenas dois, no entanto requerem uma estrutura organizada para a sua eficácia. Assim, para um melhor funcionamento e em função do sistema já implementado na embarcação *USV-enautica1*, foi projectada uma arquitectura do sistema de comunicação que, juntamente com o sistema de comunicação da embarcação, representado na figura 3.2.

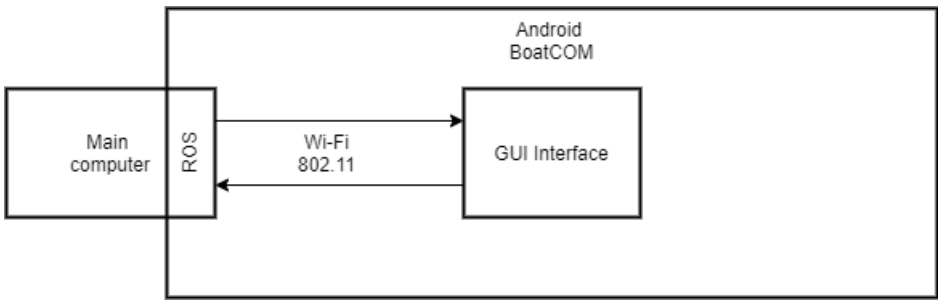

Figura 3.2: Modelo Abstracto do *Android BoatCOM*

Podemos observar então que existem apenas duas partes que intervêm na comunicação sendo estas o computador principal, que se encontra na embarcação e mantém o controlo imediato da embarcação, e a interface da aplicação.

3.3 Implementação do Robot Operating System proposta

Uma vez que o modelo implementado para a comunicação é baseado no ROS[5], foi desenvolvido um modelo que se adapta a este sistema. Como foi mencionado na secção 2.2.1 o ROS [5] funciona como um grupo de nós que publicam ou subscrevem a tópicos. O modelo proposto idealiza a implementação de um nó para cada subscrição ou publicação na aplicação de forma a comunicar com o nó principal , ROS_MASTER, que se encontra na embarcação, tal como na figura 2.2.

3.4 Interface Gráfica

Antes da implementação da aplicação, de forma a ser possível planear a estrutura da interface gráfica, foi desenhado um *mockup* utilizando a *web app* Moqups[17]. Um *mockup* é um modelo de um projeto que, neste caso, é utilizado para a demonstração do design que se pretende atingir. Assim sendo, o *mockup* da aplicação *Android BoatCom* encontra-se representada na figura 3.3.

Seguindo o fluxo de *layouts* representado na figura 3.3, a primeira página será utilizada para a realização da autenticação do utilizador, a seguinte permitirá que seja introduzido o endereço *IP* da embarcação, para que seja guardado e utilizado nas funcionalidades da aplicação e, finalmente, o utilizador terá acesso ao menu que permite o acesso às funções disponíveis, nomeadamente, Monitorização, Localização e Controlo. Tendo em conta que se trata de um *mockup*, o produto final da aplicação poderá apresentar alterações/adições face ao esboço inicial representado.

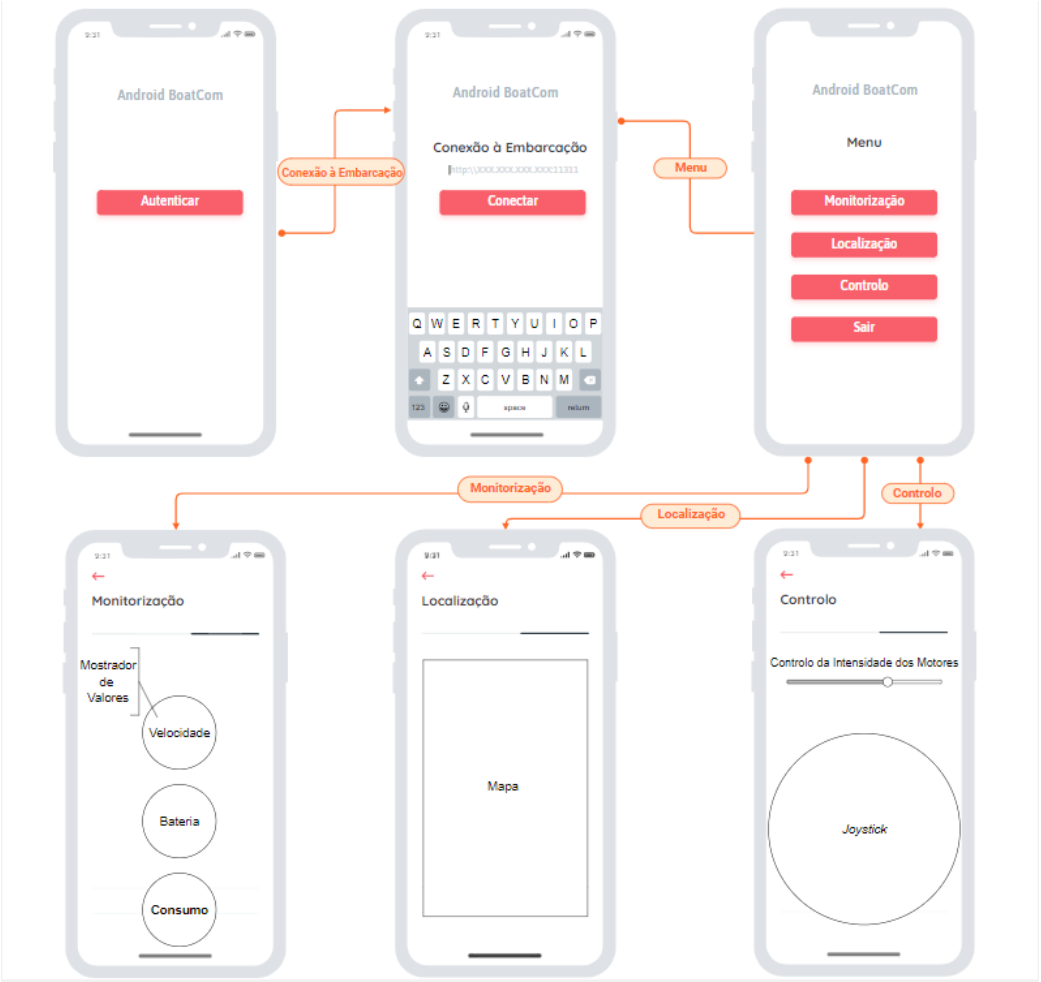

Figura 3.3: *Mokcup* da aplicação

Capítulo 4

Implementação do Modelo

Neste capítulo iremos desenvolver a forma de como foram implementadas cada fase do projeto, tal como fazer uma análise de qual as melhores escolhas para a implementação destas. Na secção 4.1, são representados os diversos *softwares* auxiliares utilizados para testar a comunicação entre a embarcação e a aplicação móvel e na secção 4.2, é apresentada a implementação de um simulador para a comunicação desejada. Por fim, na secção 4.3, são retratados os vários mecanismos utilizados para o tratamento de dados, tal como é descrito a implementação das componentes gráficas da aplicação.

4.1 *Software* auxiliar

Visto que a embarcação ainda não se encontra construída, foi necessário alguns componentes auxiliares para testar a comunicação entre a embarcação e a aplicação móvel. Portanto, para uma boa simulação da embarcação, foi instalado o `ROS` num ambiente virtual.

4.1.1 Distribuições do `Robot Operating System`

Uma das primeiras escolhas a serem feitas foi a distribuição do `ROS`[5] a ser instalada, visto que sensivelmente todos os anos é lançada uma nova distribuição. A diferença entre cada distribuição apenas afeta o suporte dado pela comunidade, uma vez que é um sistema operativo *opensource*, tal como alguns comandos disponíveis. Sendo assim, quando o projeto foi iniciado, existiam duas distribuições funcionais, uma mais recente, denominada por *Melodic Morenia* ou apenas *Melodic*, e outra denominada por *Kinetic Kame*

ou apenas *Kinetic*. O grupo achou por bem trabalhar com a distribuição mais antiga , *Kinetic*, visto que o suporte oferecido era maior.

4.1.2 *Ubuntu* (VMWare)

Dado que a distribuição escolhida apenas suporta sistemas operativos `Ubuntu`[19] versão 15.10 ou 16.04, o grupo decidiu trabalhar com a versão mais recente e conhecida como estável, 16.04. Foi então configurada numa máquina virtual, oferecida pela `VMWare Workstation`[20], o `Ubuntu`[19] 16.04 e configurado o `ROS` com a ajuda das páginas oferecidas no seu site oficial[5]. De seguida, para testar a instalação, foi instalada também a biblioteca *turtlesim* que permite controlar uma tartaruga virtual através da publicação de tópicos. Estes tópicos podem ser publicados em modo texto, mas de forma a ser mais eficiente esta biblioteca oferece também um simulador que publica tópicos através das quatro setas do teclado. Agora com o ambiente preparado, podemos testar a ligação através da biblioteca `Android Core`[7].

4.1.3 *PuTTY*

No seguimento do `Android Core`[7], após tentativas de conexão entre a aplicação móvel com o `ROS`[5] da máquina virtual sem sucesso, concluiu-se que, por ser uma maquina virtual, os dois dispositivos intervenientes na conexão não estavam dentro da mesma rede *Wi-Fi*, o que seria necessário para haver sucesso na conexão. Para contornar temporariamente o problema, usou-se o `PuTTY`[18] para criar um túnel *SSH* capaz de criar uma porta dentro do *localhost* que, sendo acedida pela aplicação móvel, esta seria então redirecionada para o endereço do `ROS`[5] na maquina virtual, estabelecendo assim uma ligação com sucesso.

Depois de novas investigações, foi possível configurar a placa de rede da máquina virtual para que esta utiliza-se um endereço *IP* na mesma rede que o *IP* do telemóvel, sendo então desnecessária a utilização do *software* `PuTTY`[18].

4.2 Simulador

Nesta secção vai ser explorada a importância do simulador, subsecção 4.2.1, tal como a sua implementação, subsecção 4.2.2.

4.2.1 Importância do Simulador

Uma vez que a embarcação com qual a aplicação deve comunicar não está construída, é importante a utilização de um simulador que consiga recriar uma situação real onde os dados são trocados entre os dois elementos do sistema. É nesta situação que o ROS [5] se destaca, pois permite garantir que, nas mesmas condições, se a comunicação funcionar com o simulador, há de certamente funcionar com a embarcação em si. Foi também muito importante, especialmente na situação corrente de confinamento, pois todo o trabalho tinha de ser feito a partir de casa.

4.2.2 Implementação do Simulador

Na criação do simulador, visto que não é necessário mensagens muito complexas, foi desenvolvido um *script* que permite criar um nó que publica e subscreve a um tópico, código 1.

Código 1: Simulador em *Python*

```
import rospy
from std_msgs.msg import Int16
from std_msgs.msg import String

def callback(data):
  rospy.loginfo(rospy.get_caller_id()+"I heard %s", data.data)

def run():

  #create publishers
  speed = rospy.Publisher('speed', Int16, queue_size=1)

  #create subscribers
  rospy.Subscriber('speed', Int16, callback)

  #init node
  rospy.init_node('sim', anonymous=False)

  rate = rospy.Rate(10)
```

```
21    #main loop
      while not rospy.is_shutdown():
23      speed.publish(45)
        rospy.sleep()
25
if __name__ == '__main__':
27    try:
        run()
29    except rospy.ROSInterruptException:
        pass
```

Este *script* dá uso à biblioteca `rospy` [21] que permite de maneira simples executar o pretendido. Podemos observar que para publicar um tópico basta criar um `Publisher` através do comando `rospy.Publisher`, que recebe como parâmetros o nome do tópico, neste caso "speed", seguido do tipo de dados do mesmo. O último parâmetro, `queue_size`, refere-se ao máximo de mensagens existentes no *buffer* até começarem a ser descartadas, isto porque a partir da distribuição *Hydro* do `ROS`[5] as mensagens publicadas começaram a tomar partido de um *buffer* para serem publicadas. Para iniciar uma subscrição de um tópico, dá-se uso da função do `rospy` [21], `rospy.Subscriber`, que tal como o `Publisher`, começa com o parâmetro do nome do tópico, seguido do tipo de dados do tópico a ser subscrito. O último parâmetro chama a função `callback` que cria um `log` da informação recebida.

De seguida é iniciado o nó através do comando `rospy.init.node` que tem como parâmetros o nome do nó, tal como o parâmetro *anonymous* que indica se este é anónimo ou não. No caso de ser anónimo, o nome é gerado aleatoriamente para evitar colisões, o que neste caso não é necessário. É definida também uma frequência de execução do nó, em 10Hz. Uma vez dentro do ciclo principal, é publicado o valor representativo de 45 através do comando `'node name'.publish('node value')`, seguido de uma chamada ao método `rospy.sleep('sleep time')` de forma a que exista um tempo de espera entre cada publicação.

Antes de podermos correr o *script* é necessário executar o nó principal, `ROS_MASTER`. Isto é possível através da consola do `Ubuntu`[19], executando o comando *roscore*. Correndo agora o *script* podemos confirmar que a sua execução é bem sucedida através dos comandos *rosnode list* e *rostopic list*. O primeiro indica quais os nós ligados ao `ROS_MASTER` e o segundo indica quais os tópicos a serem publicados, figura 4.1.

```
projeto@ubuntu:~$ rosnode list
/rosout
/sim
projeto@ubuntu:~$ rostopic list
/rosout
/rosout_agg
/speed
```

Figura 4.1: Confirmação de publicação do simulador

Podemos também observar, após a execução do *script*, o valor que a função *callback* retorna, confirmando a subscrição do tópico, figura 4.2.

```
projeto@ubuntu: ~/prj/catkin_ws/src/boat_sim/scripts
[INFO] [1599518775.396343]: /simI heard 45
[INFO] [1599518775.596934]: /simI heard 45
[INFO] [1599518775.807925]: /simI heard 45
[INFO] [1599518776.000857]: /simI heard 45
[INFO] [1599518776.202165]: /simI heard 45
[INFO] [1599518776.402895]: /simI heard 45
[INFO] [1599518776.604021]: /simI heard 45
[INFO] [1599518776.807420]: /simI heard 45
```

Figura 4.2: Confirmação de subscrição do simulador

4.3 Interface Gráfica

Tendo sido apresentado no capítulo 2 todas as ferramentas necessárias para a interface gráfica, neste capítulo irá ser explorado a maneira como estas foram implementadas.

4.3.1 Autenticação

Sendo que a autenticação da nossa aplicação é feita com recurso ao *Google SignIn*, foi necessário adicionar uma classe que fizesse essa ligação com base no `Firebase`[14].

Depois de alguma pesquisa, encontrou-se uma classe[15] no repositório do `Firebase`[14] com uma demonstração da autenticação fazendo recurso ao *Google ID Token*, sendo esta denominada de `GoogleSignInActivity` (Ver figura 4.3).

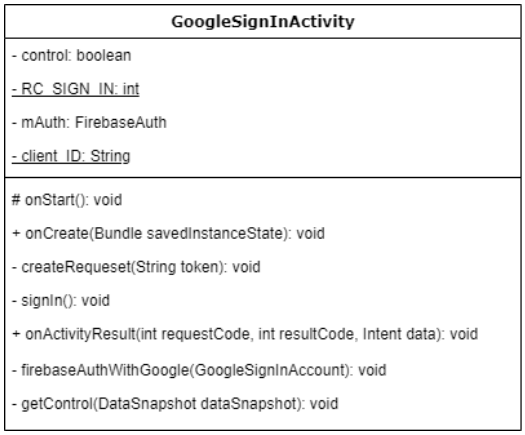

Figura 4.3: `UML` da classe `GoogleSignInActivity`

Google ID Token

Ao se tentar realizar um início de sessão utilizando esta classe, a primeira etapa a ser executada é a realização de um pedido para obter a *ID* do utilizador que se está a conectar, o seu *e-mail* e a informação básica do seu perfil *Google*. Para tal pedido ser realizado, é ncessário ter acesso ao *clientID* do servidor de *backend*, sendo que tal pode ser encontrado nas configurações do projeto de `Firebase`. Assim sendo, a função que suporta esta etapa é a `createRequest()`, a qual foi implementada tal como representado no código 2

Código 2: Pedido de *TokenID* do utilizador

```
private void createRequest(String token) {
    GoogleSignInOptions gso = new GoogleSignInOptions.
    Builder(GoogleSignInOptions.DEFAULT_SIGN_IN)
        .requestIdToken(token)
        .requestEmail()
        .build();
    mGoogleSignInClient = GoogleSignIn.getClient(this, gso);}
```

Login

Sendo o pedido realizado pela função `createRequest()` realizado com sucesso, ao ser executada a função de `signIn()`, o código que suporta as atualizações na aplicação com base no sucesso ou insucesso é o representado no código 3.

Código 3: Autenticação com `Firebase`

```
private void firebaseAuthWithGoogle(GoogleSignInAccount account)
    {

    AuthCredential credential = GoogleAuthProvider.
    getCredential(account.getIdToken(), null);
    mAuth.signInWithCredential(credential)
        .addOnCompleteListener(this,
        new OnCompleteListener<AuthResult>() {
         @Override
        public void onComplete(@NonNull Task<AuthResult> task) {
            if (task.isSuccessful()) {
                FirebaseUser user = mAuth.getCurrentUser();
                Intent i = new Intent(getApplicationContext(),
                Connect.class);
                startActivity(i);

                }
            else {
              Toast.makeText(GoogleSignInActivity.this,
              "Authentication has failed.", Toast.LENGTH_SHORT).
   show();
                        }
                    }
                });
```

4.3.2 Monitorização e Localização

Esta subsecção está divida em 2 partes, desenvolvidas pelas ordem de execução desde a ligação ao barco até à receção dos seus dados, sendo estes apresentados nos respetivos *widgets*.

Conexão à Embarcação

Após a autenticação do utilizador, para progredir, é necessária a introdução do `ROS_MASTER_IP`, isto é, o endereço *IP* do nó principal da embarcação, que neste caso, se trata da *USV-enautica1*. Este, ao ser introduzido, será guardado para depois ser efetuada a ligação ao computador principal da embarcação ao se tentar aceder a uma das 3 principais funcionalidades da aplicação. Cada uma das 3 classes que implementam a monitorização, localização e controlo (`MonitorFragment`,`LocalFragment` e `ControlFragment`)

herdam a classe `RosActivity` (Ver figura 4.4 proveniente da biblioteca `Android Core`[7].

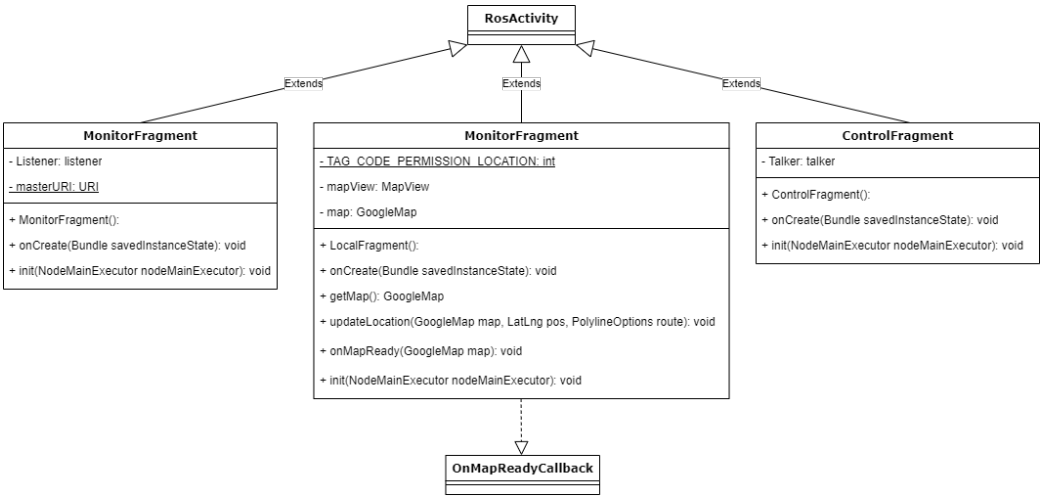

Figura 4.4: `UML` das classes `MonitorFragment`, `LocalFragment` e `ControlFragment`

Ao implementar a classe `RosActivity`, é automaticamente implementado o método `init()` representado em 4, cujo permite a ligação da aplicação à embarcação.

Código 4: Ligação com a embarcação

```
@Override
    public void init(NodeMainExecutor nodeMainExecutor) {
        listener = new Listener(this);

        NodeConfiguration nodeConfiguration = NodeConfiguration.
    newPublic(getRosHostname(), masterURI);

        nodeConfiguration.setNodeName("testing");

        nodeMainExecutor.execute(listener,nodeConfiguration);
    }
```

Analisando melhor o código representado em 4, este é constituído pela instanciação de um objeto do tipo `NodeConfiguration` que, para tal, recebe como parâmetros da função `NodeConfiguration.newPublic()` a função `getRosHostname()`, que representa o endereço *IP* do *host* da aplicação, que neste caso se trata de um telemóvel, e a variável `masterURI`, que representa o *ROS_MASTER_IP* introduzido pelo utilizador. No final deste mesmo excerto

de código, é chamada a função `execute()` que recebe a variável `listener`, uma classe onde serão definidos os nós da embarcação a serem subscritos, e a variável `nodeConfiguration` que contém a informação relacionada com a origem e o destino do pedido de conexão. Esta conexão, ao ser efetuada com sucesso, fará com que seja criada um nó no computador principal da embarcação, cujo nome é atribuído pela função `setNodeName()` apresentada também no código 4.

Subscrição e Atualização

A classe onde são definidos os nós que se pretende subscrever é a classe (Ver figura 4.5 baseada na classe `Listener` da biblioteca `Android Core`[7].

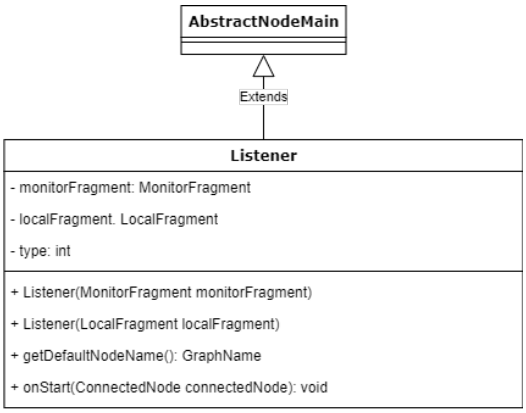

Figura 4.5: `UML` das classe `Listener`

As classes que fazem uso desta são a `MonitorFragment` e a `LocalFragment`, sendo que existem 2 construtores, 1 para cada uma deles ser definida, representados no código 5.

Código 5: Construtores de `Listener`

```
public Listener( MonitorFragment monitorFragment) {
    this.monitorFagment= monitorFragment;
    type=0;}
public Listener( LocalFragment localFragment) {
    this.localFragment= localFragment;
    type=1;}
```

A função principal desta classe é a `onStart()`, sendo que esta é chamada após a execução do método `execute()` do código 4. Os nós a serem subscritos são inicializados através de objetos do tipo `Subscriber`, sendo então necessário associar o seu nome e o tipo de valores que contém, sendo fulcral que sejam idênticos ao mesmo nome e tipo de variável atribuídos na embarcação, caso contrário não será possível a obtenção dos seus dados. Um exemplo da instanciação de um `Subscriber` encontra-se no código 6, sendo que se trata de um nó de nome `test` e do tipo `std_msgs/String`.

Código 6: Exemplo de instanciação de um `Subscriber`

```
Subscriber<String> subscriber = connectedNode.newSubscriber("
    test", "std_msgs/String");
```

Para que se possa obter os valores de um nó na aplicação, é preciso adicionar ao textttsubscriber um `MessageListener()` utilizando a função `AddMessageListener()`, o que nos permite indicar o que queremos fazer com os valores provenientes desse nó na nossa aplicação, sempre que são recebeidos (`onNewMessage()`). No âmbito desta aplicação, pretende-se atualizar em tempo real os mostradores da interface gráfica, sendo que um exemplo disso encontra-se representado no código 7.

Código 7: Exemplo de atualização de mostradores (`Gauge`)

```
subscriber.addMessageListener(new MessageListener<String>() {
    public void onNewMessage(String message) {
        Gauge g = monitorFragment.findViewById(R.id.gauge2);
        g.setValue(Float.parseFloat(message.getData()));
    }
});
```

O mesmo processo é utilizado para atualização da posição da embarcação no `MapView`[11], sendo que, em vez de se tratar de uma `std_msgs/String`, trata-se uma variável do tipo `sensor_msgs/NavSatFix`.

A obtenção dos valores de latitude e longitude a partir dos dados de *GPS* recebidos pela embarcação, é feita da forma representa em 8.

Código 8: Obtenção dos valores de Latitude e Longitude

```
 Subscriber<NavSatFix> subscriber = connectedNode.newSubscriber(
    "gps","sensor_msgs/NavSatFix");
subscriber.addMessageListener((new MessageListener<NavSatFix>()
    {
public void onNewMessage(NavSatFix message) {
```

```
4       NavSatFix gps = message;
        LatLng pos = new LatLng(gps.getLatitude(),gps.getLongitude()
        );
6       localFragment.updateLocation(localFragment.getMap(),pos);
        Global.mapCoords.add(pos);
8                   }
                }));
```

Com os valores de latitude e longitude obtidos, é então possível atualizar o curso que representa a atualização atual da embracação fazendo recurso ao código representado em 9.

Código 9: Atualização da posição da embarcação no mapa

```
1   public void updateLocation(GoogleMap map, LatLng pos,
      PolylineOptions route){
        map.moveCamera(CameraUpdateFactory.newLatLngZoom(pos,20));
3       map.addMarker(new MarkerOptions().position(pos));
        route = route.add(pos);
5       map.addPolyline(route);
        }
7
```

Para o desenho da trajetória no mapa, as posições recebidas pela embarcação guardadas em memória e recorre-se ao `Polyline`, uma biblioteca que permite desenhar linhas sobrepostas ao mapa. Portanto, a cada 2 posições em memória, é desenhada uma linha entre elas, tornando-se assim visível o movimento da embarcação durante tempo em que esteve ligado à aplicação.

Filtragem de erros na leitura de dados

Tendo noção da probabilidade de manifestação de erros na leitura dos dados, tal como a oscilação entre valores que os dados tomam, implementou-se uma técnica de filtragem mediana de forma a normalizar os valores apresentados ao utilizador. Esta filtragem baseia-se na utilização de uma memória do tipo *FIFO*, *first in, first out*, que em português significa que os primeiros valores a entrar serão os primeiros valores a serem removidos, para retirar apenas o valor que representa a mediana dos dados guardados em memória, sendo que este algoritmo pode ser visualizado na figura 4.6.

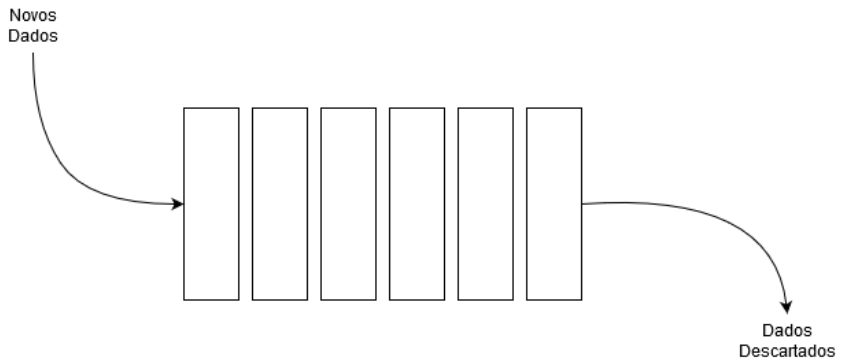

Figura 4.6: Memória FIFO

Por outro lado, para obter o tamanho desta memória, requer-se alguma atenção para não existir perda de dados na filtragem. Sabendo que uma filtragem mediana é mais eficaz quando o tamanho de casos em memória é diretamente proporcional ao número de ocorrências de erros, desenvolveu-se um mecanismo que ao analisar os dados recebidos altera o tamanho da memória FIFO em função da diferença entre o valor mais alto e o valor mais baixo dos da memória.

É de notar que este mecanismo, na ocorrência de um erro muito elevado, permite que a memória assuma valores muito elevados, dado que foram aplicados um mínimo e um máximo para o tamanho da memória. Visto que a frequência de transmissão do `ROS`[5] geralmente é de 10Hz, decidimos limitar o máximo da memória a 10 valores para não existir um atraso maior do que 0.5 segundos na leitura dos dados.

Segue-se então um gráfico que ilustra, num ambiente simulado, uma filtragem dos dados, que apesar de não ser perfeita, ajuda bastante o utilizador na sua interpretação.

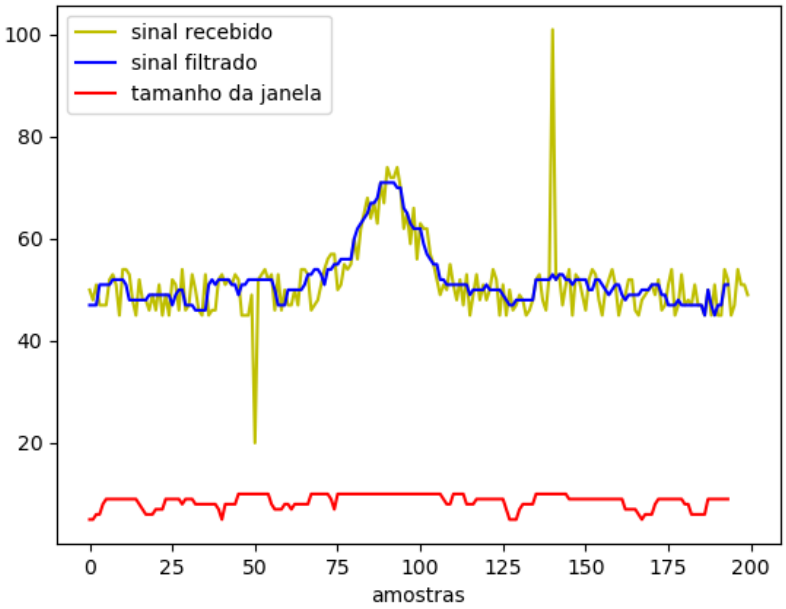

Figura 4.7: Gráfico da demonstração da aplicação do filtro mediano

Após a obtenção dos dados no tamanho da memória desejado (1), os valores da mesma serão reorganizados por ordem crescente (2) e o valor que irá ser apresentado no mostrador é o central (3). Um exemplo da aplicação deste algoritmo encontra-se na figura 4.8, estando representado em amarelo o qual irá ser escolhido e em vermelho o erro filtrado.

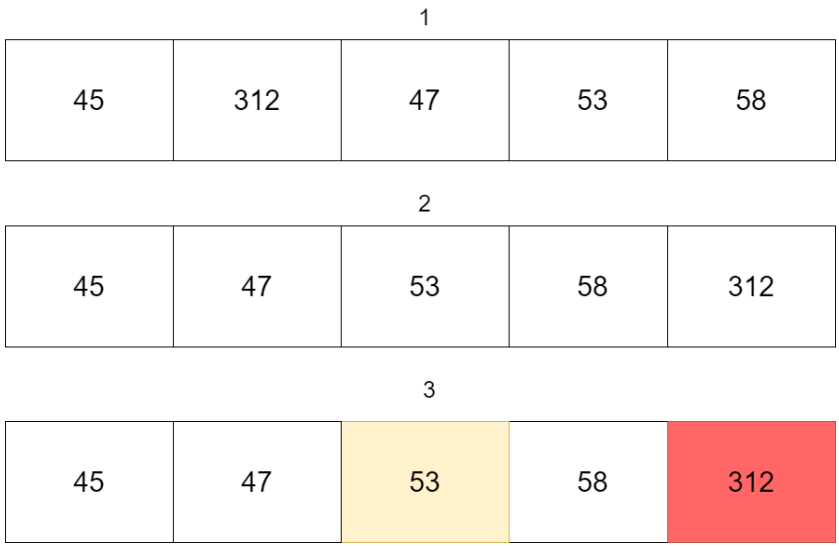

Figura 4.8: Aplicação do algoritmo

Este algoritmo foi traduzido para código (Ver 10) assumindo uma frequência de publicação de dados por parte da embarcação de 10 valores por segundo.

Código 10: Algoritmo de filtração de erros

```
public static float updateGauge(ArrayList<Float> gaugeValues){
    ArrayList<Float> newGauges = (ArrayList<Float>) gaugeValues
.clone();
        Collections.sort(newGauges);
        float high = newGauges.get(newGauges.size() - 1);
        float low = newGauges.get(0);
        diff = (int) (high - low);

        copiedGauges = new ArrayList<Float>(Arrays.asList(Arrays
.copyOfRange(((Float[])((ArrayList<Float>) gaugeValues.clone
()).toArray()),(int) (gaugeValues.size() - diff), gaugeValues
.size() - 1)));
        Collections.sort(copiedGauges);


        return copiedGauges.get((int) Math.floor(copiedGauges.
size() / 2f));
    }
```

4.3.3 Controlo

De forma a implementar o controlo, foi necessário perceber melhor a forma como a embarcação se direciona. Sendo que a embarcação tem dois motores

estáticos, o direcionamento da embarcação é dado pela diferença de potência entre cada um. Posto isto, a embarcação dirige-se na direção oposta ao motor com mais potência. Com este conhecimento, foi então implementado um mecanismo de regulação da potência para cada motor fazendo recurso ao *joystick* virtual para a realização do controlo juntamente com o *slider* da velocidade que permite definir a potência máxima a atingir pelos motores.

De forma a que se entenda melhor o impacto de cada movimento do *joystick* nos motores do barco, foi desenhado o diagrama representado na figura 4.9

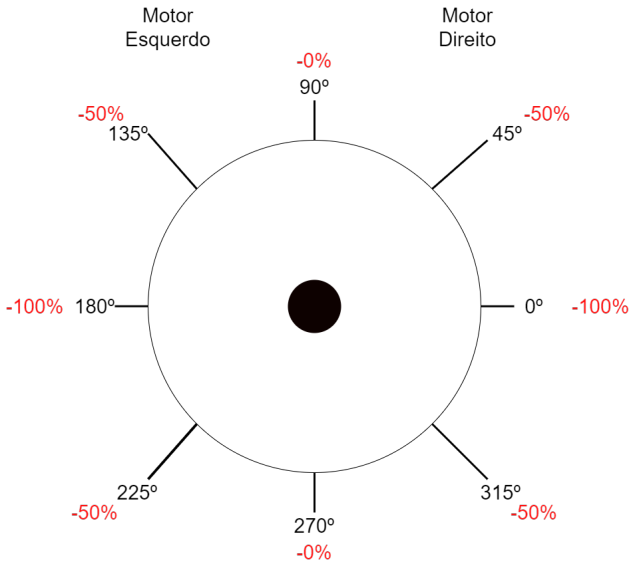

Figura 4.9: Impacto do *joystick* nos motores

O círculo preto da figura 4.9 representa o mesmo analógico que se poderá mover na aplicação, sendo que cada posição do mesmo dentro do círculo maior terá um ângulo associado. Este analógico, quando posicionado no semi-círculo direito, causará uma redução da percentagem máxima do motor direito conforme a escala dos valores apresentados a vermelho no diagrama, o que fará com que a embarcação se vire para a direita, sendo que a situação se espelha quando o analógico se encontra no semi-círculo esquerdo. Para fazer com que a embarcação se mova em linha reta, basta não mover o analógico ou mantê-lo tanto num ângulo de 90º como num de 270º, desde que a velocidade definida pelo *slider*[13] seja superior a 0%.

Publicação

Com o conhecimento obtido sobre o funcionamento dos motores e do *joystick* através do diagrama da figura 4.9, foi então possível implementar o mecanismo de controlo usando a classe `ControlFragment`.

O código de inicialização é semelhante ao apresentado no código 4, sendo que neste caso se usa um objeto da classe `Talker` (Ver figura 4.10), criada com base na classe `Talker` da biblioteca `Android Core`[7].

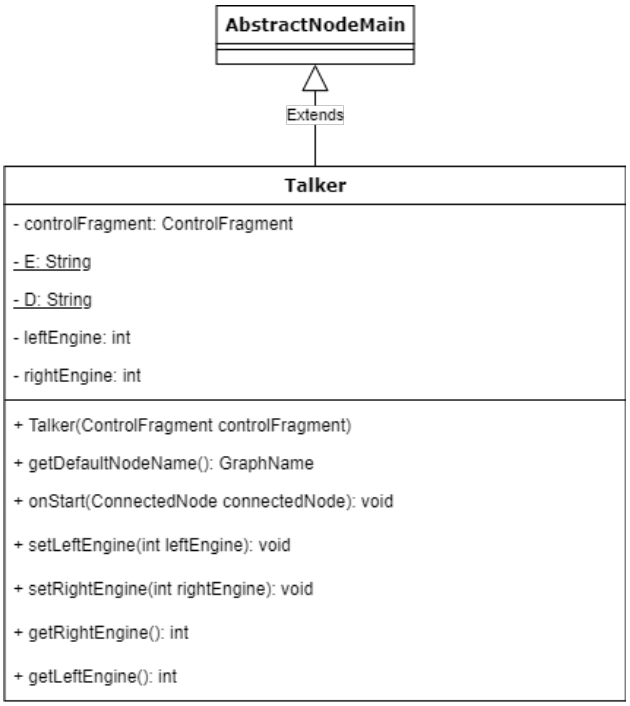

Figura 4.10: `UML` das classe `Talker`

Para que seja estabelecida a ligação com a embarcação, esta classe é utilizada na inicialização da classe `ControlFragment`, tal como se pode ver na figura 11.

Código 11: Inicialização da classe `ControlFragment`

```
1 public void init(NodeMainExecutor nodeMainExecutor) {
3     talker = new Talker(this);
5     NodeConfiguration nodeConfiguration = NodeConfiguration.
    newPublic(getRosHostname(),getMasterUri());
```

```
7       nodeConfiguration.setNodeName("testingTalker");
        nodeMainExecutor.execute(talker,nodeConfiguration);
9       }
```

A função principal da classe é o método `onStart()` pois é onde é realizada a publicação de valores. Em primeiro lugar, é necessária a criação de objetos do tipo `Publisher`, onde serão definidos quais os nós em que vão ser publicados valores provenientes do *joystick* e do *slider*. Um exemplo da instanciação desses objetos está representado no código 12.

Código 12: Instanciação de objetos `Publisher`

```
final Publisher<std_msgs.Int16> publisherE = connectedNode.
    newPublisher("left_engine", "std_msgs/Int16");
2 final Publisher<std_msgs.Int16> publisherD = connectedNode.
    newPublisher("right_engine", "std_msgs/Int16");
```

Após a função `execute()` ser executada, o método que estará sempre em execução durante o controlo da aplicação é o `loop`, onde são feitos os cálculos da potência atual de cada motor consoante o valor atual da *seekbar* e da posição atual do analógico do *joystick*, sendo que este se encontra implementado no código 13.

Código 13: Mecanismo de controlo da embarcação

```
protected void loop() throws InterruptedException {
2 final std_msgs.Int16 strE = (std_msgs.Int16)publisherE.
    newMessage();
final std_msgs.Int16 strD = (std_msgs.Int16)publisherD.
    newMessage();
4
joystickView.setOnMoveListener(new JoystickView.OnMoveListener()
    {
6   @Override
    public void onMove(int angle, int strength) {
8
    setLeftEngine(0);
10  setRightEngine(0);

12  if (angle >= 0 && angle <= 90){
        setRightEngine( 100 - angle * 100 / 90);
14      }
    else if (angle >= 90 && angle <= 180){
16      setLeftEngine(100 - angle * 100 / 90 * -1);
        }
```

```
18      else if (angle >= 180 && angle <= 270) {
            setLeftEngine(300 - angle * 100 / 90);
20          }
        else if (angle >= 270 && angle <= 360) {
22          setRightEngine( 300 - angle * 100 / 90 * -1);
            }
24  }});
    Integer E = seekBar.getProgress()*(1-getLeftEngine()/100);
26  Integer D = seekBar.getProgress()*(1-getRightEngine()/100);

28  strE.setData(E.shortValue());
    strD.setData(D.shortValue());
30
    publisherE.publish(strE);
32  publisherD.publish(strD);

34  Thread.sleep(100);
    }
```

4.3.4 Consola de Eventos

De forma a que o utilizador tenha acesso ao conjunto de eventos realizados durante a utilização da aplicação, foi adicionada uma consola onde estes são registados utilizando a classe `LogFragment` (Ver figura 4.11).

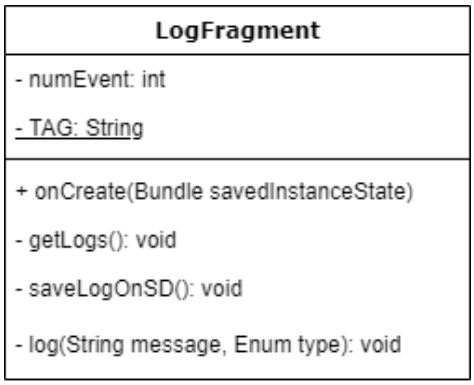

Figura 4.11: `UML` da classe `LogFragment`

A cada ocorrência de um evento, uma mensagem associada ao mesmo é guardada em memória juntamente com a hora do acontecimento, sendo que, ao utilizador aceder, todas estas mensagens vão ser impressas. Posto isto, o processo que permite imprimir os eventos na consola está representado no código 14.

Código 14: Imprimir eventos na consola

```
private void log(String message, Enum type){
    Log.d(TAG, message);

    LinearLayout EventsLayout = findViewById(R.id.EventsLayout);
    TextView valueTV = new TextView(this);

    valueTV.setId(numEvent++);

    LinearLayout.LayoutParams layout = new LinearLayout.
    LayoutParams(

    LinearLayout.LayoutParams.MATCH_PARENT, LinearLayout
    .LayoutParams.WRAP_CONTENT);

    layout.setMargins(5, 5, 5,5);
    valueTV.setLayoutParams(layout);
    valueTV.setTextSize(14);

    valueTV.setBackgroundColor(getResources().getColor(R.color.
    yellowGreen));
        }
```

Deste modo, ao ocorrer um evento, este será apresentado numa consola tal como exemplificado na figura 4.12.

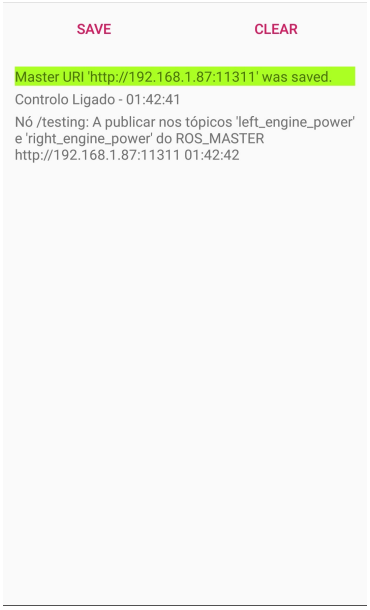

Figura 4.12: Consola de eventos da aplicação

Finalmente, tal como se pode ver na figura 4.12, foram adicionados 2 botões: O *SAVE*, que permite ao utilizador guardar o conteúdo da consola num ficheiro "*.txt"e o *CLEAR*, que permite apagar o conteúdo da consola. Sendo que para a implementação do segundo basta apagar as mensagens que são guardadas em memória, para a implementação do primeiro optou-se por um método que cria um ficheiro de texto e que o guarda na memória externa do utilizador, sendo que outros métodos como *download* do ficheiro foram impossíveis de implementar. Assim, foi possível implementar a exportação dos registos da aplicação utilizando o código representado em 15.

Código 15: Guardar os eventos da consola

```
public void saveLogOnSD() {
  try {
       File root = new File(Environment.
  getExternalStorageDirectory(), "Notes");
       if (!root.exists()) {
           root.mkdirs();
       }
       SimpleDateFormat formatter = new SimpleDateFormat("dd/MM
  /yyyy HH:mm:ss");
       Date date = new Date();
       String sFileName = "AndroidBoatCom_Log_" + formatter.
  format(date).replace(" ","_")+".txt";
       File gpxfile = new File(root, sFileName);
       FileWriter writer = new FileWriter(gpxfile);
       for(int i=0;i<Global.messages.size();i++) {
           writer.append(Global.messages.get(i)+"\n");
       }
       writer.flush();
       writer.close();
       Toast.makeText(getApplicationContext(), "O seu Log foi
  guardado.", Toast.LENGTH_SHORT).show();
       } catch (IOException e) {
           e.printStackTrace();
       }
   }
 }
```

4.3.5 Sobre

Finalmente, foi implementada uma página "Sobre"fazendo recurso a componentes`TextView` e `ImageView`. Esta área da aplicação foi criada com o objetivo de informar sobre a origem do projeto, os seus autores, a sua aplicação e

o principal meio de comunicação utilizada no sistema Aplicação-Embarcação. Esta área encontra-se representada na figura 4.13.

Figura 4.13: Página Sobre

Capítulo 5

Validação e Testes

Visto que a embarcação *USV-enautica1* ainda se encontra em fase de construção, não foi possível realizar testes em contexto real, ou seja, no mar. No entanto, todo o projeto foi desenvolvido com o apoio de um simulador virtual para testar o seu bom funcionamento. Foi também possível testar com alguns sensores na ENIDH que permitiram retirar valores reais e visualizá-los na aplicação, contudo, as visitas à ENIDH foram limitadas, visto que nos encontramos a meio de uma pandemia com medidas impostas para distanciamento social.

Pondo isto, as simulações foram criadas em ambiente controlado, onde a conexão à rede *Wi-fi* é estável e os valores apresentados são pré-definidos, apesar de haver simulação de ruído.

Ainda assim, para testar o bom funcionamento da aplicação e receção de dados, foi utilizado o simulador desenvolvido, definindo o nome dos tópicos a serem publicados idênticos aos nomes dos tópicos subscritos na aplicação. Para testar o controlo, apenas o podemos fazer em modo de texto, onde são representados os níveis de potência de cada motor através de nós que subscrevem a tópicos publicados pela aplicação.

Após a elaboração dos testes, verificou-se que havia oscilações nos tópicos subscritos, o que tornavam a leitura dos dados imprecisa. Posto isto foi desenvolvido o método de filtragem mencionado na secção 4.3.2 o que reduziu estas oscilações, no entanto, não as eliminou completamente.

Capítulo 6

Conclusões e Trabalho Futuro

O desenvolvimento deste projeto foi um desafio e um percurso bastante interessante por várias razões. Utilizar o `Robot Operating System` foi bastante motivador, pois trata-se de uma *framework* com que nunca tivemos contacto, daí ser necessário a realização de um estudo intensivo sobre a sua utilização, aplicações e sobre como utilizá-lo em `Android`. Durante o processo de aprendizagem com este sistema, para além das pesquisas e estudo autónomo, foi fulcral a discussão sobre dúvidas que foram ocorrendo e apresentação de soluções aos nossos orientadores Carlos Gonçalves e Mário Assunção, assim como do nosso colega que desenvolveu o projeto anterior a este, Frederico Cardoso, devido à sua experiência com este sistema operativo.

As maiores dificuldades encontradas passaram pela incompatibilidades existentes ao adicionar a biblioteca `Android Core` ao projeto de `Android Studio` já existente, o que resultou em múltiplos projetos corrompidos ao tentar juntar a componente `ROS` à componente `Android`. Após a resolução desse problema, ocorreram problemas durante a subscrição dos valores publicados pelo `ROS_MASTER` tanto da máquina virtual como do *Raspberry Pi 3 Model B+* pois, mesmo estando ligados ao seu `ROS_MASTER_IP`, o valor dos dados subscritos eram nulos independentemente do que era publicado. Estes problemas davam-se devido à má configurações da rede que não permitia o reconhecimento dos nós entre si. Com os problemas assim descritos resolvidos, foi possível atingir os objetivos do projeto com sucesso e, mesmo que o testes tenham sido realizados maioritariamente recorrendo à maquina virtual, sendo que o `ROS` é uma Sistema Operativo modular, o que funciona com o simulador funcionará certamente com a embarcação.

Futuramente, pretende-se oferecer uma maior autonomia ao utilizador dando a opção de adição de mostradores da biblioteca `Gauges` com os tópicos de `ROS` escolhidos por ele, o que tornará a aplicação escalável, melhorias na consola de eventos da aplicação, apresentando todos os problemas específicos que poderão ocorrer na tentativa de ligação com a embarcação ou durante a conexão, a implementação de um sistema de *AIS*, sendo assim possível a localização e monitorização de curto alcance de outras embarcações e, finalmente, a apresentação de uma página de estatísticas, onde o utilizador poderá analisar a variação dos valores correspondentes aos módulos da embarcação ao longo do tempo em que esteve a utilizar a aplicação.

Bibliografia

[1] Carlos Viegas e Nuno Brás, Android BoatCOM, 2020
https://github.com/carlosviegas-nunobras/PRJ-AndroidBoatCOM

[2] Frederico Cardoso, BoatCOM, 2019
https://github.com/fred-cardoso/BoatCOM

[3] GitHub — The world's leading software development platform
https://github.com/

[4] Android https://www.android.com/intl/pt_pt/

[5] ROS — Powering the world's robots
https://www.ros.org/

[6] rosbridge suite ROS Wiki,
http://wiki.ros.org/rosbridge_suite

[7] android_core 0.1.0 documentation,
http://rosjava.github.io/android_core/latest/

[8] Java — Oracle
https://www.java.com/

[9] RosJava
http://wiki.ros.org/rosjava

[10] Gauge — A Gauge View for Android,
https://github.com/Pygmalion69/Gauge

[11] MapView — Google APIs for Android,
https://developers.google.com/android/
reference/com/google/android/gms/maps/MapView

[12] JoystickView — A virtual joystick for android,
https://github.com/controlwear/virtual-joystick-android

[13] SeekBar — Android Developers,
https://developer.android.com/reference/android/widget/SeekBar

[14] Firebase — Google,
https://firebase.google.com/

[15] GoogleSignInActivity.java - Firebase
https://github.com/firebase/quickstart-
android/blob/master/auth/app/src/main/java/com/google/firebase/quickstart

[16] Android Studio — Android Developers
https://developer.android.com/studio

[17] Moqups — Online Mockup, Wireframe & UI Prototyping Tool
https://moqups.com/

[18] PuTTY — A free SSH and telnet client for Windows
https://www.putty.org/

[19] Ubuntu — More than Linux. Security and compliance for the full stack.
https://ubuntu.com/

[20] VMWare Workstation — VMware Workstation Player allows you to run
a second, isolated operating system on a single PC
https://www.vmware.com/products/workstation-player.html

[21] rospy — ROS
texttthttp://wiki.ros.org/rospy

[22] JSON
texttthttps://www.json.org/json-en.html

Apêndice A

Diagrama UML da aplicação

Visto que o diagrama *UML* gerado apresenta uma dimensão elevada, o mesmo apresenta-se isolado na página seguinte identficado como figura A.1

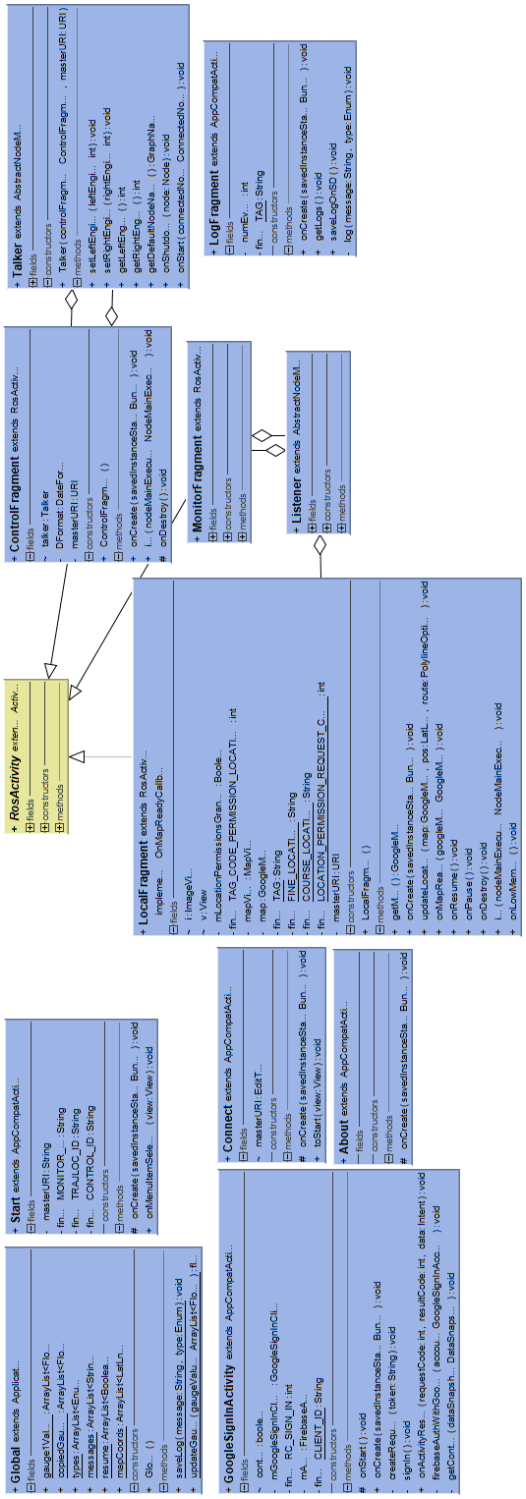

Figura A.1: Diagrama UML da aplicação