

UNIVERSIDADE FEDERAL DOS VALES DO JEQUITINHONHA E MUCURI
Bacharelado em Sistemas de Informação
Caio Teixeira de Farias

**DESENVOLVIMENTO DE UMA APLICAÇÃO ANDROID PARA TELEOPERAÇÃO
E INTERAÇÃO COM A TÉCNICA SLAM EM UM QUADROTOR UTILIZANDO O
ROS**

Diamantina
2018

Caio Teixeira de Farias

**DESENVOLVIMENTO DE UMA APLICAÇÃO ANDROID PARA TELEOPERAÇÃO
E INTERAÇÃO COM A TÉCNICA SLAM EM UM QUADROTOR UTILIZANDO O
ROS**

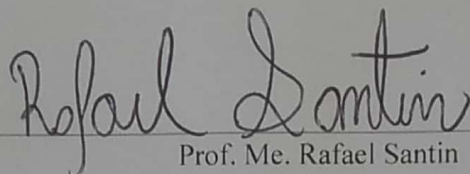
Trabalho de Conclusão de Curso apresentado ao curso de graduação em Sistemas de Informação, como parte dos requisitos exigidos para a obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Me. Rafael Santin

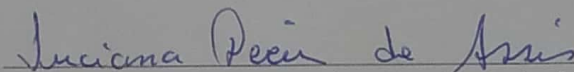
**Diamantina
2018**

Monografia de projeto final de graduação sob o título “Desenvolvimento de uma aplicação Android para teleoperação e interação com a técnica SLAM em um Quadrotor utilizando o ROS”, defendida por Caio Teixeira de Farias e aprovada em 2 de agosto de 2018, em Diamantina, Minas Gerais.

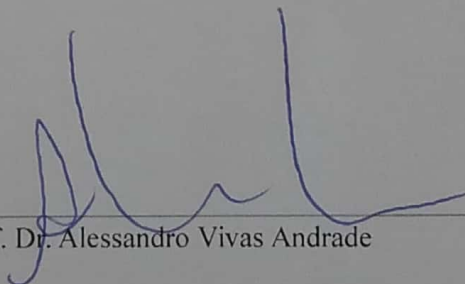
Banca Examinadora:



Prof. Me. Rafael Santin
Orientador



Prof.ª Dra. Luciana Pereira de Assis



Prof. Dr. Alessandro Vivas Andrade

Dedico aos meus pais, irmãos, amigos e mestres, que me deram base e suporte para concluir esse trabalho.

AGRADECIMENTOS

Agradeço, primeiramente, a deus pela minha vida, a minha família que me proporcionou estrutura suficiente durante esses anos de graduação, para que chegasse ao objetivo final. Ao meu orientador Rafael Santin pela dedicação, confiança e generosidade. A todos os professores, técnicos e funcionários da Universidade Federal dos Vales do Jequitinhonha e Mucuri, que se dedicaram sempre em proporcionar o melhor ensino para nós estudantes. A cidade de Diamantina e aos amigos que a que fiz, em especial aos da República Rabo de Vaca, que tornaram essa caminhada mais agradável e me ajudaram a concluir essa etapa importante na minha vida.

Se valorizamos a nossa liberdade, podemos mantê-la e defendê-la. (Richard Stallman)

RESUMO

Na teleoperação de robôs móveis, um robô pode ser controlado remotamente para executar tarefas específicas. A teleoperação é útil quando o ambiente operacional é perigoso, impraticável ou economicamente inviável. Para operar à distância um robô móvel de maneira eficiente, é importante que o sistema de teleoperação permita que o operador esteja ciente do ambiente de navegação e possa dar instruções precisas ao robô. Técnicas de mapeamento, como o SLAM, auxiliam veículos autônomos e sistemas de teleoperação a obter informações sobre o ambiente de navegação do robô. Sistemas operacionais para robôs, fornecem abstração de hardware e outras ferramentas que auxiliam no desenvolvimento de softwares para robótica. O presente trabalho consiste no desenvolvimento de uma aplicação para dispositivos móveis, com Sistema Operacional Android, para teleoperação e interação com a técnica SLAM em um Quadrotor (simulado) utilizando o ROS (*Robot Operating System*). A principal motivação deste trabalho, foi a expansão do uso de robôs em conjunto com *smartphones*, aproveitando a popularidade desses dispositivos e seus recursos computacionais. Para o desenvolvimento da aplicação, foi necessário o entendimento, principalmente, do *framework* ROS, da plataforma Android e da linguagem de programação Java. Ao final deste trabalho, foi possível obter uma ferramenta que permite controlar remotamente um *drone* de maneira assistida, em ambientes conhecidos ou desconhecidos, e obter um mapa do espaço de navegação.

Palavras-chave: Robótica Móvel. Dispositivos Móveis. Android. ROS (*Robot Operating System*). SLAM.

ABSTRACT

In mobile robot teleoperation, a robot can be controlled remotely to perform specific tasks. Teleoperation is useful when the operating environment is dangerous, impractical or economically unfeasible. To remotely operate a mobile robot efficiently, it is important that the teleoperation system allows the operator to be aware of the navigation environment and can give precise instructions to the robot. Mapping techniques, such as SLAM, assist autonomous vehicles and teleoperation systems to obtain information about the robot's navigational environment. Robot operating systems provide hardware abstraction and other tools that aid in the development of software for robotics. The present work consists in the development of an application for mobile devices, with Android Operating System, for teleoperation and interaction with the SLAM technique in a Quadrotor (simulated) using ROS (Robot Operating System). The main motivation of this work, was the expansion of the use of robots in conjunction with smartphones, taking advantage of the popularity of these devices and their computational resources. For the development of the application, it was necessary to understand, mainly, the framework, the Android platform and the Java programming language. At the end of this work, it was possible to obtain a tool that allows remote control of a drone in an assisted way in known or unknown environments, and to obtain a navigation space map.

Keywords: Mobile Robotics. Mobile devices. Android. ROS (Robot Operating System). SLAM.

LISTA DE ILUSTRAÇÕES

Figura 1 – A pilha de software do Android	30
Figura 2 – Ciclo de vida de uma Atividade Android	32
Figura 3 – Funcionamento conceitual do Acelerômetro	34
Figura 4 – Orientação dos eixos que definem as componentes de aceleração em um <i>smartphone</i>	35
Figura 5 – Distribuições ROS ao longo do tempo	36
Figura 6 – Distribuições recentes do ROS	37
Figura 7 – Sistema de arquivos ROS	37
Figura 8 – Conceitos básicos do Grafo de Computação do ROS	39
Figura 9 – Comunicação entre um nó <i>publisher</i> (a) e um <i>subscriber</i> (b) pelo <i>ROS Master</i>	41
Figura 10 – Comando <code>rosmmsg</code> para mensagem <code>geometry_msgs/Twist</code>	42
Figura 11 – Comunicação de mensagens ROS por tópicos	44
Figura 12 – Comunicação por serviços ROS	46
Figura 13 – Diagrama dos conceitos do Grafo de Computação ROS	46
Figura 14 – Ferramenta <code>rqt_graph</code>	47
Figura 15 – <i>Namespaces</i> no <code>rqt_graph</code>	50
Figura 16 – Carro robô utilizando SLAM	51
Figura 17 – Quadrotor	53
Figura 18 – Simulação de um braço robô no Gazebo	53
Figura 19 – Controle Joystick Android	60
Figura 20 – Posições possíveis retornadas do <i>joystick</i>	63
Figura 21 – Interface tela inicial - <i>Master Chooser</i>	65
Figura 22 – Interface do operador	66
Figura 23 – Exemplo de mapa gerado pelo SLAM utilizando a aplicação desenvolvida	68
Figura 24 – Ambiente <i>indoor</i> executado no Gazebo	69
Figura 25 – Grafo do ROS durante a execução da aplicação	70
Figura 26 – Diagrama do sistema final obtido	71
Figura 27 – <i>SyncNow</i> Android Studio	82

LISTA DE CÓDIGOS

Código A.1 – Repositório Rosjava	81
Código A.2 – Pacotes Rosjava utilizados	82
Código A.3 – Arquivos excluídos no Android	82
Código A.4 – Substituir ícone Android	83
Código A.5 – Permissões do app	83

LISTA DE TABELAS

Tabela 1 – Métodos do ciclo de vida de uma Atividade Android	33
Tabela 2 – Ferramenta <code>roscnode</code> do ROS	40
Tabela 3 – Ferramenta <code>rosmmsg</code> do ROS	42
Tabela 4 – Principais mensagens do ROS	43
Tabela 5 – Ferramenta <code>rostopic</code> do ROS	44
Tabela 6 – Pacotes do <code>hector_quadrotor</code>	59
Tabela 7 – Pacotes do <code>hector_slam</code>	59
Tabela 8 – Variáveis da classe <code>JoyStickClass</code> definidas para as posições da alavanca do <i>joystick</i>	62
Tabela 9 – Componentes da interface do usuário	67

LISTA DE ABREVIATURAS E SIGLAS

API – *Application Programming Interface*

DNS – *Domain Name System*

EOL – *End Of Life*

GB – *Gigabytes*

GHz – *Gigahertz*

IDE – *Integrated Development Environment*

IP – *Internet Protocol*

LTS – *Long Term Support*

ROS – *Robot Operating System*

SO – *Sistema Operacional*

TCP – *Transmission Control Protocol*

UDP – *User Datagram Protocol*

UFVJM – *Universidade Federal dos Vales do Jequitinhonha e Mucuri*

URI – *Uniform Resource Identifier*

VANT – *Veículo Aéreo Não Tripulado*

VCS – *Version Control Systems*

XML – *Extensible Markup Language*

SUMÁRIO

1	INTRODUÇÃO	27
1.1	Objetivos	28
1.1.1	Objetivo geral	28
1.1.1.1	Objetivos específicos	28
1.2	Organização do trabalho	28
2	FUNDAMENTAÇÃO TEÓRICA	29
2.1	Android	29
2.1.1	Arquitetura	29
2.1.2	Elementos básicos de um projeto Android	31
2.1.2.1	AndroidManifest.xml	31
2.1.2.2	Gradle	31
2.1.2.3	Atividade (<i>Activity</i>)	31
2.1.2.4	Arquivos XML de Layout	33
2.1.3	Sensor Acelerômetro	33
2.2	ROS	35
2.2.1	Sistemas de Arquivos	37
2.2.2	Grafo de Computação	39
2.2.2.1	Nós	39
2.2.2.2	Mestre	40
2.2.2.3	Servidor de parâmetros	41
2.2.2.4	Mensagens	41
2.2.2.5	Tópicos	43
2.2.2.6	Serviços	45
2.2.2.7	Bolsas	45
2.2.2.8	Visualizando o grafo ROS	46
2.2.3	Comunidade	47
2.2.4	Rosjava	48
2.2.5	Outros conceitos ROS	48
2.2.5.1	Catkin	48
2.2.5.2	Variáveis de ambiente ROS	49
2.2.5.3	roscore	49
2.2.5.4	roslaunch	49
2.2.5.5	Names	50
2.2.5.6	Namespaces	50

2.3	SLAM	50
2.4	Robótica móvel	51
2.4.1	Drones	52
2.4.1.1	Quadrotores	52
2.5	Simulação robótica	52
3	METODOLOGIA	55
3.1	Trabalhos relacionados e proposta	55
3.2	Ferramentas utilizadas	56
3.2.1	Hardware	56
3.2.2	ROS <i>Kinetic</i>	56
3.2.3	Android Studio	57
3.2.4	Gazebo	57
3.2.5	Linguagens de programação	57
3.2.5.1	Java	57
3.2.5.2	XML	58
3.2.6	android_core	58
3.2.7	Hector ROS	58
3.2.8	Android Joystick	59
3.3	Implementação	60
3.3.1	Classe <i>MainActivity.java</i>	60
3.3.2	Classe <i>SystemCommands.java</i>	62
3.3.3	Classe <i>JoyStickClass.java</i>	62
3.3.4	Classe <i>NodeControle.java</i>	63
4	RESULTADOS	65
4.1	Executando o ROS/Gazebo no computador	68
5	CONCLUSÃO	73
5.1	Trabalhos futuros	74
	REFERÊNCIAS	75
	A CONFIGURAÇÃO DO AMBIENTE DE DESENVOLVIMENTO	
	ROSJAVA/ANDROID STUDIO	81
A.1	Gradle	81
A.2	AndroidManifest	83
	B CÓDIGO FONTE	85
B.1	Classe <i>MainActivity.java</i>	85
B.2	Classe <i>NodeControle.java</i>	89

B.3	Classe <i>SystemCommands.java</i>	98
------------	--	-----------

1 INTRODUÇÃO

A robótica sofreu uma mudança transformacional na última década. O advento de novos *frameworks* de código aberto, que fornecem um conjunto de ferramentas, infraestrutura e melhores práticas para construir novos robôs e aplicações, tornou a robótica mais acessível a novos usuários, tanto em pesquisas quanto em aplicativos de consumo (KOUBAA, 2016). Dessa forma, são desenvolvidos robôs e ferramentas para a robótica cada vez mais inteligentes, tornando o uso desses dispositivos, já difundido no setor industrial, gradativamente presente em diversas áreas de negócio.

Robôs estão sendo usados para uma série de tarefas, incluindo as impraticáveis e perigosas para o ser humano. No setor de mineração por exemplo, veículos aéreos não tripulados (VANTs) com sensores infravermelhos e zoom telescópico podem sobrevoar minas e ajudar a mostrar quando uma área de detonação foi totalmente evacuada de pessoas e equipamentos (ALMEIDA, 2016). Ainda no âmbito da mineração, sistemas robóticos teleoperados, onde robôs são controlados remotamente por um operador humano, são usados para não expor o operador a um ambiente hostil de operação e para aumentar a produtividade, pois ao inserir o operador virtualmente em um ambiente de trabalho, evita-se os deslocamentos aos locais de operação (COTA *et al.*, 2017). Na produção cinematográfica, os VANTs também estão revolucionando o setor. Capturar imagens, antes impossíveis fisicamente ou economicamente inviáveis, são cada vez mais fáceis devido as características desses veículos aéreos, que são pequenos, leves e possuem um custo bem inferior em relação a outros meios de captação de imagens aéreas (BARBOSA, 2016).

O mapeamento de áreas também é uma atividade relevante que vem sendo usada em conjunto com sistemas robóticos. Robôs autônomos, utilizam-se de técnicas de mapeamento através de sensores acoplados para se localizarem e navegarem em diversos tipos de ambientes. Técnicas de mapeamento também vêm sendo utilizadas juntamente com robôs para obter informações sobre ambientes desconhecidos, e ajudar, por exemplo, equipes em atividades de busca e salvamento (SILVA; YEPES, 2016).

O já mencionado advento de novos *frameworks*, que apoiam o desenvolvimento de sistemas robóticos, também facilitou a integração de robôs com outros dispositivos. A popularização dos *smartphones* possibilitou uma grande mudança no desenvolvimento de robôs e aplicativos para a área da robótica. Os *smartphones* são dispositivos inteligentes de alto poder computacional e possuem diversos sensores que podem ser aproveitados pelos robôs. Robôs móveis podem, por exemplo, utilizar das tecnologias *wireless* dos *smartphones* para serem controlados remotamente e transmitirem diversos tipos de informação relevantes para apoiar o operador durante o controle do robô.

1.1 Objetivos

1.1.1 Objetivo geral

O principal objetivo desse trabalho é desenvolver uma aplicação para dispositivos com sistema operacional Android, integrada com o sistema ROS (*Robot Operating System*), que permita controlar um robô (quadrotor) remotamente, que possibilite visualizar o ambiente remoto através do sensor da câmera do robô e gerar um mapa do espaço de navegação, através de uma técnica de mapeamento (SLAM).

1.1.1.1 Objetivos específicos

Os objetivos específicos do trabalho são:

- Entender os conceitos básicos da plataforma Android e o funcionamento dos aplicativos para esse sistema.
- Compreender o *framework* ROS e o seu uso no desenvolvimento de aplicações para robôs.
- Abordar a técnica de mapeamento SLAM e sua utilização através de robôs.
- Desenvolver uma aplicação Android apoiada pela biblioteca do ROS, Rosjava.
- Criar uma base de estudos para integração do sistema ROS com dispositivos Android.

1.2 Organização do trabalho

No [Capítulo 1](#) é feita uma apresentação do tema a ser abordado no trabalho, mostrando algumas aplicações na área da robótica e os objetivos que pretendem ser cumpridos com esse projeto. A fundamentação teórica é feita no [Capítulo 2](#), abordando os conceitos que são utilizados na aplicação proposta e situando esse trabalho na área da robótica juntamente com o desenvolvimento de aplicações Android. O [Capítulo 3](#) trata aborda as contribuições para o projeto, que motivaram a proposta da aplicação desenvolvida nesse presente trabalho, e as ferramentas utilizadas durante o processo de desenvolvimento. No [Capítulo 4](#) é mostrado a interface final da aplicação desenvolvida, a descrição de cada componente presente nela e suas aplicabilidades, e as classes utilizadas para manipular e executar as funcionalidades da aplicação. No [Capítulo 5](#) é feita as conclusões sobre o trabalho desenvolvido e as propostas para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Android

O Android surgiu em 2003, na cidade de Palo Alto, localizada no estado da Califórnia nos EUA. Na ocasião, foi criada a Android Inc. Um de seus fundadores, Andy Rubin, descreveu a Android Inc. como: “Dispositivos móveis mais inteligentes, mais conscientes da localização e das preferências de seus donos.” De forma intrigante, no ano de 2005 a Google compra a Android Inc. e pouco se sabia das intenções da gigante da tecnologia com esse negócio até 2007. Nesse ano, a *Open Handset Alliance*¹ anuncia as intenções da primeira plataforma realmente aberta do mundo para dispositivos móveis: “promover a inovação em dispositivos móveis e oferecer aos consumidores uma experiência de usuário muito melhor do que grande parte do que está disponível nas plataformas móveis atuais”. (KRAJCI; CUMMINGS, 2013)

Com a introdução do Android, um único sistema operacional removeu a necessidade de reimplementação de aplicativos de telefone e middleware. As empresas que criam novos dispositivos agora podem se concentrar muito mais no hardware e nos componentes subjacentes. Através dessa mudança revolucionária, não só as empresas foram beneficiadas, os desenvolvedores de software agora poderiam liberar aplicativos para vários dispositivos com muito poucas alterações na base de código. Devido a essas características e o fato do sistema possuir uma licença inspirada nas ideias de software livre, os desenvolvedores podiam passar mais tempo trabalhando nos aplicativos que esses telefones executavam, criando as ricas e poderosas aplicações que temos hoje (KRAJCI; CUMMINGS, 2013).

2.1.1 Arquitetura

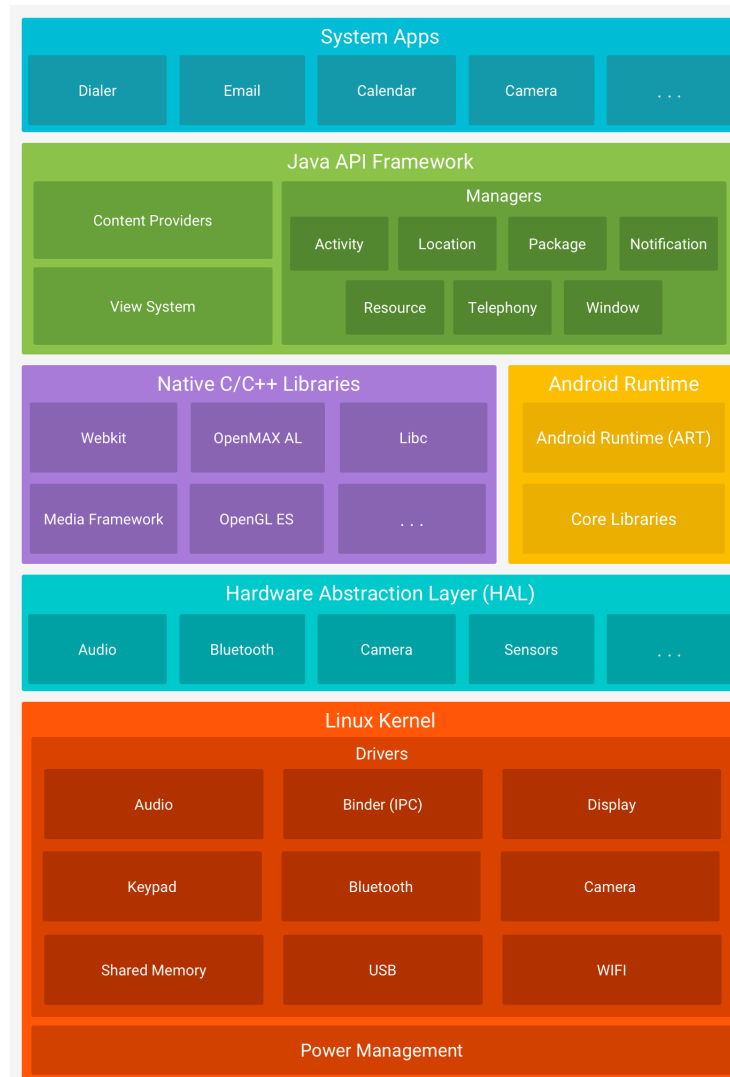
O Android é uma pilha de software² baseado no Linux. A [Figura 1](#) mostra os principais componentes dessa estrutura que é dividida em cinco camadas e inclui seis grupos diferentes (DEVELOPER, 2018a).

- **Kernel do Linux:** Os serviços principais, incluindo os drivers de hardware, o gerenciamento de processos e de memória, a rede de segurança e o gerenciamento de energia são gerenciados por um kernel do Linux (DEVELOPER, 2018a).
- **Camada de abstração de hardware (HAL):** Essa camada fornece interfaces para os recursos de hardware, como câmera e sensores do dispositivo, para a estrutura da Java API. Quando uma *Framework* API necessita de um desses recursos, ela faz uma chamada para o sistema Android que carrega o módulo da biblioteca para o componente de hardware específico (DEVELOPER, 2018a).

¹ Consórcio de grandes empresas de tecnologia como Google, Samsung, Sony, Qualcomm e HTC

² Grupo de programas que trabalham em conjunto para produzir um resultado ou atingir um objetivo comum.

Figura 1 – A pilha de software do Android



Fonte – Developer (2018a)

- **Android Runtime (ART):** O ART do Android é o mecanismo que alimenta os aplicativos e, junto com as bibliotecas, formam a base para a estrutura do aplicativo. Ele foi introduzido no Android a partir da versão 5.0 e é o sucessor da *Dalvik VM*, a máquina virtual utilizada nas versões mais antigas do Android. O ART trouxe uma série de novos recursos que melhoram o desempenho e a estabilidade da plataforma e dos aplicativos Android. O ART e o *Dalvik* são compatíveis executando o Dex, o bytecode utilizado no Android. Portanto, os aplicativos desenvolvidos para o *Dalvik* devem funcionar ao executar o ART. No entanto, algumas técnicas que funcionam nos desenvolvidos para o *Dalvik* não funcionam com o ART (ANDROID, 2018c).
- **Bibliotecas C/C++ nativas:** Os elementos básicos do sistema Android, como o HAL, exigem bibliotecas nativas programadas em C e C++. Através da Java *Framework APIs* é possível expor a funcionalidade de algumas dessas bibliotecas nativas aos aplicativos

([DEVELOPER, 2018a](#)).

- **Java API Framework:** Fornece as classes usadas para criar aplicativos Android. A *Java API Framework* também fornece uma abstração genérica para acesso ao hardware e gerencia a interface do usuário, bem como os recursos do aplicativo.
- **Aplicativos do sistema:** A camada dos aplicativos é executada no tempo de execução do Android. Todos os aplicativos, tanto nativos do sistema quanto de terceiros, são executados sem status especial, de modo que um aplicativo de terceiro pode ser utilizado no lugar de um aplicativo nativo. Os aplicativos de terceiros também podem ser executados utilizando aplicativos nativos do sistema Android. Por exemplo, aplicativos de serviços de SMS e Email podem ser invocados por outros aplicativos desenvolvidos ([DEVELOPER, 2018a](#)).

2.1.2 Elementos básicos de um projeto Android

2.1.2.1 AndroidManifest.xml

Todas as aplicações Android contém o arquivo `AndroidManifest.xml`. Este documento XML contém diversas informações que são carregadas previamente a execução do código do aplicativo, sendo necessárias para seu funcionamento. No arquivo `AndroidManifest.xml` são predefinidos elementos que podem especificar, entre outras coisas, as permissões que o aplicativo deve ter para acessar partes protegidas do sistema, as bibliotecas necessárias, os ícones e nome da aplicação ([DEVELOPER, 2018e](#)).

2.1.2.2 Gradle

Segundo [Developer \(2018c\)](#), o *Gradle* é “um kit de ferramentas de compilação avançado para automatizar e gerenciar o processo de compilação, permitindo que você defina configurações de compilação personalizadas e flexíveis”. A função do *Gradle* é pegar todos os arquivos de origem necessários (`.xml` ou `.java` por exemplo) para criar um arquivo APK compactado. De maneira mais simples, ele é capaz de copiar arquivos de um diretório para outro antes da construção real do aplicativo, permitindo que o programador tenha uma topologia estruturada do projeto (por exemplo, uma pasta específica para arquivos de linguagem de programação específicas), sem a necessidade de escrever manualmente um *script* dedicado para vincular esses arquivos ([HANËIN, 2017](#)). O *script* de construção do projeto é feito através de um arquivo chamado `build.gradle`.

2.1.2.3 Atividade (*Activity*)

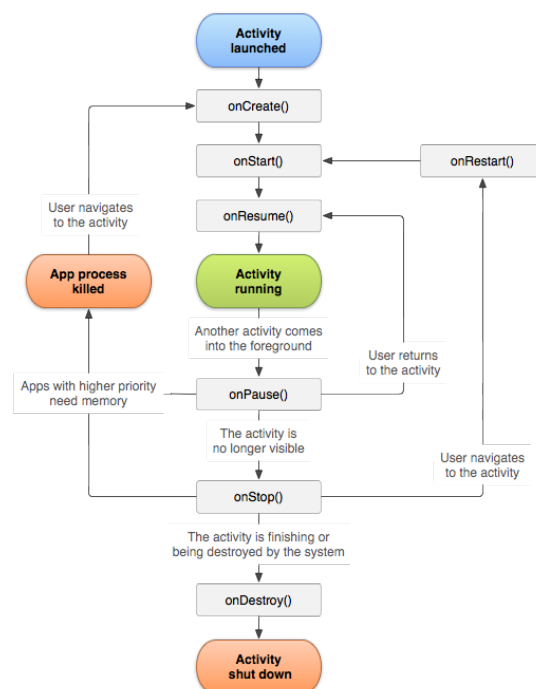
Cada tela de uma aplicação onde o usuário interage e executa alguma tarefa específica é chamada de *Activity*. Um aplicativo pode conter várias atividades (*Activities*), onde estão localizadas todos os componentes visuais e interativos do aplicativo, como botões, textos e

imagens. Os recursos fornecidos pelo Android para as *Activities* são acessados através da classe *Activity*. Através dela, podemos exibir a interface do usuário, alocar memória para os objetos dessa interface e criar um novo processo (POUDEL, 2013). Estendemos a classe *Activity*, ou alguma de suas subclasses do pacote `android.app`, para criarmos as atividades. Essa classe fornece todas as especificações características de uma atividade.

Normalmente, uma atividade em um aplicativo é especificada como a atividade principal, sendo essa a primeira tela a ser exibida quando o usuário inicia o aplicativo. Cada atividade pode então iniciar outra atividade para realizar ações diferentes. Por exemplo, a atividade principal em um aplicativo simples de e-mail, pode fornecer a tela que mostra uma caixa de entrada de e-mail. A partir daí, a atividade principal pode iniciar outras atividades que fornecem telas para tarefas como escrever e-mails e abrir e-mails individuais (DEVELOPER, 2018d).

Quando uma nova *Activity* é iniciada, a atividade anterior é interrompida e colocada em uma “pilha de retorno” que segue o funcionamento básico de uma pilha (LIFO³). Esse mecanismo permite que uma atividade possa ser preservada quando novas são executadas. Por exemplo, quando o usuário inicia uma nova atividade ele pode, através do botão voltar, retomar a *Activity* anterior que foi colocada na pilha, fazendo com que a atividade atual seja então destruída (DEVELOPER, 2018b). Para poder desenvolver uma aplicação Android é preciso compreender os conceitos básicos desse ciclo de vida das atividades. A Figura 2 mostra um diagrama dos métodos que controlam esse ciclo e a descrição de cada um deles é mostrada na Tabela 1.

Figura 2 – Ciclo de vida de uma Atividade Android



Fonte – Developer (2018b)

³ *Last In, First Out* - O último que entra é o primeiro que sai

Tabela 1 – Métodos do ciclo de vida de uma Atividade Android

Método	Descrição
<code>onCreate()</code>	Chamado quando a atividade é criada pela primeira vez. É onde se deve fazer toda a configuração estática normal — criar exibições, vincular dados a listas, etc. Sempre seguido de <code>onStart()</code> ;
<code>onRestart()</code>	Chamado depois que atividade tiver sido interrompida, logo antes de ser reiniciada. Sempre seguido de <code>onStart()</code> ;
<code>onStart()</code>	Chamado logo antes de a atividade se tornar visível ao usuário. Seguido de <code>onResume()</code> se a atividade for para segundo plano ou <code>onStop()</code> se ficar oculta;
<code>onResume()</code>	Chamado logo antes de a atividade iniciar a interação com o usuário. Nesse ponto, a atividade estará no topo da pilha de atividades com a entrada do usuário direcionada a ela. Sempre seguido de <code>onPause()</code> ;
<code>onPause()</code>	Chamado quando o sistema está prestes a retomar outra atividade. Esse método precisa salvar rapidamente dados não confirmados e interromper o trabalho intensivo da CPU, para preparar a Atividade para perder o foco e ir para o segundo plano. Seguido de <code>onResume()</code> , se a atividade retornar para a frente, ou de <code>onStop()</code> se ficar invisível ao usuário;
<code>onStop()</code>	Chamado quando a atividade não está mais visível para o usuário. Isso pode acontecer porque a atividade está sendo destruída, uma nova atividade está sendo iniciada ou uma atividade existente está sendo retomada e cobrindo a atividade interrompida. Em todos esses casos, a atividade interrompida não é mais visível. Seguido de <code>onRestart()</code> , se a atividade estiver voltando a interagir com o usuário, ou <code>onDestroy()</code> se estiver saindo;
<code>onDestroy()</code>	Chamado antes de a atividade ser destruída. É a última chamada que a atividade receberá. Pode ser chamado porque a atividade está finalizando ou porque o sistema está destruindo temporariamente essa instância da atividade para poupar espaço.

Fonte: [Developer \(2018b\)](#)

2.1.2.4 Arquivos XML de Layout

Projetos de desenvolvimento de aplicativos Android possuem arquivos de layout XML que definem o design das interfaces do aplicativo. Com o uso de XML, um programador pode definir rapidamente como deve ser o layout da atividade. Cada atividade é vinculada a um determinado arquivo .xml contendo vários componentes de layout. Esses componentes são vinculados posteriormente a partir do código Java da atividade ([HANČIN, 2017](#)).

2.1.3 Sensor Acelerômetro

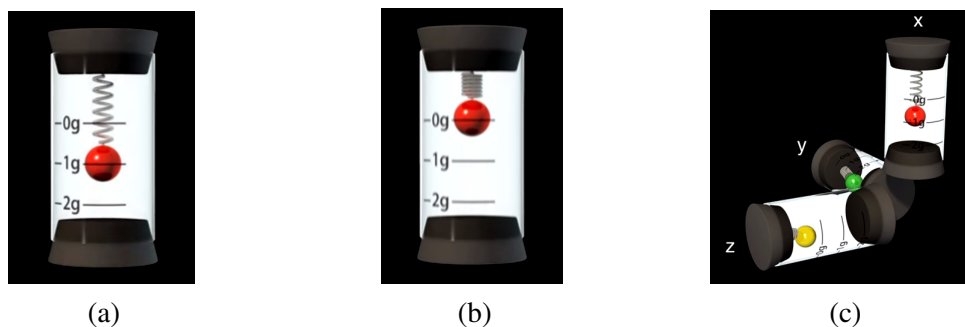
A maioria dos dispositivos com Android possui sensores integrados que medem movimento, orientação e várias condições ambientais. Esses sensores são capazes de fornecer dados brutos com alta precisão e são úteis quando deseja-se monitorar o movimento ou o posicionamento tridimensional do dispositivo, ou ainda monitorar alterações no ambiente ambiente próximo a um dispositivo. Um exemplo da utilização desses sensores, pode ser facilmente vista

em desenvolvimento de jogos, que empregam leituras do sensor de gravidade de um dispositivo para inferir gestos e movimentos complexos do usuário, como inclinação, oscilação ou rotação (DEVELOPER, 2018f).

Um dos sensores de movimento do Android é o acelerômetro. O acelerômetro é um dispositivo capaz de medir a aceleração em um objeto. O acelerômetro pode medir as forças estáticas e as forças dinâmicas. As forças estáticas consistem em forças como a atração da gravidade, enquanto as forças dinâmicas se referem a vibrações e movimentos do acelerômetro. Medição de aceleração estática dá a capacidade de determinar o ângulo exato do dispositivo em relação à terra. Analisando as forças dinâmicas é possível determinar se o dispositivo está se movendo (CARLSON *et al.*, 2015).

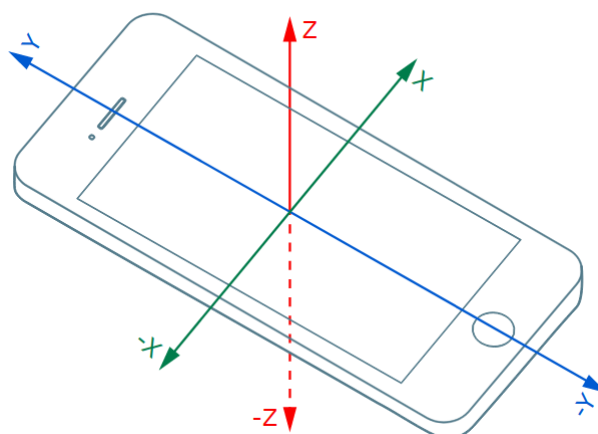
A Figura 3 ilustra o funcionamento conceitual do acelerômetro. Na Figura 3 (a), dentro de um compartimento, uma bola de metal é presa em uma mola de forma calibrada e enquanto presa ainda pode se mover. Se o compartimento se mover para cima ou para baixo a mola, irá se alongar ou retrair (Figura 3 (b)). Se medirmos a deformação da mola, podemos calcular a força aplicada na mola e retornar a aceleração. Colocando cada conjunto desse mecanismo em um eixo diferente (Figura 3 (c)), é possível determinar a orientação de um objeto tridimensional, como em um *smartphone* mostrado na Figura 4. Embora mais complexo que o simples modelo de bola e mola, o acelerômetro de um dispositivo Android possui esses mesmos fundamentos (ENGINEERGUY, 2012).

Figura 3 – Funcionamento conceitual do Acelerômetro



Fonte – EngineerGuy (2012)

Figura 4 – Orientação dos eixos que definem as componentes de aceleração em um *smartphone*



Fonte – W3C (2018)

2.2 ROS

Antes dos sistemas operacionais para robôs, era necessário que cada pesquisador projetasse o software para incorporar ao seu robô, o que exigia habilidades em múltiplos campos do conhecimento, como engenharia mecânica, eletrônica e programação embutida (MAZZARI, 2016). Deste modo, grande parte do tempo dos projetos de robótica ficava a cargo do desenvolvimento do software embutido nos robôs. A ideia principal dos sistemas operacionais para robôs é fornecer aos projetistas e pesquisadores funcionalidades padronizadas, de forma que os desenvolvedores não precisem “reinventar a roda” em seus projetos (NYRO, 2017).

O ROS é um metassistema operacional de código aberto para robôs que fornece serviços que se espera de um sistema operacional convencional, como abstração de hardware, controle de dispositivo de baixo nível, implementação de funcionalidades, passagem de mensagens entre processos e gerenciamento de pacotes. Ele também fornece ferramentas e bibliotecas para obter, criar, gravar e executar código em vários computadores (DOCUMENTATION, 2018).

O ROS foi originalmente desenvolvido em 2007, com o nome de Switchyard, pelo Laboratório de Inteligência Artificial da Universidade de Stanford e posteriormente aperfeiçoado no laboratório de pesquisa em robótica Willow Garage, com contribuições de todo o mundo. Atualmente é executado em plataformas baseadas no Unix, testado principalmente nos sistemas Ubuntu e Mac OS X. A comunidade ROS tem contribuído com suporte para outras plataformas Linux e uma versão para o Microsoft Windows também está atualmente em desenvolvimento, mas ainda em fase experimental (DIFFOUO, 2013).

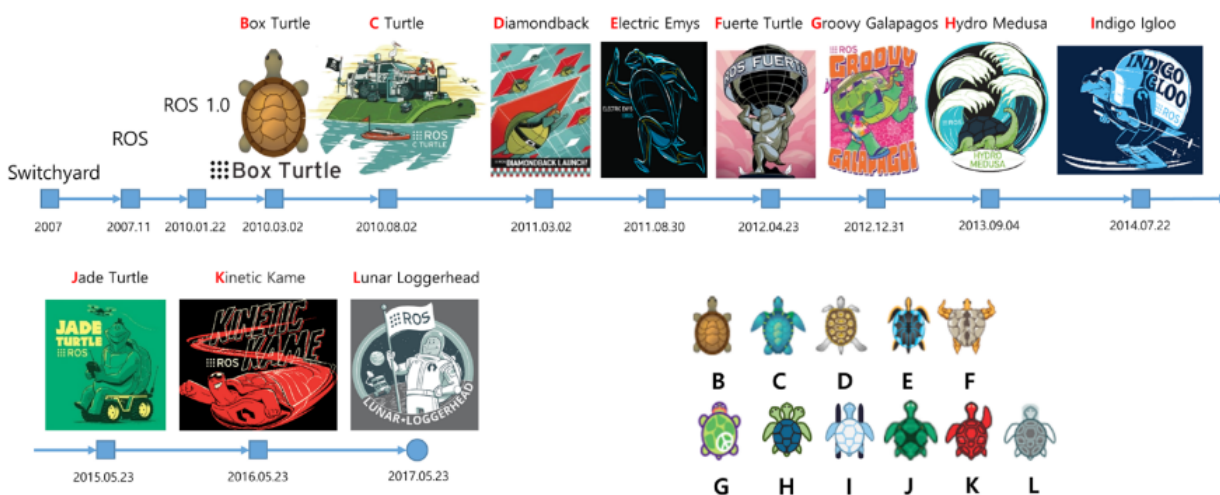
Uma distribuição ROS (WOODALL, 2018) é um conjunto versionado de pacotes ROS. As distribuições são semelhantes as do Linux, e o objetivo é permitir que os desenvolvedores trabalhem em uma base de código relativamente estável até que estejam prontos para

avançar. Uma vez que uma distribuição é lançada, mudanças ocorrem com correções de bugs e melhorias sem quebra para os pacotes principais. As Regras de liberação das distribuições são:

- Há uma liberação de ROS todos os anos em maio.
- Lançamentos em anos pares serão uma versão LTS, com suporte por cinco anos.
- Lançamentos em anos ímpares são edições normais do ROS, suportados por dois anos.
- As versões do ROS deixarão de oferecer suporte às distribuições do EOL Ubuntu, mesmo que a versão do ROS ainda seja suportada.

O ROS rotula a primeira letra de cada nome de lançamento em ordem alfabética e usa uma tartaruga como símbolo. Esse símbolo também é usado no tutorial oficial chamado “*turtlesim*” (PYO *et al.*, 2017). A Figura 5 mostra a evolução dos nomes e dos símbolos das distribuições ROS ao longo dos lançamentos, desde o seu princípio.

Figura 5 – Distribuições ROS ao longo do tempo



Fonte – Pyo *et al.* (2017)

Atualmente a [Open Robotics](#) vem desenvolvendo, operando e gerenciando o ROS. Mais recentemente, a 12ª versão do ROS, ROS Melodic Morenia, foi lançada em 23 de maio de 2018. A Figura 6 mostra as distribuições mais recentes lançadas do ROS.

Figura 6 – Distribuições recentes do ROS

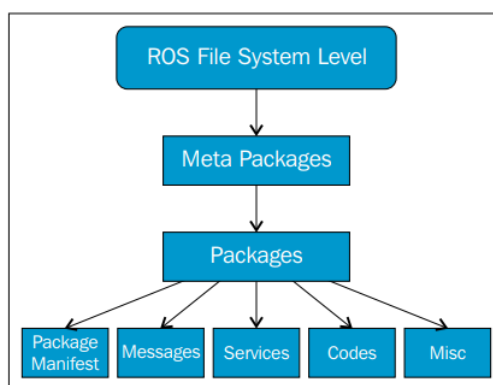
Distribuição	Data de lançamento	Pôster	Turtle tutorial	Data EOL
ROS Melodic Morenia	May 23rd, 2018			May, 2023 (Bionic EOL)
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame (Recommended)	May 23rd, 2016			April, 2021 (Xenial EOL)

Fonte – imagem adaptada de Woodall (2018)

O ROS é dividido em três níveis de conceitos: o nível do Sistema de Arquivos, o nível do Grafo de Computação e o nível da Comunidade (ROMERO, 2014). Esses três níveis são detalhados a seguir.

2.2.1 Sistemas de Arquivos

Figura 7 – Sistema de arquivos ROS



Fonte – Joseph (2015)

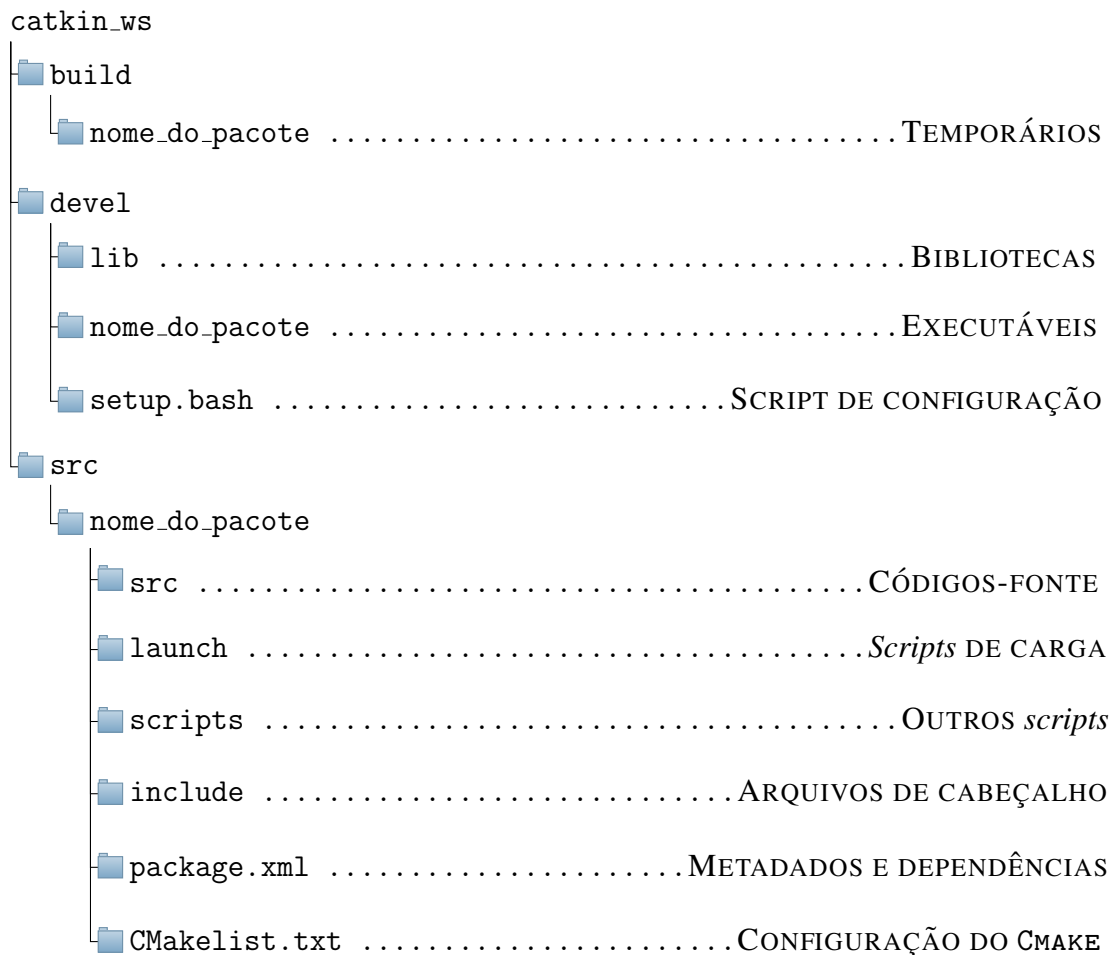
Como mostrado na Figura 7, o sistema de arquivos do ROS é dividido basicamente em meta-pacotes, pacotes, manifesto de pacotes, mensagens, serviços, códigos e arquivos diversos.

- **Metapackages (metapacotes):** *Metapackages* são pacotes especializados que servem apenas para representar um grupo de outros pacotes relacionados. Em versões mais antigas do ROS, como Electric e Fuerte, ele era chamado de *stacks* (pilhas), mas depois foi removido, à medida que os metapacotes passaram a existir. Um dos exemplos de um metapacote é a pilha de navegação⁴ do ROS (JOSEPH, 2015).

⁴ <http://wiki.ros.org/navigation>

- **Packages (pacotes):** os pacotes são a unidade principal para organizar o software no ROS. Um pacote pode conter processos executáveis (nós), uma biblioteca dependente do ROS, conjuntos de dados, arquivos de configuração ou qualquer outra coisa que seja organizada de maneira útil (ROMERO, 2014).

Estrutura de um pacote ROS (LAGES, 2017):



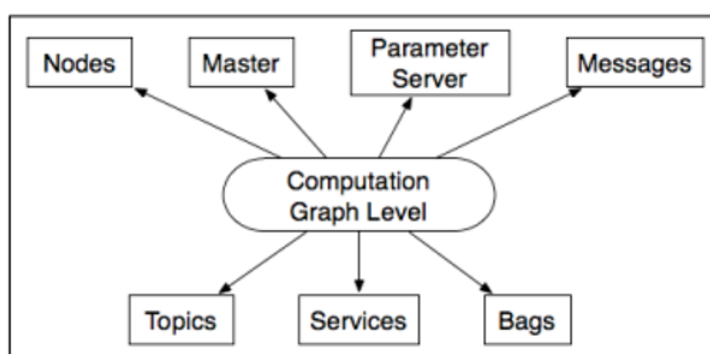
- **Package Manifests (pacotes manifestos):** fornecem metadados sobre um pacote, incluindo seu nome, versão, descrição, informações de licença, dependências e outras informações (ROMERO, 2014).
- **Tipos de mensagem:** são arquivos `.msg` que definem as estruturas de dados das mensagens enviadas no ROS (THOMAS, 2017).
- **Serviços:** são arquivos `.srv`, localizados no subdiretório `srv/` de um pacote, que descrevem os serviços no ROS (SAITO, 2017b).
- **Repositórios:** A maioria dos pacotes ROS é mantida usando um Sistema de Controle de Versão (VCS), como Git. A coleção de pacotes que compartilham um VCS comum pode ser chamada de repositórios. O pacote nos repositórios pode ser liberado usando uma ferramenta de automação de liberação catkin chamada bloom.(JOSEPH, 2017)

2.2.2 Grafo de Computação

Acessar, recuperar e processar um grande número de dados em paralelo é um dos princípios de um sistema operacional robótico. O conceito que o ROS utiliza, que permite gerenciar todas essas atividades de forma eficiente, é o do Grafo de Computação (*Computation Graph*), uma rede peer-to-peer de processos ROS que processam dados juntos (ROMERO, 2014). Qualquer nó no sistema pode acessar essa rede, interagir com outros nós, ver as informações que eles estão enviando e transmitir dados para a rede.

Enquanto o nível do Sistema de arquivos do ROS é um conceito estático, os conceitos do nível do Grafo de Computação são utilizados enquanto o sistema está rodando (MAZZARI, 2016). Os conceitos básicos desse nível são: nós, mestre, servidor de parâmetros, mensagens, tópicos, serviços e bolsas. Todos esses fornecendo dados para o grafo de maneira diferente (ROMERO, 2014).

Figura 8 – Conceitos básicos do Grafo de Computação do ROS



Fonte – Martinez e Fernández (2013)

2.2.2.1 Nós

Um nó é a menor unidade de processo em execução no ROS. O ROS recomenda criar um único nó para cada finalidade, desta forma cada nó é especializado em uma função. Por exemplo, um nó é responsável pelo acionamento do motor, outro pela conversão de dados do sensor, pela localização, visão gráfica e assim por diante (PYO *et al.*, 2017). Assim, um sistema que utiliza o ROS, normalmente possui vários nós, que executam diversas funções, podendo estar sendo executados numa mesma máquina ou em computadores distintos.

O uso de nós no ROS fornece vários benefícios para o sistema em geral. Existe tolerância a falhas, pois essas são isoladas em nós individuais. A complexidade do código é reduzida em comparação com sistemas monolíticos. Os nós expõem uma API mínima ao resto do Grafo de Computação, e implementações alternativas, mesmo em outras linguagens de programação, podem ser facilmente substituídas. Todos os nós em execução possuem um nome de recurso que os identifica exclusivamente para o restante do sistema (CONLEY, 2012a).

Na inicialização, um nó registra informações como nome, tipo de mensagem, endereço URI⁵ e número da porta do nó. O nó registrado pode atuar como um publicador (*publisher*), assinante (*subscriber*), servidor de serviço ou cliente de serviço, com base nas informações registradas. Os nós podem trocar mensagens usando tópicos e serviços (PYO *et al.*, 2017).

Cada nó deve ter um nome único, de modo a permitir que um nó se comunique com outro usando seu nome sem ambiguidade. Um nó pode ser escrito usando bibliotecas diferentes. Esse trabalho foi desenvolvido utilizando a biblioteca `rosjava`, que será abordada posteriormente (MARTINEZ; FERNÁNDEZ, 2013).

O ROS possui ferramentas de linha de comando para manipular e obter informações sobre os nós. O `rostopic` é uma dessas ferramentas. A Tabela 2 mostra as instruções suportadas por esse comando no terminal do SO onde o ROS estiver rodando.

Tabela 2 – Ferramenta `rostopic` do ROS

Comando	Descrição
<code>\$ rostopic info /node_name</code>	Imprime informações sobre o nó;
<code>\$ rostopic kill</code>	Finaliza um nó em execução;
<code>\$ rostopic list</code>	Lista os nós ativos;
<code>\$ rostopic machine <machine-name></code>	Lista os nós em execução em uma determinada máquina ou numa lista de máquinas;
<code>\$ rostopic ping /node_name --all</code>	Testa a conectividade com um nó ou com todos os nós;
<code>\$ rostopic cleanup</code>	Elimina o registro de nós inacessíveis.

Fonte: Saito (2013)

2.2.2.2 Mestre

O mestre ROS (*ROS Master*) fornece serviços de nomeação e registro para o restante dos nós no sistema ROS. A função do mestre é permitir que os nós ROS individuais se localizem entre si. Uma vez que esses nós se localizam, eles se comunicam *peer-to-peer* (WU, 2018).

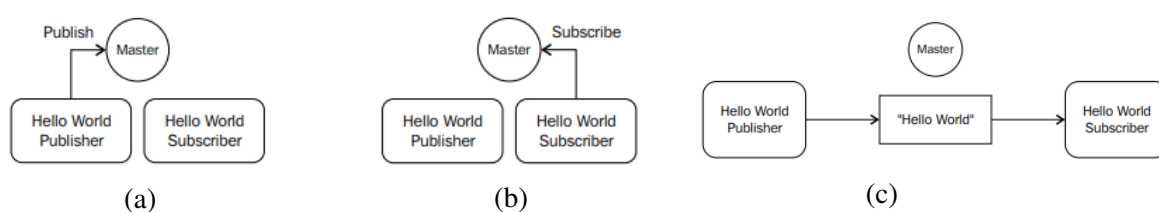
O mestre ROS funciona de forma parecida com um servidor DNS. Quando qualquer nó iniciar no sistema ROS, ele começará a procurar pelo *ROS Master* e registrará o nome do nó nele (JOSEPH, 2017). O mestre possui todos os detalhes sobre todos os nós em execução no ambiente do ROS. Ele trocará detalhes de um nó com outro para estabelecer uma conexão entre eles. Depois de trocar as informações, a comunicação será iniciada entre os dois nós ROS (MARTINEZ; FERNÁNDEZ, 2013). O mestre também fornece o *Servidor de parâmetros*, que será abordado posteriormente. O mestre é mais comumente executado usando o comando `roscore`, que carrega o mestre ROS juntamente com outros componentes essenciais (JOSEPH, 2015).

⁵ Endereço de um recurso disponível em uma rede

Quando executamos o ROS, o mestre será configurado com o endereço IP ou um *hostname* e a porta configurada na variável de ambiente `ROS_MASTER_URI`. Por padrão, o endereço URI usa o endereço IP do PC local e o número da porta 11311, a menos que seja modificado de outra forma (PYO *et al.*, 2017). Em um sistema distribuído apenas um mestre ROS é necessário, e esse deve ser executado em um computador acessível aos demais computadores, garantindo a requisição aos serviços pelo mestre (JOSEPH, 2015).

A Figura 9 mostra uma ilustração de como o ROS *Master* interage com um nó de publicação (*publisher*) e assinatura (*subscriber*). O nó do publicador publicando em um tópico de tipo de mensagem *string* e o nó do assinante se inscreve neste tópico (JOSEPH, 2015).

Figura 9 – Comunicação entre um nó *publisher* (a) e um *subscriber* (b) pelo ROS *Master*



Fonte – Joseph (2015)

Quando o nó do publicador começa a publicar a mensagem “Hello World” em um tópico específico, o ROS *Master* obtém os detalhes do tópico e os detalhes do nó. Ele procurará se algum nó está assinando o mesmo tópico. Se não houver nós assinando o mesmo tópico nesse momento, os dois nós permanecerão desconectados. Se houver um nó assinando um tópico que está sendo publicado, o ROS *Master* trocará os detalhes do publicador com o assinante e eles se conectarão, podendo assim trocar dados por meio de mensagens definidas pelo sistema ROS (JOSEPH, 2015).

2.2.2.3 Servidor de parâmetros

Além das mensagens, o ROS fornece outro mecanismo chamado parâmetros para obter informações para os nós. A ideia é que um servidor de parâmetros centralizado rastreie uma coleção de valores, permitindo que dados possam ser armazenados por chaves em um único local (FERREIRA, 2015). Esse servidor, como dito na subseção 2.2.2.2, é fornecido pelo mestre ROS. Os nós podem armazenar dados estáticos, como parâmetros de configuração, acessíveis pela rede ROS. Tipos de dados suportados incluem estrutura de dados e tipos primitivos comuns (MATHWORKS, 2018). Se o parâmetro de um nó tiver um escopo global, ele poderá ser acessado por todos os outros nós em tempo de execução.

2.2.2.4 Mensagens

O ROS utiliza um sistema de comunicação de processos através de mensagens. Um nó se comunica com outro passando uma mensagem, assim uma mensagem é a informação que

um processo passa para outro. As mensagens são estruturas de dados comuns, suportando tipos primitivos de dados padrões (inteiro, ponto flutuante, booleano). Mensagens são armazenadas em arquivos `.msg` que ficam em um subdiretório de um pacote. Nós podemos observar os detalhes de uma mensagem com a ferramenta de linha de comando `rosmmsg`. A [Tabela 3](#) mostra as descrições dos comandos suportados por essa ferramenta ([SAITO, 2017a](#)).

Tabela 3 – Ferramenta `rosmmsg` do ROS

Comando	Descrição
<code>\$ rosmmsg show <message-type></code>	Exibe os campos em um tipo de mensagem ROS;
<code>\$ rosmmsg list</code>	Exibe uma lista todas as mensagens;
<code>\$ rosmmsg package <package-name></code>	Exibe uma lista de todas as mensagens em um pacote;
<code>\$ rosmmsg packages</code>	Exibe uma lista de todos os pacotes com mensagens;
<code>\$ rosmmsg users <message-type></code>	Procura arquivos de código que usam o tipo de mensagem;
<code>\$ rosmmsg md5 <message-type></code>	Exibe a soma md5 ⁶ de uma mensagem

Fonte: [Saito \(2017a\)](#)

Figura 10 – Comando `rosmmsg` para mensagem `geometry_msgs/Twist`

```

caio@caio-linux: ~
caio@caio-linux:~$ rosmmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
float64 x
float64 y
float64 z
geometry_msgs/Vector3 angular
float64 x
float64 y
float64 z
caio@caio-linux:~$

```

Fonte – Captura de tela - Terminal Ubuntu

O pacote `geometry_msgs` fornece mensagens para primitivas geométricas comuns, como pontos, vetores e posições. Essas primitivas são projetadas para fornecer um tipo de dado comum e facilitar a interoperabilidade em todo o sistema ([ROS, 2018](#)). A [Figura 10](#) mostra o comando `rosmmsg` que detalha a mensagem `geometry_msgs/Twist`. Pode-se observar que,

⁶ Função *hash* usada principalmente para verificar a integridade de dados.

o tipo de mensagem `geometry_msgs/Twist` é uma mensagem composta de duas mensagens `geometry_msgs/Vector3`. Uma com componente linear para as velocidades nos eixos x, y e z, e outra com componente angular para a taxa angular em torno dos eixos x, y e z. Por padrão, as unidades de medidas das velocidades lineares e angulares são metros por segundo (m/s) e radiano por segundo (rad/s), respectivamente (FOOTE; PURVIS, 2014). A Tabela 4 mostra algumas das principais mensagens ROS e a área de aplicação desses recursos.

Tabela 4 – Principais mensagens do ROS

Package	Tipo de mensagem	Orientadas às áreas
<code>std_msgs</code>	<code>string</code> , <code>bool</code> , <code>int8</code> , <code>int16</code> , <code>char</code> , <code>float</code> , <code>MultiArrays</code>	Mensagens no formato comum da programação, como tipos de dados primitivos e outras construções básicas;
<code>geometry_msgs</code>	<code>twist</code> , <code>transform</code> , <code>pose</code> , <code>point</code>	Geometria primitiva como pontos, vetores, posições ou velocidades;
<code>nav_msgs</code>	<code>odometry</code> , <code>path</code> , <code>occupancyGrid</code>	Mensagens indicadas para a navegação;
<code>trajectory_msgs</code>	<code>JointTrajectory</code>	Orientação para junção de trajetórias;
<code>visualization_msgs</code>	<code>ImageMarker</code> , <code>InteractiveMarker</code>	Utilizadas na camada de alto nível, como a ferramenta <code>rviz</code> ;
<code>stereo_msgs</code>	<code>DisparityImage</code>	Específicas para processamento estéreo, como disparidade de imagens;
<code>sensor_msgs</code>	<code>Image</code> , <code>PointCloud</code> , <code>LaserScan</code>	Definidas para usar em sensores, como câmaras, laser scan, infravermelhos;

Fonte: Santos (2013)

2.2.2.5 Tópicos

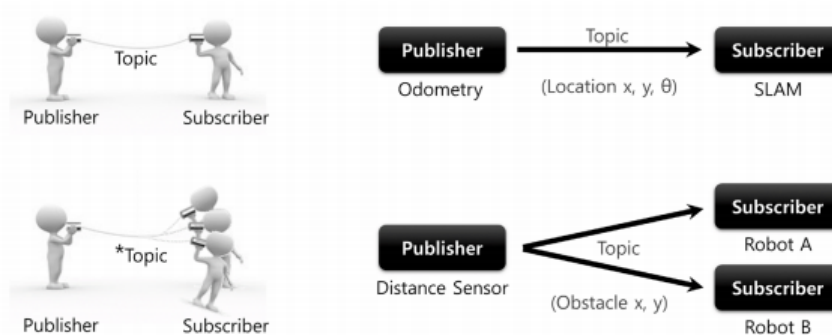
Um dos métodos para se comunicar e trocar mensagens ROS entre dois nós é chamado de tópicos ROS. Cada tópico tem um nome exclusivo e qualquer nó pode acessar esse tópico e enviar dados por meio dele, desde que tenham o tipo de mensagem correto (JOSEPH, 2017). Assim, os tópicos são fortemente tipados pelo tipo de mensagem ROS. Quando um nó envia uma mensagem através de um tópico, dizemos que o nó está publicando (*publishing*) um tópico. Quando um nó recebe uma mensagem através de um tópico, podemos dizer que o nó está assinando ou se inscrevendo (*subscribing*) em um tópico.

Os nós ROS não estão interessados em saber qual nó está publicando ou assinando o tópico, ele procura apenas o nome do tópico e se os tipos de mensagem estão correspondentes. Os tópicos são destinados para comunicação unidirecional de fluxo contínuo, para implementar chamadas de procedimento remoto como comunicação, é preciso usar os Serviços ROS (FOUROHER, 2014).

Como os tópicos são unidirecionais e permanecem conectados para enviar ou receber mensagens continuamente, esta comunicação é adequada para a transmissão dos dados de sensores, que exigem a publicação periódica de mensagens. Além disso, múltiplos os assinantes

podem receber mensagens de um publicador e vice-versa, o que facilita que um sensor possa ser acessado ao mesmo tempo por diferentes nós (PYO *et al.*, 2017).

Figura 11 – Comunicação de mensagens ROS por tópicos



Fonte – Pyo *et al.* (2017)

O ROS suporta atualmente o transporte de mensagens baseado nos protocolos da camada de transporte TCP e UDP, e no protocolo da camada de rede IP. O transporte baseado nesses protocolos é conhecido como TCPROS e transmite dados de mensagens por meio de conexões TCP/IP persistentes. TCPROS é o transporte padrão usado no ROS e é o único transporte que as bibliotecas clientes devem oferecer suporte (FOROUHER, 2014). O UDPROS é a camada de transporte para mensagens e serviços ROS baseado no protocolo UDP. Ele ainda está em desenvolvimento e está disponível apenas na biblioteca com implementação do C++, `roscpp`. Ele é útil quando a latência é mais importante que o transporte confiável, por exemplo em um *streaming* de áudio (UDPROS, 2013).

O ROS possui a ferramenta de linha de comando `rostopic` que exibe informações sobre os tópicos. A Tabela 5 mostra os comandos suportados por essa ferramenta.

Tabela 5 – Ferramenta `rostopic` do ROS

Comando	Descrição
\$ <code>rostopic bw /topic_name</code>	Exibe a largura de banda usada pelo tópico fornecido;
\$ <code>rostopic delay /topic_name</code>	Exibe atraso do tópico conforme <i>timestamp</i> do cabeçalho;
\$ <code>rostopic echo /topic_name</code>	Exibe os dados publicados em um tópico;
\$ <code>rostopic find <msg-type></code>	Localiza tópicos por tipo de mensagem;
\$ <code>rostopic hz /topic_name</code>	Exibe a taxa de publicação de um tópico.;
\$ <code>rostopic info /topic_name</code>	Exibe informações sobre o tópico ativo;
\$ <code>rostopic list</code>	Lista os tópicos ativos;
\$ <code>rostopic pub /topic_name</code>	Publica dados no tópico;
\$ <code>rostopic type /topic_name</code>	Exibe o tipo do tópico

Fonte: Forouher (2014)

2.2.2.6 Serviços

O modelo de envio de mensagens através dos tópicos (“publicar” e “subscriver”) é um método assíncrono que é vantajoso na transmissão intermitente de dados. Porém, muitas vezes existe uma necessidade de comunicação síncrona que usa solicitação e resposta (CONLEY, 2012b). O ROS fornece um método de comunicação de mensagens sincronizadas chamado “serviço”, uma comunicação síncrona bidirecional entre o cliente de serviço que está solicitando um serviço e o servidor de serviço que responde à solicitação (PYO *et al.*, 2017).

“Apesar do modelo de publicação/subscrição ser um paradigma de comunicação muito flexível, o seu modelo de comunicação de N para N participantes unidirecional não é apropriado para interações pedido/resposta, que são muitas vezes necessárias num sistema distribuído. Este tipo de interações é por isso feito através de serviços, que são definidos por um par de estruturas de mensagens, um para o pedido e outro para a resposta” (FERREIRA, 2015, p. 14).

Semelhante às definições de mensagem usando o arquivo `.msg`, temos que definir os serviços em outro arquivo chamado `.srv`, que deve ser mantido dentro do subdiretório `/srv` do pacote (JOSEPH, 2015). Um servidor de serviço ROS, por exemplo um nó que informa a hora atual (Figura 12), responde apenas quando há uma solicitação de um cliente de serviço. Esse pode enviar solicitações, bem como receber respostas. O nó cliente deve esperar até que o servidor responda com os resultados. Quando a solicitação e a resposta do serviço forem concluídas, a conexão entre os dois nós será encerrada. (PYO *et al.*, 2017).

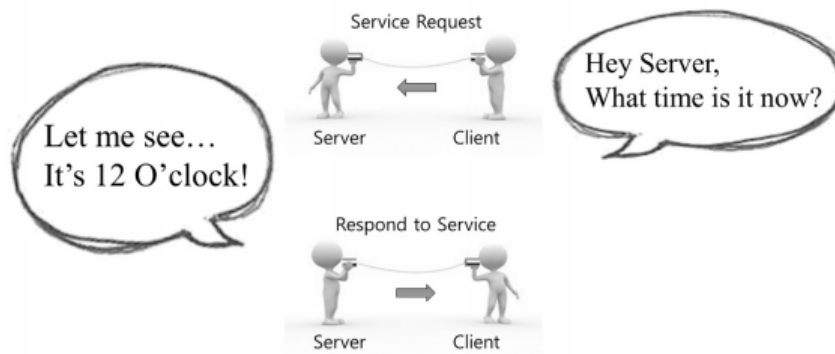
Como explicado anteriormente, chamadas de serviço são bidirecionais. Um nó envia informações para outro nó e aguarda uma resposta. A informação flui em ambas as direções. Ao contrário de uma mensagem, que é publicada e não há nenhum conceito de resposta e nem garantia de que alguém esteja se inscrevendo nessas mensagens (O’KANE, 2014).

Como o serviço não mantém a conexão, ele é útil para reduzir a carga da rede em comparação com os tópicos que mantêm a transmissão constantemente. Um serviço é frequentemente usado para comandar um robô para executar uma ação específica ou nós para executar determinados eventos com uma condição específica (PYO *et al.*, 2017).

2.2.2.7 Bolsas

Podemos gravar os dados de mensagens ROS em arquivos `.bag` chamados de bolsas (*bags*), e reproduzi-los quando necessário recriar o ambiente de quando as mensagens foram gravadas. Por exemplo, ao realizar um experimento de robô usando um sensor, os valores do sensor são armazenados no formulário de mensagem usando as *bags*. Esta mensagem gravada pode ser repetidamente carregada sem realizar o mesmo teste, reproduzindo o arquivo de bolsa salvo (PYO *et al.*, 2017). A ferramenta de linha de comando `rosbag` fornece funcionalidade para leitura/gravação de bolsas ROS. As funções de gravação e reprodução do `rosbag` são úteis ao desenvolver um algoritmo com modificações frequentes no programa e facilita a depuração (SPRICKERHOF, 2015).

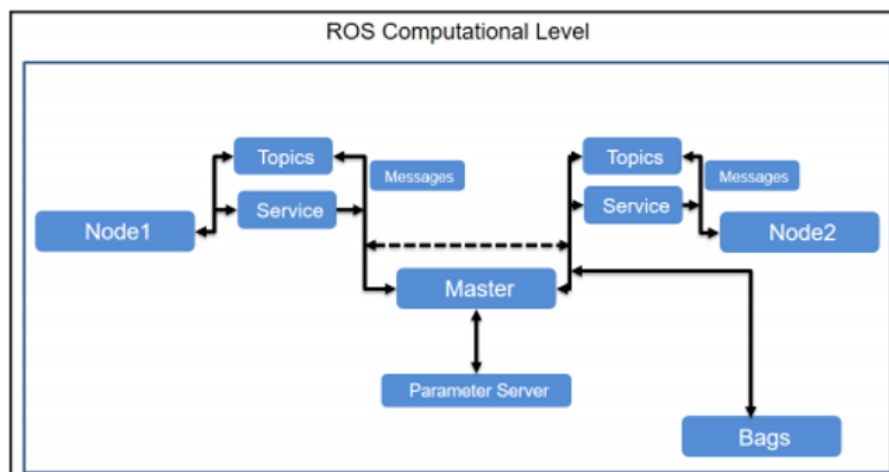
Figura 12 – Comunicação por serviços ROS



Fonte – [Pyo et al. \(2017\)](#)

A [Figura 13](#) mostra um diagrama que relaciona os conceitos do nível do Grafo de Computação do ROS.

Figura 13 – Diagrama dos conceitos do Grafo de Computação ROS



Fonte – [Joseph \(2017\)](#)

2.2.2.8 Visualizando o grafo ROS

A relação entre nós, tópicos, publicadores e assinantes apresentadas acima, pode ser visualizada através da ferramenta de linha de comando `rqt_graph`. Através dela, podemos visualizar graficamente o grafo de conexões entre os nós ROS. A representação gráfica da comunicação de mensagens não inclui o serviço, uma vez que esse só acontece uma vez ([PYO et al., 2017](#)).

A ferramenta `rqt_graph` é encontrada no pacote `rqt`, podemos instalá-la pelo terminal através dos comandos ([EDNO, 2014](#)):

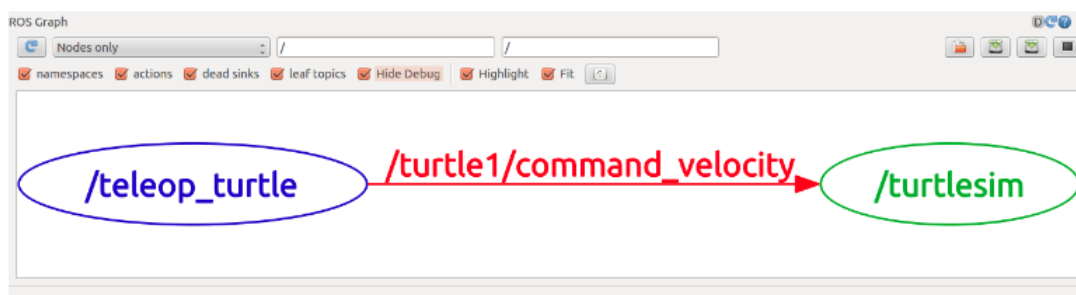

```
$ sudo apt-get install ros-<distro>-rqt
$ sudo apt-get install ros-<distro>-rqt-common-plugins
```

Executamos o `rqt_graph` através do comando:

```
$ rosrun rqt_graph rqt_graph
```

A [Figura 14](#) mostra a execução da ferramenta `rqt_graph`. Os nós `/teleop_turtle` e `/turtlesim` estão se comunicando através do tópico `/turtle1/command_velocity`. Os círculos representam os nós. Destacados em vermelho, a seta indica a transmissão da mensagem e o nome do tópico é dado logo acima dessa. Podemos perceber que o nó `/teleop_turtle` está publicando no tópico `/turtle1/command_velocity` e o nó `/turtlesim` se inscrevendo nesse tópico. Desta forma, o `rqt_graph` se torna muito útil para verificar a correlação de nós na rede ROS.

Figura 14 – Ferramenta `rqt_graph`



Fonte – Edno (2014)

2.2.3 Comunidade

Os conceitos do nível da comunidade ROS são recursos ROS que permitem que comunidades separadas troquem software e conhecimento. Esses recursos incluem ([ROMERO, 2014](#)):

- **Distribuições:** Distribuições ROS são coleções versionadas de pacotes ROS, como distribuições Linux. As distribuições ROS facilitam a instalação de uma coleção de softwares e também mantêm versões consistentes através de um conjunto de softwares ([MARTINEZ; FERNÁNDEZ, 2013](#)).
- **Repositórios:** O ROS conta com uma rede de repositórios de código, onde diferentes instituições podem desenvolver e liberar seus próprios componentes de software do robô ([ROMERO, 2014](#)).
- **Wiki ROS:** Centro de conhecimento do ROS, no qual qualquer pessoa pode criar documentação para seus pacotes. Pode-se encontrar documentação padrão e tutoriais sobre o ROS no [Wiki ROS](#) ([JOSEPH, 2017](#)).

- **Bug ticket system:** Recurso utilizado para relatar erros encontrados no sistema ou adicionar um novo recurso (JOSEPH, 2015).
- **Lista de discussão:** Principal canal de comunicação sobre novas atualizações para o ROS, bem como um fórum para fazer perguntas sobre o software ROS (ROMERO, 2014).
- **ROS Answers:** Site onde os usuários podem fazer perguntas sobre ROS e áreas relacionadas (<https://answers.ros.org/>) (ROMERO, 2014).
- **Blog ROS:** O blog do ROS fornece atualizações regulares sobre a comunidade ROS com fotos e vídeos (<http://www.ros.org/news>) (JOSEPH, 2017).

2.2.4 Rosjava

Rosjava é uma implementação pura de Java para o ROS, criada e mantida pelo Google e pela *Willow Garage*⁷, inicialmente. Em vez de ter uma biblioteca cliente em Java dando acesso ao núcleo do ROS, que é escrito em C++, o projeto Rosjava reescreveu totalmente o núcleo do ROS em Java. O objetivo do Google é ter uma versão do ROS totalmente compatível com o Android (MAZZARI, 2016).

Essa interface entre o Android e o ROS possibilitou que dispositivos móveis pudessem integrar a rede ROS com maior facilidade e suporte. Os dispositivos móveis atuais possuem poderosos recursos de processamento e sensoriamento. Assim, foi possível aproveitar esses recursos na rede ROS para, por exemplo, processar dados recebidos de um robô e aproveitar os sensores integrados dos *smartphones* ou *tablets* para executar funcionalidades como controle, mapeamento e localização.

2.2.5 Outros conceitos ROS

Outros conceitos do ROS também são pertinentes para trabalhar com o sistema, sendo necessário pelo menos um entendimento básico deles. A seguir é dada a descrição de alguns dos mais importantes.

2.2.5.1 Catkin

Catkin é o sistema de construção do ROS. Um sistema de compilação que basicamente usa o *CMake* (*Cross Platform Make*)⁸ e é descrito no arquivo `CMakeLists.txt` na pasta do pacote. O *CMake* foi modificado no ROS para criar um sistema de compilação específico do ROS. O *Catkin* começou a ser trabalhado na versão ROS *Fuerte* e aplicado à maioria dos pacotes a partir da versão ROS *Hydro*. O sistema de compilação do *Catkin* facilita o uso de compilações relacionadas ao ROS, gerenciamento de pacotes e dependências entre pacotes. O sistema de compilação anterior ao *Catkin* é o ROS *Build*, e atualmente não é oficialmente recomendado seu uso (PYO *et al.*, 2017).

⁷ Foi um laboratório de pesquisa em robótica dedicado ao desenvolvimento de hardware e software de código aberto para aplicações de robótica pessoal.

⁸ Um programa para construir pacotes de instalação.

2.2.5.2 Variáveis de ambiente ROS

Existem muitas variáveis de ambiente que podem ser definidas para afetar o comportamento do ROS. Destas, as mais importantes são `ROS_MASTER_URI`, `ROS_ROOT`, `ROS_IP` e `ROS_HOSTNAME` (ROS, 2016).

- `ROS_MASTER_URI` - Essa variável de ambiente contém o IP e a porta do ROS *Master*. Usando essa variável, os nós ROS podem localizar o mestre. Se esta variável estiver errada, a comunicação entre os nós não ocorrerá.
- `ROS_ROOT` - Define o local onde os pacotes principais do ROS estão instalados.
- `ROS_IP/ROS_HOSTNAME` - O ROS precisa saber o nome de cada máquina em que está sendo executado. Isso é feito configurando a variável de ambiente `ROS_IP` ou `ROS_HOSTNAME` de cada máquina com seu próprio endereço IP ou nome. As opções são mutuamente exclusivas, se ambas forem configuradas, `ROS_HOSTNAME` terá precedência.

2.2.5.3 roscore

`roscore` é o comando que executa o mestre ROS. Se vários computadores estiverem na mesma rede, ele poderá ser executado em qualquer computador da rede. No entanto, com exceção do caso especial que suporta múltiplos `roscore`, apenas um `roscore` deve estar em execução na rede. Quando o mestre do ROS está em execução, o endereço do URI e o número da porta designado para as variáveis de ambiente `ROS_MASTER_URI` são usados. Se o usuário não tiver configurado a variável de ambiente, o endereço IP local atual será usado como o endereço URI e o número de porta 11311 será usado, que é um número de porta padrão para o mestre (PYO *et al.*, 2017).

2.2.5.4 rosrn

`rosrn` é o comando básico de execução do ROS. É usado para executar um único nó no pacote. O nó usa a variável de ambiente `ROS_HOSTNAME`, armazenada no computador no qual o nó está sendo executado, como o endereço URI e a porta é configurada com um valor exclusivo arbitrário. Desta forma, o `rosrn` permite rodar um executável em um pacote aleatório de qualquer lugar sem ter que dar seu caminho completo (PYO *et al.*, 2017).

2.2.5.5 roslaunch

Enquanto `rosrn` é um comando para executar um único nó, o `roslaunch` executa vários nós. É um comando ROS especializado em execução de nós com funções adicionais, como alterar parâmetros de pacotes ou nomes de nós e alterar variáveis de ambiente ao executar nós. O `roslaunch` usa o arquivo `.launch` para definir quais nós devem ser executados. O arquivo é baseado em XML e oferece uma variedade de opções na forma de *tags* XML. Quando é usado o `roslaunch` o `roscore` é executado automaticamente (PYO *et al.*, 2017).

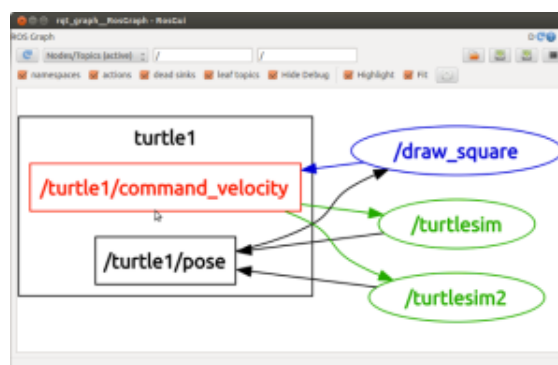
2.2.5.6 Names

Nós, parâmetros, tópicos e serviços, todos têm nomes. Esses nomes são registrados no mestre e buscados para transferir mensagens ao usar os parâmetros, tópicos e serviços de cada nó. Os nomes são flexíveis porque podem ser alterados ao serem executados e diferentes nomes podem ser atribuídos ao executar nós, parâmetros, tópicos e serviços idênticos várias vezes. O uso de nomes torna o ROS adequado para projetos de grande escala e sistemas complexos (PYO *et al.*, 2017).

2.2.5.7 Namespaces

Um *namespace* pode ser visto como um diretório cujo conteúdo é um item de nome diferente. Esses itens podem ser nós, tópicos ou até mesmo outros *namespaces*. *Namespaces* podem ser organizados em hierarquias. Na Figura 15 por exemplo, existe o *namespace* raiz, referido por uma barra “/”. O *namespace* raiz da Figura 15 inclui quatro itens. Três deles são nós (*draw_square*, *turtlesim* e *turtlesim2*), enquanto o quarto é o *namespace* *turtle1*. O *namespace* *turtle1* inclui dois itens que são os dois tópicos: *command_velocity* e *pose* (NOOTRIX, 2018).

Figura 15 – *Namespaces* no *rqt_graph*



Fonte – Nootrix (2018)

2.3 SLAM

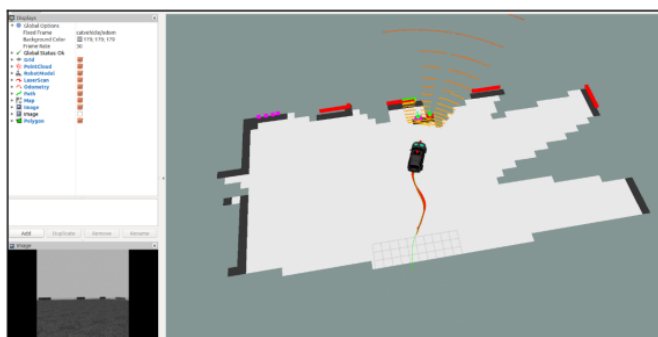
Localização e Mapeamento Simultâneos (SLAM, da sigla em inglês) é uma técnica importante que permite ao robô reconhecer os obstáculos em torno dele e localizar-se. Quando combinado com alguns outros métodos, como o planejamento de caminhos, é possível permitir que robôs naveguem em ambientes desconhecidos. Essa técnica consiste em mapear um ambiente ao mesmo tempo em que o robô está se movendo, ou seja, enquanto o robô navega através de um ambiente, ele reúne informações do meio ambiente através de seus sensores e gera um mapa. Desta forma é possível ter uma base móvel capaz não só de gerar um mapa de um ambiente desconhecido, bem como atualizar o mapa existente, permitindo assim a previsão de colisões em ambientes dinâmicos (KOUBAA, 2016).

O processo SLAM consiste em vários passos. O objetivo desse processo é usar o ambiente para atualizar a posição do robô. Como a odometria do robô (que dá a posição do robô) é muitas vezes errônea, não podemos confiar diretamente nela. Desta forma, sensores de distância, como sensores ultrassônicos, detectores de alcance a laser e *scanners* de infravermelho, são usados para corrigir a posição do robô. Isso é feito extraindo recursos do ambiente e observando novamente quando o robô se movimenta. Isso faz com que a tecnologia SLAM seja basicamente uma questão de medição e matemática (RIISGAARD; BLAS, 2005).

Figura 16 – Carro robô utilizando SLAM



(a) Simulação de um carro robô em um ambiente urbano



(b) Construção de um mapa 2D usando SLAM

Fonte – Joseph (2017)

A Figura 16(a) mostra a execução de um carro robô com sensores de navegação num ambiente urbano simulado e a Figura 16(b) mostra o mapa 2D desse ambiente sendo construído utilizando o SLAM.

2.4 Robótica móvel

Segundo Santos (2008), um robô é um sistema autônomo que pode ser capaz de sentir seu ambiente e agir para atingir objetivos. Um robô móvel acrescenta o fato de que não está confinado a um local específico, pois tem a capacidade de se mover em seu ambiente. A principal característica que define um robô autônomo é a capacidade de agir com base em

suas próprias decisões, e não através do controle de um humano. No entanto, há ainda um limite na complexidade das tarefas que os robôs podem executar de forma autônoma. Às vezes, é vital que um ser humano seja capaz de controlar o robô de uma distância. Isto levou ao desenvolvimento de robôs teleoperados, onde um operador humano emite comandos para o robô enquanto tem acesso perceptivo ao espaço de trabalho. Algumas das áreas de aplicação mais comuns de teleoperação são: exploração espacial, veículos submarinos, mineração, agricultura, vigilância, resgate, cirurgia e entretenimento (GLOVER *et al.*, 2009).

2.4.1 Drones

O uso potencial de robôs voadores em aplicações militares e civis e os desafios por trás de seu desenvolvimento estão atraindo a comunidade científica e industrial. Graças a isto, os veículos aéreos não tripulados (VANT), também conhecidos como Drones, tornaram-se consideravelmente populares. Hoje, eles estão sendo usados principalmente para tarefas de vigilância e inspeção. No entanto, avanços recentes em processadores embarcados de baixa potência, sensores em miniatura e teoria de controle, estão abrindo novos horizontes em termos de miniaturização e campos de uso (BECKER *et al.*, 2014). Os Drones podem operar de forma autônoma, mas também são constantemente trabalhados através de teleoperação.

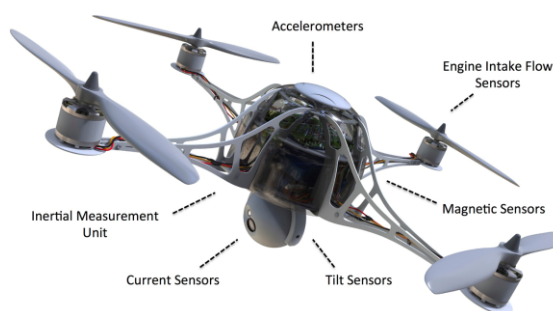
2.4.1.1 Quadrotores

Com diversos tamanhos, os Quadrotores, também referidos como Helicóptero Quadrotor, Quadricópteros e Quadrirrotores, são um tipo de VANT que possuem topologia semelhante a um helicóptero. No formato padrão, possuem quatro motores, sendo duas hélices girando no sentido horário, com as outras duas girando no sentido anti-horário, permitindo que o veículo suba verticalmente, flutue no ar e voe em uma direção designada, o que o torna um veículo altamente manobrável (O'HORA, 2017). Devido sua estrutura simples e características de operação, os quadrotores foram popularizados para pesquisa e aplicações de VANTs, como mapeamento, busca e resgate, entretenimento comercial e pessoal. Um quadrotor pode ser equipado com sensores de distância e ser usado, por exemplo, para mapear locais de acesso difícil ou inseguro para o homem ou ainda se locomover em meio a destroços de um prédio que desabou, equipado com câmeras e sensores térmicos que auxiliam no resgate de sobreviventes.

2.5 Simulação robótica

A simulação é um processo para desenvolver um modelo virtual capaz de emular o processo do mundo real. Aplicado em robótica, o processo de simulação é usado para criar um modelo virtual de um robô, incluindo o código de projeto e programação (ROBOTICS, 2014). O aumento exponencial do poder de processamento dos computadores e de seus recursos gráficos, mudou drasticamente a paisagem no campo da simulação robótica. Os ambientes de simulação de robôs se tornaram mais acessíveis e fieis as necessidades dos projetos de robótica. Hoje, é

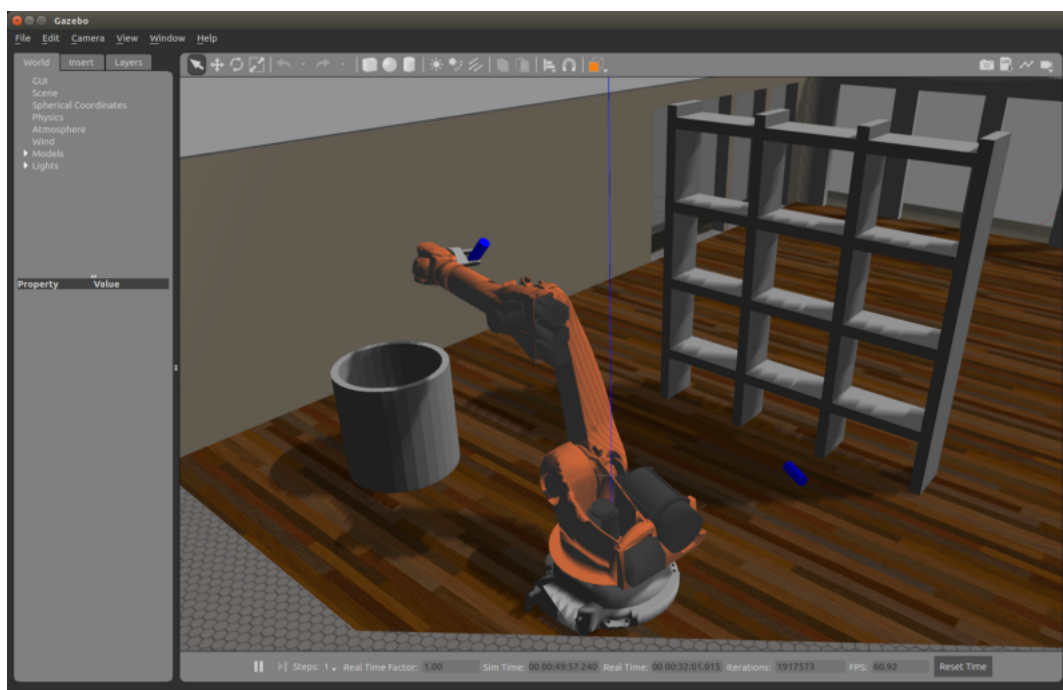
Figura 17 – Quadrotor

Fonte – [Winkler \(2016\)](#)

possível encontrar diversos softwares gratuitos para simulação robótica, dentre esses destacam-se o [V-Rep](#), [MORSE](#), [ARS](#) e [Gazebo](#), sendo o último utilizado no desenvolvimento da aplicação desse trabalho e abordado de forma mais detalhada posteriormente.

Dentre diversas vantagens como redução de custo e riscos para integridade física de desenvolvedores, os softwares de simulação de robôs garantem que possíveis falhas não sejam projetadas no sistema desenvolvido e que o sistema robótico funcione exatamente como previsto. Uma vez que a simulação é concluída, o resultado final é apenas um carregamento do sistema desenvolvido para a aplicação robótica real. Desta forma, o sistema real irá corresponder exatamente à simulação.

Figura 18 – Simulação de um braço robô no Gazebo

Fonte – [Hortovanyi \(2017\)](#)

3 METODOLOGIA

Navegação é um elemento essencial de muitas operações de robôs móveis. Planejar e executar um trajeto de um ponto de partida até um destino, é uma atividade frequente no uso de robôs em diversas áreas. A navegação requer conhecimento do ambiente que se deseja navegar, seja por parte dos robôs, no caso de navegações autônomas, ou por parte dos operadores, em situações de teleoperação de robôs. A navegação de robôs em ambientes desconhecidos fornece um problema único, pois não existe conhecimento do espaço de trabalho. Assim, técnicas como o SLAM são bastante utilizadas com robôs móveis no mapeamento de ambientes desconhecidos. Foi visto na [seção 2.4](#), que a principal função de um sistema de teleoperação é apoiar o operador na realização de tarefas complexas, de forma que o operador emita comandos para o robô enquanto tem acesso perceptivo ao espaço de trabalho. Para operar um robô de maneira eficiente em distâncias remotas, é importante que o operador esteja ciente do ambiente ao redor do robô, para que o operador possa dar instruções precisas ao robô. Desta forma, a elaboração de bons sistemas de teleoperação é fundamental para o sucesso de operações com robôs móveis em qualquer tipo de ambiente.

O presente trabalho propõe compor os dois paradigmas abordados no parágrafo anterior - navegação e teleoperação - e os conceitos apresentados no [Capítulo 2](#), aplicados a uma aplicação Android, denominada “*Teleoperação Quadrotor ROS*”, para teleoperação de um quadrotor utilizando o ROS, que possibilite uma navegação em ambientes conhecidos ou desconhecidos, através da imagem capturada pelo quadrotor e pelo mapa do ambiente. No caso de ambientes desconhecidos sem o mapa, realiza-se sua construção empregando os pacotes de SLAM fornecidos pelo ROS.

3.1 Trabalhos relacionados e proposta

Ao longo do estudo realizado para esse trabalho, foram pesquisadas ferramentas construídas para o sistema ROS que abordaram os conceitos vistos no [Capítulo 2](#) e que serviram de contribuição para o desenvolvimento final da aplicação proposta.

[Joseph \(2017, p. 163\)](#) apresenta uma aplicação Android desenvolvida por [Michael T Brunson](#), nomeada “*ROS Control*”, para teleoperação e visualização de dados de sensores para uma plataforma robótica, suportada para o ROS, chamada HUSKY. O HUSKY é um veículo terrestre de tamanho médio, não tripulado, para uso externo, adequado para aplicações de pesquisa e prototipagem rápida ([ROBOTICS, 2018](#)). A aplicação “*ROS Control*” fornece uma interface completa para obter informações sobre direcionamento e navegação do HUSKY. O aplicativo processa dados de vários sensores no robô e exibe as informações de forma compreensível para o usuário. É possível ainda, inserir múltiplos perfis de robôs no aplicativo. Isso permite executar diferentes configurações programadas no HUSKY. O aplicativo no entanto se limita a um único

tipo de plataforma robótica, de forma que o controle de outros modelos de robôs, como no caso dos Quadrotores, se torna inviável. Assim, a aplicação desse presente trabalho propõe compor a visualização de sensores de robôs na interface de operação, também para um Quadrotor ROS.

[Figueiredo \(2016\)](#) propõe uma aplicação Android voltada para a teleoperação de Quadrotores. A aplicação permite controlar esse modelo de robô remotamente e visualizar o ambiente remoto na mesma interface. O aplicativo fornece ao usuário funcionalidades para o controle completo do quadrotor e exibe na mesma interface a imagem da câmera acoplada no robô. O aplicativo desenvolvido nesse presente trabalho, propõe utilizar a mesma ideia de percepção do espaço de navegação através do sensor da câmera do robô, proposta por [Figueiredo \(2016\)](#), e aprimorar as funcionalidades de controle do *drone*. Através de dois controles virtuais Joystick e do uso do sensor acelerômetro do *smartphone*, foi proposto uma operação mais natural e intuitiva do robô, em relação ao uso de botões feito na aplicação desenvolvida por [Figueiredo \(2016\)](#). Para fornecer mais informações espaciais ao operador, foi colocado também uma opção de visualização, em tempo de execução, do mapa do ambiente de navegação do drone, construído através do SLAM. Sendo possível também, salvar esse mapa construído no formato digital de imagens *GeoTiff*.

3.2 Ferramentas utilizadas

Para criação da aplicação, foi necessário recorrer a tecnologias e ferramentas que deram suporte ao desenvolvimento e teste do aplicativo.

3.2.1 Hardware

Em relação as ferramentas de hardware foram utilizados um *smartphone* Motorola Moto G2, com o sistema operacional Android Nougat 7.1 e um notebook da marca *Evolute* rodando o SO Ubuntu 16.04. As especificações técnicas desses equipamentos são dadas a seguir.

- **Motorola Moto G2** : processador *Quad Core* com 1.2GHz de velocidade, 1GB de memória RAM, 8GB de memória interna e tela de 5 polegadas.
- **Notebook Evolute SPX-65**: processador *Intel Core i5-2520M* com 2,50GHz de velocidade, 8GB de memória RAM e placa gráfica *Intel HD Graphics 3000*.

3.2.2 ROS *Kinetic*

O uso do ROS nesse trabalho se deu devido as vantagens já apresentadas na [seção 2.2](#), ao grande crescimento desse sistema nos últimos anos e o fato desse possuir uma licença de código aberto que traz, entre outros benefícios, um alto e crescente número de colaboradores em todo mundo. A utilização do ROS ainda proporciona um bom ambiente integrado de simulação para desenvolvimento e torna fácil a transposição do ambiente simulado para o concreto, de modo que o sistema desenvolvido nesse trabalho pode ser tanto utilizado no simulador quanto

no robô real. A distribuição do ROS usada no desenvolvimento da aplicação foi a *Kinetic*, uma versão LTS que possui, assim, maior suporte aos principais pacotes ROS e vasta documentação frente as outras versões.

3.2.3 Android Studio

O Android Studio é a IDE (*Integrated Development Environment*) oficial para o desenvolvimento de aplicativos para Android, baseado no IntelliJ IDEA¹. Além do poderoso editor de código e das ferramentas de desenvolvedor do IntelliJ, o Android Studio oferece diversos recursos que aumentam a produtividade ao criar aplicativos para Android, como: um sistema de compilação flexível baseado no Gradle, um emulador rápido e rico em recursos, um ambiente unificado onde é possível desenvolver para todos os dispositivos Android, modelos de código e integração com o GitHub que ajuda a criar recursos comuns de aplicativos e importar códigos de amostra (ANDROID, 2018a).

3.2.4 Gazebo

Conforme visto na [seção 2.5](#), a simulação de robô é essencial nos projetos de robótica. O Gazebo é um programa de simulação robótica, gratuito, que oferece a capacidade de simular com precisão e eficiência diversos modelos de robôs em ambientes internos e externos complexos (GAZEBO, 2018).

As distribuições ROS vêm por padrão com uma versão do Gazebo já integrada no sistema. O ROS Kinetic por exemplo, que foi utilizado nesse trabalho, vem com a versão 7 do Gazebo. No entanto, o Gazebo oferece um pacote que integra o ROS de forma independente chamado `gazebo_ros_pkgs`. Esse pacote fornece as interfaces necessárias para simular um robô no Gazebo usando mensagens e serviços ROS, e mantém dependências que não possuem ligações diretas com o ROS, melhorando o suporte ao Gazebo e limpando códigos antigos de versões anteriores do ROS e do Gazebo (GAZEBO, 2014).

3.2.5 Linguagens de programação

3.2.5.1 Java

Aplicações nativas Android podem ser escritas utilizando as linguagens C/C++, Kotlin ou Java. A mais comumente usada é o Java. Java é uma popular linguagem de programação orientada a objetos desenvolvida pela *Sun Microsystems*. Um dos principais objetivos dessa linguagem é poder escrever programas que serão executados em uma grande variedade de sistemas de computadores e dispositivos. Programas escritos em Java são convertidos para os chamados Bytecodes que são executados pela JVM, a máquina virtual do Java. Os códigos de bytecodes são independentes de plataforma, eles não dependem de uma plataforma de hardware específica.

¹ Ambiente de desenvolvimento integrado (IDE) Java para desenvolvimento de software.

Assim, os bytecodes do Java são portáveis, sem recompilar o código-fonte, os mesmos bytecodes podem ser executados em qualquer plataforma que contenha uma JVM que entenda a versão do Java na qual os bytecodes foram compilados (DEITEL; DEITEL, 2012).

Para poder desenvolver as aplicações Android em Java é necessário a instalação do JDK (Kit de Desenvolvimento Java). O JDK inclui ferramentas úteis para desenvolver e testar programas escritos na linguagem de programação Java e em execução na plataforma Java.

3.2.5.2 XML

A Linguagem de Marcação Extensível (da sigla inglês XML) é uma linguagem de programação usada para codificação simples de documentos. Essa linguagem é outra das ferramentas básicas usadas na criação da maioria dos aplicativos Android. É usada para especificar o conteúdo de arquivos necessários de um projeto android, como do `AndroidManifest` ou dos *layouts* das telas, conforme visto na [subseção 2.1.2](#). Um entendimento profundo dessa linguagem não é necessário para o desenvolvimento de aplicações android, entretanto, é necessário entender seus princípios básicos e sua sintaxe (HANËIN, 2017).

3.2.6 android_core

Vimos na [subseção 2.2.4](#) que o Rosjava possibilitou o uso do ROS em aplicativos Android. A aplicação desenvolvida para esse trabalho foi desenvolvida com base no pacote `android_core` do Rosjava. O `android_core` é uma coleção de componentes e exemplos úteis para o desenvolvimento de aplicativos ROS no Android (ROSJAVA, 2013). Esse conjunto de bibliotecas fornece vários módulos pertinentes ao desenvolvimento de aplicativos Android ROS, como interfaces para visualização de sensores, mapas, teleoperação e outras funcionalidades que um nó ROS pode executar. A configuração do Rosjava e do pacote `android_core` no Android Studio é detalhada no [Apêndice A](#).

3.2.7 Hector ROS

Hector é uma coleção de pacotes ROS originalmente desenvolvida pela Universidade Técnica de Darmstadt, na Alemanha. Ele fornece várias ferramentas para simular ou integrar com robôs (ROS, 2013). Nesse trabalho, foram utilizados os metapacotes `hector_quadrotor` e `hector_slam` para simular um modelo de quadrotor 3D no Gazebo e ter a construção de um mapa 2D do ambiente simulado, com uma ferramenta SLAM.

O `hector_quadrotor` contém pacotes relacionados à modelagem, controle e simulação de quadrotores. A [Tabela 6](#) lista e descreve alguns desses pacotes.

Para a tarefa de construção do mapa do ambiente simulado, o metapacote `hector_slam` fornece as ferramentas necessárias. O `hector_slam` contém pacotes ROS relacionados à execução do SLAM em ambientes não gerenciados, como os encontrados nos cenários de busca e resgate (ROS, 2014b). A [Tabela 7](#) descreve alguns dos pacotes do `hector_slam`.

Tabela 6 – Pacotes do `hector_quadrotor`

Pacote	Descrição
<code>hector_quadrotor_description</code>	Fornecer um modelo URDF ² de quadrotor genérico, bem como variantes com vários sensores;
<code>hector_quadrotor_gazebo</code>	Contém os arquivos de inicialização necessários e as informações de dependência para simulação do modelo de quadrotor no Gazebo;
<code>hector_quadrotor_teleop</code>	Contém um nó que permite controlar o quadrotor usando um <i>gamepad</i> ;
<code>hector_quadrotor_gazebo_plugins</code>	Fornecer plugins que são específicos para a simulação de quadrotores no Gazebo.
<code>hector_quadrotor_demo</code>	Fornecer arquivos <code>.launch</code> e dependências necessárias para demonstração do <code>hector_quadrotor</code> no Gazebo.

Fonte: ROS (2014a)

Tabela 7 – Pacotes do `hector_slam`

Pacote	Descrição
<code>hector_mapping</code>	Nó do SLAM;
<code>hector_geotiff</code>	Fornecer um nó que pode ser usado para salvar mapas de grade de ocupação, trajetória de robô e dados de objeto de interesse para imagens no formato <i>GeoTiff</i> ;
<code>hector_trajectory_server</code>	Controla as trajetórias extraídas dos dados e torna esses dados acessíveis através de um serviço e tópico.

Fonte: ROS (2014b)

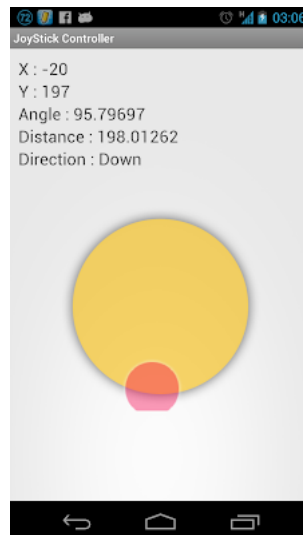
O quadrotor do `hector_quadrotor` aceita mensagens `geometry_msgs/Twist` (visto na [subseção 2.2.2.4](#)) no tópico chamado “*cmd_vel*”, assim como muitos outros robôs. Para o controle do drone na aplicação proposta, serão utilizadas as velocidades lineares X, Y e Z, que deslocam o quadrotor para frente/trás, direita/esquerda e cima/baixo, respectivamente. Será utilizada também a velocidade angular Z, que rotaciona o quadrotor para direita ou para esquerda em torno do seu próprio eixo central.

3.2.8 Android Joystick

O Android possui uma grande comunidade atuante em todo o mundo. Pesquisadores, desenvolvedores e usuários da plataforma trocam, através de fóruns, blogs e redes sociais, informações, dicas e trabalhos desenvolvidos para o Android. Consequentemente, torna-se fácil esclarecer dúvidas, encontrar informações desejadas e contribuições para projetos de aplicações para essa plataforma. Akexorcist (2012), disponibiliza um controle *Joystick* implementado em Java que pode ser usado no desenvolvimento de uma variedade de aplicativos. Através da classe fornecida *JoyStickClass.java*, podemos encontrar todas as funcionalidades esperadas para um controle *Joystick* virtual para o Android. Akexorcist (2012) disponibiliza ainda um layout em XML para o controle virtual, de modo que com apenas algumas edições é possível ter o design

do *Joystick* desejado para sua aplicação. A Figura 19 mostra a execução desse controle num dispositivo Android, onde o círculo amarelo define o escopo do *joystick* e o vermelho a alavanca desse. Todos os detalhes de como utilizar esse *Joystick* nas aplicações podem ser encontrados no [site do desenvolvedor](#) ³.

Figura 19 – Controle Joystick Android



Fonte – Akexorcist (2012)

3.3 Implementação

Esta seção é dedicada à descrição da implementação da aplicação desenvolvida. Foram utilizadas quatro classes durante o desenvolvimento. Sendo elas: *MainActivity.java*, *SystemCommands.java*, *JoyStickClass.java* e *NodeControle.java*. A seguir será feita uma descrição de cada uma delas.

3.3.1 Classe *MainActivity.java*

A classe *MainActivity* é a classe principal do sistema, ela estende a classe *RosActivity*, que por sua vez estende a classe *Activity* do android. Dessa forma, a classe *RosActivity* fornece todas as especificações para criação de uma atividade android, além de outras funcionalidades do ROS, como o *MasterChooser* (DOCUMENTATION, 2013) da tela inicial.

A classe *MainActivity* é responsável também por declarar a interface da aplicação definida no arquivo `activity_main.xml`, iniciar todos os nós da aplicação através do método `init()`, de especificar a visualização da câmera do quadrotor e as funcionalidades do mapa construído pelo SLAM. O objeto *rosImageView*, da classe *RosImageView<T>*, define o nó responsável pela visualização da câmera do robô. Esse nó, sobrescreve o tópico `/front_cam/camera/image/compressed` e recebe do *drone* a imagem do sensor da câmera. Para

³ <http://www.akexorcist.com/2012/10/android-code-joystick-controller.html>

visualização do mapa sendo construído pelo SLAM, é necessário instanciar a classe *VisualizationView*. O objeto dessa classe recebe quatro camadas (*Layer*), necessárias para visualização e manipulação do mapa na aplicação. São elas:

- *cameraControlLayer* - Responsável pela configuração da manipulação feita com mapa, como zooms, rotações e deslocamento da imagem gerada pelo mapa.
- *OccupancyGridLayer* - Necessário para o desenho feito do mapa na grade de visualização, a partir das informações do sensor do laser.
- *LaserScanLayer* - Define o tópico utilizado para obter informações do sensor do laser do robô. Foi utilizado o tópico `/scan`.
- *RobotLayer* - Define o quadro de coordenadas a ser utilizado pelo robô. O quadro de coordenadas especifica a partir de onde será realizado o cálculo das informações obtidas do laser. Nessa aplicação foi utilizado o quadro *base_link*, definido na variável *ROBOT_FRAME*, que realiza esse cálculo das informações a partir da base do *drone*.

Outros métodos complementam as opções disponibilizadas pela aplicação para interação com o mapa do SLAM. São eles:

- *enableFollowMe()* - Esse método ativa a visualização do mapa a partir da base do robô, de modo que o mapa acompanha o deslocamento do robô. O quadrotor é mostrado na forma de uma seta no mapa (Tabela 9). O *enableFollowMe()* é chamado quando o *toggle button* “SEGUIR” está ativado.
- *disableFollowMe()* - Esse método desativa a visualização do mapa a partir da base do robô, de modo que a visualização do mapa não acompanha mais a base do robô. O *disableFollowMe()* é chamado quando o *toggle button* “SEGUIR” está desativado, que acontece automaticamente quando manipulamos diretamente o mapa na tela ou quando pressionamos o botão “SEGUIR”.
- *onClearMapButtonClicked()* - Utilizado pelo botão “LIMPAR MAPA” para limpar o mapa gerado até o momento e reiniciar a construção feita pelo SLAM.
- *onSaveMapButtonClicked()* - Utilizado pelo botão “SALVAR MAPA” para salvar o esboço do mapa construído. O mapa é salvo no formato GeoTiff dentro do diretório `/hector_slam/hector_geotiff/maps`.
- *onMapButtonClicked()* - Utilizado pelo botão “MAPA” para mostrar ou esconder a interface de visualização e dos botões de interação com o mapa. O objetivo desse botão é disponibilizar uma interface mais limpa para o operador do *drone*, de acordo com a preferência do usuário.

Tabela 8 – Variáveis da classe *JoyStickClass* definidas para as posições da alavanca do *joystick*

Variável	Especificação
STICK_UP	Posição da alavanca para cima;
STICK_DOWN	Posição da alavanca para baixo;
STICK_LEFT	Posição da alavanca para esquerda;
STICK_RIGHT	Posição da alavanca para direita;
STICK_UPRIGHT	Posição da alavanca para diagonal superior direita;
STICK_UPLEFT	Posição da alavanca para diagonal superior esquerda;
STICK_DOWNRIGHT	Posição da alavanca para diagonal inferior direita;
STICK_DOWNLEFT	Posição da alavanca para diagonal inferior esquerda;
STICK_NONE	Posição fora do escopo da alavanca.

Fonte: Akexorcist (2012)

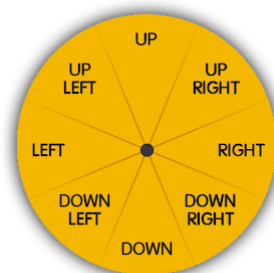
3.3.2 Classe *SystemCommands.java*

A classe *SystemCommands* é fornecida pelo pacote `android_core`, e não foi alterada. Ela define o nó responsável por executar as funções de limpar e salvar o mapa. O método `reset()` dessa classe, é chamado no `onClearMapButtonClicked()` da classe *MainActivity*, através do objeto `systemCommands`, para fazer o `reset` do mapa. O método `saveGeotiff()` é chamado dentro do `onSaveMapButtonClicked()`, também através do objeto `systemCommands`, para executar a gravação do mapa construído pelo SLAM. Esses dois métodos, `reset()` e `saveGeotiff()`, definem a mensagem publicada no tópico `/syscommand`, que recebe mensagens do tipo `std_msgs/String`. Sendo a string “`reset`” definindo a mensagem utilizada para limpar o mapa e a string “`savegeotiff`” a mensagem para gravar o mapa.

3.3.3 Classe *JoyStickClass.java*

Conforme já apresentado na [subseção 3.2.8](#), na classe *JoyStickClass* podemos encontrar todas as funcionalidades esperadas para um controle *Joystick* virtual para o Android. Nessa classe estão especificadas funções para configurarmos o design do *joystick* e as ações executadas por ele. Existem nove variáveis de classe que retornam a posição atual da alavanca do *joystick* (ver [Tabela 8](#)). Dessa forma, podemos configurar ações específicas a partir da interação do usuário com esse controle virtual. A [Figura 20](#) mostra as posições possíveis retornadas pelo *joystick*. Podemos encontrar mais informações sobre essa classe e o código fonte completo no [site do desenvolvedor da ferramenta](#)⁴.

⁴ <http://www.akexorcist.com/2012/10/android-code-joystick-controller.html>

Figura 20 – Posições possíveis retornadas do *joystick*

Fonte – Akexorcist (2012)

3.3.4 Classe *NodeControle.java*

A classe *NodeControle* define as configurações para a operação do quadrotor. Nessa classe, estão definidas as especificações do controle do *drone* através de dois *joysticks* virtuais e do acelerômetro. Tanto os dois *joysticks* como o acelerômetro, publicam mensagens *geometry_msgs/Twist* no tópico *cmd_vel* para movimentar o quadrotor, com valor de 1m/s no caso das velocidades lineares e 1 rad/s para velocidade angular. O *joystick* da esquerda, publica, de acordo com a posição da alavanca do *joystick*, mensagens para velocidade linear Z e angular Z através dos métodos *getLinear().setZ()* e *getAngular().setZ()*, respectivamente. O *joystick* da direita, publica, também de acordo com a posição da alavanca do *joystick*, mensagens para velocidades lineares X e Y através dos métodos *getLinear().setX()* e *getLinear().setY()*, respectivamente. Esses métodos para publicação das velocidades lineares e angulares do robô, também são usados para o controle através do acelerômetro. Através da *interface SensorEventListener*, implementamos o método *onSensorChanged()* e trabalhamos com informações ocasionadas por um evento em um sensor do *smartphone*. Dessa forma, é possível estabelecer ações na aplicação conforme um valor retornado pelo sensor do acelerômetro, que no caso foi a publicação das velocidades angular Z e linear X no tópico *cmd_vel*.

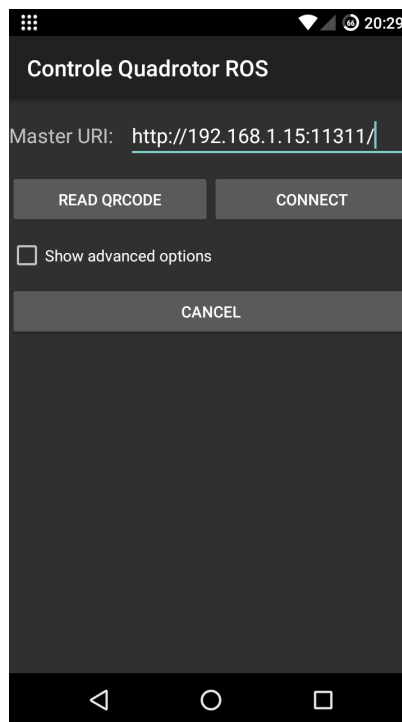
Mais detalhes sobre as classes *MainActivity*, *NodeControle* e *SystemCommands* podem ser vistos no [Apêndice B](#) desse trabalho.

4 RESULTADOS

A aplicação desenvolvida é composta de duas telas que definem a interface do usuário. A tela inicial, mostrada na [Figura 21](#), é fornecida pelo Rosjava e não foi alterada. Ela disponibiliza as configurações para conexão com o ROS *Master*, isso é conhecido como “*Master Chooser*”. Para permitir a comunicação entre nós em diferentes computadores, a variável de ambiente ROS_MASTER_URI em cada PC cliente deve ser configurada para o endereço IP do PC onde o nó mestre é lançado, ou seja, o PC mestre ([JOSEPH, 2017](#)). Assim, nessa tela inicial devemos colocar o endereço URI do PC mestre.

A segunda tela é a que contém todos os elementos para a interação do usuário com o quadrotor. A [Figura 22](#) mostra a interface dessa tela e a [Tabela 9](#) descreve seus componentes.

Figura 21 – Interface tela inicial - *Master Chooser*



Fonte – Captura de Tela App “*Teleoperação Quadrotor ROS*” - Master Chooser Rosjava

Figura 22 – Interface do operador



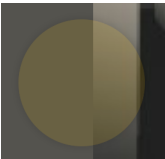




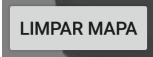



(a) Interface com botões desativados



(b) Interface com botões ativados

Fonte – Captura de Tela App “Teleoperação Quadrotor ROS”

Tabela 9 – Componentes da interface do usuário

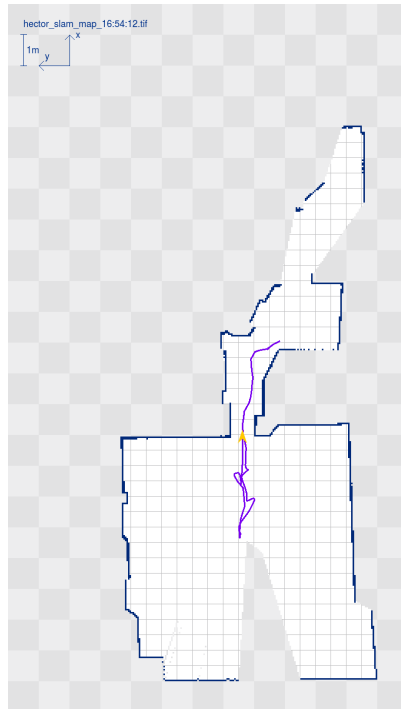
Componente	Funcionalidade
	Joystick esquerdo responsável por setar a velocidade linear e angular Z do drone, movendo assim o robô para cima ou para baixo e girando-o para esquerda ou direita;
	Joystick direito responsável por setar as velocidades lineares X e Y do drone, deslocando o quadrotor para frente, para trás, para direita ou esquerda;
	<i>Toggle button</i> ¹ que ativa e desativa o controle das velocidades angular Z e linear X pelo acelerômetro do <i>smartphone</i> . Quando essa função é ativada, as velocidades angular Z e linear X dos <i>joysticks</i> são desabilitadas, passando a responsabilidade da publicação dessas velocidades apenas para o acelerômetro;
	<i>Toggle button</i> que ativa e desativa as funcionalidades do mapa do ambiente, construído pelo SLAM;
	Botão que salva o mapa do ambiente, construído pelo SLAM, no diretório <code>/hector_geotiff/maps</code> do <code>hector_slam</code> ;
	Botão que permite limpar o mapa do ambiente, construído pelo SLAM, reiniciando o processo de construção;
	<i>Toggle button</i> que ativa e desativa o modo seguir no mapa. Se o modo seguir não for desativado, o mapa acompanhará o quadrotor na sua posição atual, caso contrário o mapa ficará em modo estático e poderá ser manipulado;
	Transmissão do mapa do ambiente de navegação, durante seu processo de construção pelo SLAM;
	Transmissão da imagem gerada pela câmera frontal do quadrotor.

A [Figura 23](#) mostra o exemplo de um mapa construído pelo SLAM, durante a execução da aplicação desenvolvida nesse trabalho. O mapa foi salvo, através do botão “SALVAR” da aplicação, no diretório `/hector_slam/hector_geotiff/maps` localizado no com-

¹ Um botão de alternância que permite que o usuário altere uma configuração entre dois estados (ligado/desligado).

putador onde está sendo executado o ROS e o Gazebo. O mapa construído pelo SLAM na aplicação é do tipo grade de ocupação. A ideia básica da grade de ocupação é representar um mapa do ambiente como um conjunto de células, cada uma representando uma área ocupada ou não no ambiente. A parte branca do mapa representa a área mapeada, sendo as linhas azuis a parte ocupada no ambiente. O espaço cinza quadriculado, representa a área não mapeada e a linha roxa mostra o trajeto feito pelo quadrotor durante a navegação.

Figura 23 – Exemplo de mapa gerado pelo SLAM utilizando a aplicação desenvolvida



Fonte – App Controle Quadrotor ROS - Mapa SLAM

4.1 Executando o ROS/Gazebo no computador

Para que se estabeleça a conexão entre a aplicação executada no Android e o robô simulado, devemos inicialmente configurar as variáveis de ambiente `ROS_IP` e `ROS_MASTER_URI`. A variável `ROS_IP` deve ser configurada com o IP da máquina onde estão sendo executadas os nós que desejamos nos conectar. A variável `ROS_MASTER_URI` deve ser configurada com o endereço URI de onde será executado o ROS *Master*. Ambas configurações dessas variáveis podem ser feitas no terminal do Linux com os comandos dados a seguir, sendo o IP 192.168.1.15 definido apenas como exemplo.

```
$ export ROS_IP=192.168.1.15
$ export ROS_MASTER_URI=http://192.168.1.15:11311/
```

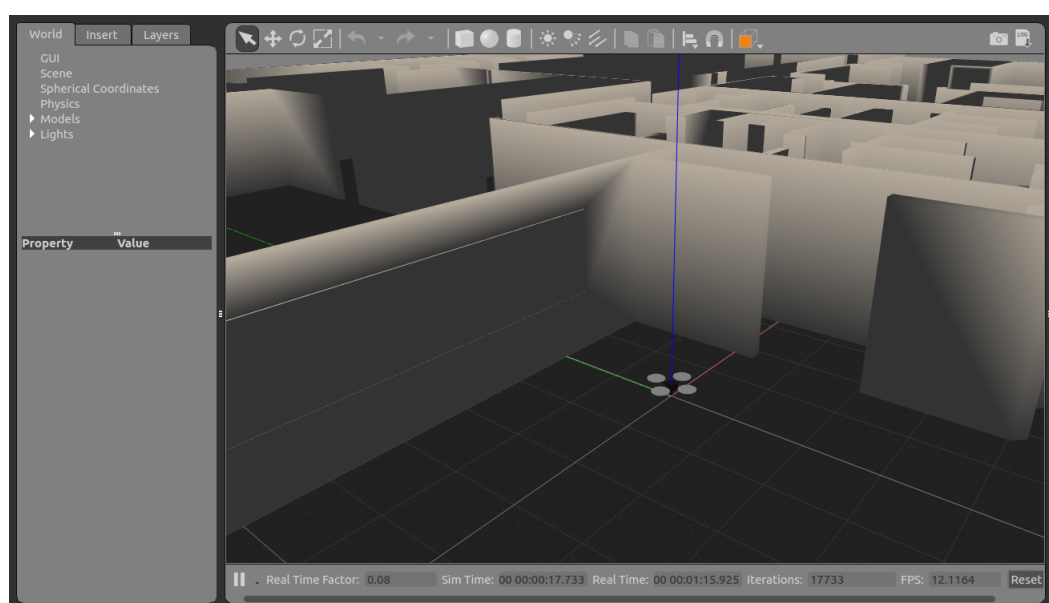
Com as variáveis de ambiente definidas, podemos iniciar o ROS e o simulador Gazebo. Essas duas inicializações podem ser feitas em conjunto, através do comando `roslaunch`

que executa um arquivo `.launch` disponibilizado pelo pacote `hector_quadrotor_demo`. Nesse arquivo estão as configurações do ambiente de simulação do Gazebo, do quadrotor, do SLAM e de todos os nós necessários para a simulação do robô. O ambiente virtual ou o “mundo” do Gazebo utilizado nessa aplicação é denominado *willow_garage*. Nesse ambiente é simulado um voo do *Hector Quadrotor* em um local interno (*indoor*), como uma casa. Executamos o arquivo `.launch` do `hector_quadrotor_demo` para esse ambiente através do comando:

```
$ roslaunch hector_quadrotor_demo indoor_slam_gazebo.launch
```

A [Figura 24](#) mostra o ambiente de simulação do quadrotor executado pelo Gazebo.

Figura 24 – Ambiente *indoor* executado no Gazebo



Fonte – Captura de tela - *Gazebo em Execução*

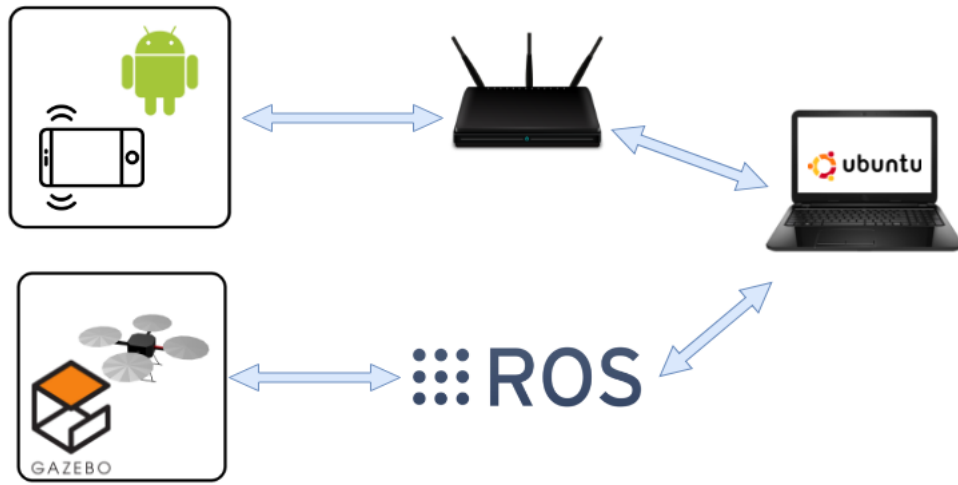
Na versão *Kinetic* do ROS, é necessário ainda ativar os motores do robô para iniciar o voo. Essa ativação pode ser feita chamando um serviço ROS através do comando:

```
$ rosservice call \enable_motors true
```

Com o ROS e o Gazebo sendo executados, podemos fazer a conexão da aplicação Android com o quadrotor inserindo na tela inicial do aplicativo o endereço do ROS *Master* já configurado na variável de ambiente `ROS_MASTER_URI`. Com a conexão estabelecida entre o *smartphone* e o quadrotor simulado, os nós executados pela aplicação e pelo quadrotor podem então se comunicar através dos tópicos específicos. A relação entre nós a partir dos tópicos, pode ser visualizada no grafo do ROS através da ferramenta de linha de comando `rqt_graph`, vista na [subseção 2.2.2.8](#). A [Figura 25](#) mostra o grafo do ROS durante a execução da aplicação desenvolvida nesse trabalho.

A [Figura 26](#) mostra um diagrama que ilustra o sistema final obtido a partir do uso da aplicação desenvolvida. O Gazebo transmite informações ao ROS que, sendo executado em um

Figura 26 – Diagrama do sistema final obtido



Fonte – próprio autor

5 CONCLUSÃO

Este trabalho apresentou o desenvolvimento de uma aplicação Android para um quadrotor utilizando o *framework* ROS. O aplicativo tem o objetivo de fornecer a comunidade ROS uma ferramenta para teleoperação de *drones*, priorizando um controle natural e intuitivo, e que apoie o operador com informações do ambiente remoto, através da transmissão do sensor da câmera do robô e da visualização do mapa do espaço de navegação gerado a partir da técnica SLAM. O aplicativo também pode ser usado por um VANT para mapeamento de ambientes desconhecidos, já que é possível salvar em formato de imagem o mapa construído pelo SLAM, em um diretório de arquivo local.

A primeira parte do trabalho apresentou os estudos feitos sobre o tema robótica e suas aplicações mais frequentes atualmente, principalmente na área da robótica móvel. A plataforma Android foi apresentada a fim de trazer os principais conceitos por traz dos aplicativos desse sistema. Assim, foi possível compreender o funcionamento desses aplicativos no sistema operacional Android e os elementos básicos de um projeto de desenvolvimento dessas aplicações. Foi possível ainda, compreender o potencial do *framework* ROS para a área da robótica a partir da apresentação dos conceitos por traz dessa plataforma e de exemplos do uso desse sistema no campo da robótica. O estudo da biblioteca Rosjava demonstrou como integrar o ROS junto ao Android, mostrando as possibilidades que podem ser criadas com essa união e algumas aplicações Android já criadas para o ROS.

As ferramentas utilizadas no desenvolvimento da aplicação, apresentadas no [Capítulo 3](#), se mostraram adequadas para elaboração e teste do aplicativo proposto. A IDE Android Studio proporcionou um ambiente de desenvolvimento completo e facilitou o esboço tanto da interface do usuário quanto da lógica de programação por traz da aplicação. Utilizar o Gazebo, ajudou a testar a aplicação num ambiente virtual fiel aos ambientes reais, assegurando a integridade física do usuário e diminuindo os custos do projeto, já que não foi necessário a utilização de um quadrotor real nos testes do aplicativo construído. No [Capítulo 4](#), foi detalhado as classes e os principais métodos presentes no código fonte da aplicação. Isso possibilitou um melhor entendimento do funcionamento do aplicativo desenvolvido e proporciona que outros desenvolvedores possam trabalhar no aprimoramento da aplicação com maior facilidade.

Como conclusões, podemos dizer que objetivo principal foi cumprido, sendo possível criar a ferramenta proposta utilizando as tecnologias e conceitos apresentados no projeto. O estudo da plataforma Android e do *framework* ROS possibilitou criar uma base de estudos para o desenvolvimento de aplicações para dispositivos móveis voltadas para robótica. A apresentação da técnica SLAM, viabilizou demonstrar uma solução para navegação de robôs em espaços imprevisíveis e o mapeamento desses tipos de ambientes.

5.1 Trabalhos futuros

A partir da aplicação e da base de estudos desenvolvida nesse trabalho, cria-se uma referência para a criação de aplicações para robôs utilizando o ROS. Outros desenvolvedores podem, a partir desse trabalho, aprimorar a aplicação construída ou desenvolver novas aplicações.

Como proposta para outros projetos, sugere-se a sofisticação do uso dos *joysticks* e de outros sensores dos dispositivos com sistema operacional Android. Na aplicação desenvolvida para esse trabalho, as mensagens para as velocidades lineares e angulares são sempre para o valor 1. Logo, a velocidade no quadrotor permanece constante em 1 m/s ou 1 rad/s, independentemente da força aplicada nos *joysticks* ou da aceleração medida pelo sensor do acelerômetro. O uso de mensagens publicadas que variam de acordo com a força aplicada nos controles, pode proporcionar um controle mais preciso do quadrotor. A utilização de outros sensores, como o Giroscópio, também pode proporcionar o uso de dados mais completos sobre a posição do *smartphone*, podendo essas informações serem aplicadas para publicar mensagens de velocidade em outras direções, diferentes das já utilizadas pelo sensor do acelerômetro na aplicação construída nesse trabalho.

A utilização de outros dados na aplicação também pode ajudar no apoio ao operador do *drone*. O uso de um sensor que retorne a altitude de uma aeronave pode, por exemplo, ajudar o operador a pousar o veículo quando as condições de visibilidade através do sensor da câmera não forem boas. O mapa gerado pelo SLAM e disponibilizado na aplicação desse trabalho é de duas dimensões (2D), com uma vista superior do ambiente. Esse tipo de mapa pode funcionar bem em locais cuja as dimensões das estruturas são normalmente uniformes, como casas e prédios. Porém, para trabalhos em áreas onde existe um relevo irregular, a utilização de um mapa 3D, pode retornar melhores informações do ambiente. Dessa forma, disponibilizar na aplicação a opção para esse tipo de mapa, pode apoiar melhor a operação de um robô em outros tipos de ambiente e fornecer um mapeamento mais fiel ao espaço de navegação.

Outra sugestão, seria a criação de um menu para configuração de publicação e subscrição de tópicos, que pode facilitar o uso dos recursos do ROS. Através desse menu, poderia-se criar configurações personalizadas de acordo com a preferência do usuário, que pode ativar ou desativar um recurso quando desejar, sem a necessidade de se fazer a alteração no código fonte da aplicação.

REFERÊNCIAS

- AKEXORCIST. [Android Code] **JoyStick Controller**. 2012. Disponível em: <http://www.akexorcist.com/2012/10/android-code-joystick-controller.html>. Acesso em: 24.06.2018.
- ALMEIDA, T. **Drones e robôs tornam mineração mais produtiva**. 2016. Disponível em: <https://www.industriahoje.com.br/drones-e-robos-tornam-mineracao-mais-produtiva/>. Acesso em: 16.07.2018.
- ANDROID, D. **Conheça o Android Studio**. 2018. Disponível em: <https://developer.android.com/studio/intro/>. Acesso em: 23.06.2018.
- ANDROID, D. **Manifest.permission - WAKE_LOCK**. 2018. Disponível em: https://developer.android.com/reference/android/Manifest.permission#WAKE_LOCK. Acesso em: 04.06.2018.
- ANDROID, D. **Verificação do comportamento do aplicativo no Android Runtime (ART)**. 2018. Disponível em: <https://developer.android.com/guide/practices/verifying-apps-art?hl=pt-br>. Acesso em: 06.06.2018.
- BARBOSA, G. Y. **Os drones e a produção cinematográfica**. 2016. Disponível em: <http://www.droneshowla.com/artigo-os-drones-e-producao-cinematografica-2/>. Acesso em: 16.07.2018.
- BECKER, M.; SAMPAIO, R. C. B.; BOUABDALLAH, S.; PERROT, V. de; SIEGWART, R. **In flight collision avoidance for a MiniUAV robot based on onboard sensors**. 2014. Disponível em: https://www.researchgate.net/publication/262687466_In_flight_collision_avoidance_for_a_Mini-UAV_robot_based_on_onboard_sensors. Acesso em: 23.06.2018.
- CARLSON, C.; CHEN, T.; CRUZ, J.; MAGHSOUDI, J.; ZHAO, H.; MONACO, J. V. **User Authentication with Android Accelerometer and Gyroscope Sensors**. 2015. Disponível em: <https://pdfs.semanticscholar.org/5b4c/ef41b056013c4ea20356a896ad9834ea3f2e.pdf>. Acesso em: 17.06.2018.
- CONLEY, K. **Nodes**. 2012. Disponível em: <http://wiki.ros.org/Nodes>. Acesso em: 04.06.2018.
- CONLEY, K. **Services**. 2012. Disponível em: <http://wiki.ros.org/Services>.
- COTA, E.; TORRE, M. P.; FERREIRA, J. A. T.; FIDÊNCIO, A. X.; RODRIGUES, G. B.; ROCHA, F. A. S.; AZPÚRUA, H.; FREITAS, G. M.; MIOLA, W. **ROBÓTICA NA MINERAÇÃO**. 2017. Disponível em: https://www.researchgate.net/publication/320725502_ROBOTICA_NA_MINERACAO. Acesso em: 16.07.2018.
- DEITEL, P.; DEITEL, H. **Java: how to program**. [S.l.]: Deitel, 2012.
- DEVELOPER, A. **Arquitetura da plataforma**. 2018. Disponível em: <https://developer.android.com/guide/platform/?hl=pt-br>.
- DEVELOPER, A. **Atividades**. 2018. Disponível em: <https://developer.android.com/guide/components/activities?hl=pt-br>. Acesso em: 17.07.2018.

- DEVELOPER, A. **Configure sua compilação**. 2018. Disponível em: <https://developer.android.com/studio/build/>. Acesso em: 16.06.2018.
- DEVELOPER, A. **Introduction to Activities**. 2018. Disponível em: <https://developer.android.com/guide/components/activities/intro-activities>. Acesso em: 17.06.2018.
- DEVELOPER, A. **Manifesto do aplicativo**. 2018. Disponível em: <https://developer.android.com/guide/topics/manifest/manifest-intro>.
- DEVELOPER, A. **Sensors Overview**. 2018. Disponível em: https://developer.android.com/guide/topics/sensors/sensors_overview. Acesso em: 17.06.2018.
- DIFFOUO, R. **Introduction to ROS (Robot Operating System)**. 2013.
- DOCUMENTATION. 2018. Disponível em: <http://wiki.ros.org/>. Acesso em: 27.05.2018.
- DOCUMENTATION android_core. **Using RosActivity**. 2013. Disponível em: http://rosjava.github.io/android_core/latest/getting_started.html. Acesso em: 04.07.2018.
- EDNO, F. **Understanding ROS Topics**. 2014. Disponível em: http://wiki.ros.org/pt_BR/ROS/Tutorials/UnderstandingTopics#Using_rqt_graph.
- ENGINEERGUY. **How a Smartphone Knows Up from Down (accelerometer)**. 2012. Disponível em: <https://www.youtube.com/watch?v=KZVgKu6v808&t=200s>. Acesso em: 17.06.2018.
- FERREIRA, D. M. da S. **Robô Rececionista**. Dissertação (Mestrado) — Instituto Politécnico de Tomar, 2015. Disponível em: <https://comum.rcaap.pt/handle/10400.26/12646>.
- FIGUEIREDO, C. E. G. **DESENVOLVIMENTO DE UMA APLICAÇÃO ANDROID PARA TELEOPERAÇÃO DE UM DRONE QUADRIRROTOR UTILIZANDO O ROS**. 2016.
- FOOTE, T.; PURVIS, M. **Standard Units of Measure and Coordinate Conventions**. 2014. Disponível em: <http://www.ros.org/repos/rep-0103.html>. Acesso em: 18.07.2018.
- FOROUHER, D. **Topics**. 2014. Disponível em: <http://wiki.ros.org/Topics>.
- GAZEBO. **Tutorial: ROS integration overview**. 2014. Disponível em: http://gazebosim.org/tutorials/?tut=ros_overview. Acesso em: 23.06.2018.
- GAZEBO. **Why Gazebo?** 2018. Disponível em: <http://gazebosim.org/>. Acesso em: 23.06.2018.
- GLOVER, C.; RUSSELL, B.; WHITE, A.; MILLER, M.; STOYTCHEV, A. **An Effective and Intuitive Control Interface for Remote Robot Teleoperation with Complete Haptic Feedback**. 2009. Disponível em: http://www.ece.iastate.edu/~alexs/lab/publications/papers/ETC_2009/ETC_2009.pdf. Acesso em: 22.06.2018.
- HANČIN, J. **A Mobile Android Application for Music Recommendation**. 2017. Disponível em: https://is.muni.cz/th/eakqe/Jakub_Hancin_-_Bakalarka_-_DIGITAL.pdf. Acesso em: 16.06.2018.
- HORTOVANYI, N. **Kinematics Pick & Place Project**. 2017. Disponível em: <https://medium.com/@NickHortovanyi/robotic-arm-pick-place-d93c71368b64>. Acesso em: 23.06.2018.

IRVS. **how to configure rosjava apps with gradle**. 2016. Disponível em: https://github.com/irvs/ros_tms/wiki/how-to-configure-rosjava-apps-with-gradle.

JOSEPH, L. **Mastering ROS for Robotics Programming**. Birmingham: Packt Publishing, 2015.

JOSEPH, L. **ROS Robotics Projects**. Birmingham: Packt Publishing, 2017.

KOUBAA, A. (Ed.). **Robot Operating System (ROS) - The Complete Reference (Volume 1)**. [S.l.]: Springer, 2016.

KRAJCI, I.; CUMMINGS, D. **Android on x86: An Introduction to Optimizing for Intel Architecture**. [S.l.]: Apress, 2013.

LAGES, W. F. **Robot Operating System (ROS) - Introdução e Implementação de Controladores**. 2017. Disponível em: <http://www.ece.ufrgs.br/~fetter/sbai2017-ros/intro.pdf>. Acesso em: 30.05.2018.

MARTINEZ, A.; FERNÁNDEZ, E. **Learning ROS for Robotics Programming**. [S.l.]: Packt Publishing, 2013.

MATHWORKS. **Access the ROS Parameter Server**. 2018. Disponível em: <https://www.mathworks.com/help/robotics/examples/access-the-ros-parameter-server.html>. Acesso em: 05.06.2018.

MAZZARI, V. **ROS – Robot Operating System**. 2016. Disponível em: <https://www.generationrobots.com/blog/en/2016/03/ros-robot-operating-system-2/>. Acesso em: 27.05.2018.

NOOTRIX. **ROS Naming and Namespaces**. 2018. Disponível em: <https://nootrix.com/diy-tutos/ros-namespaces/>. Acesso em: 25.06.2018.

NYRO. **How to learn robotics with ROS**. 2017. Disponível em: <https://niry.com/2017/03/09/learn-robotics-ros/>. Acesso em: 27.05.2018.

O'HORA, K. **Lights, Camera, Quadrotors!** 2017. Disponível em: <http://amt-lab.org/blog/2017/5/lights-camera-quadrotors>. Acesso em: 22.06.2018.

O'KANE, J. M. **A Gentle Introduction to ROS**. 2014. Disponível em: <https://www.cse.sc.edu/~jokane/agitr/agitr-letter.pdf>.

POUDEL, A. **MOBILE APPLICATION DEVELOPMENT FOR ANDROID OPERATING SYSTEM**. 2013. Disponível em: https://www.theseus.fi/bitstream/handle/10024/64719/Poudel_Amrit.pdf. Acesso em: 17.06.2018.

PYO, Y.; CHO, H.; JUNG, R.; LIM, T. **ROS Robot Programming**. [S.l.]: ROBOTIS Co.,Ltd., 2017.

RIISGAARD, S.; BLAS, M. R. **SLAM for Dummies: A Tutorial Approach to Simultaneous Localization and Mapping**. 2005. Disponível em: https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-412j-cognitive-robotics-spring-2005/projects/1aslambblas_repo.pdf. Acesso em: 19.06.2018.

ROBOTICS, C. **HUSKY**. 2018. Disponível em: <https://www.clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/>. Acesso em: 23.06.2018.

- ROBOTICS, I. **Robotics Simulation Softwares With 3D Modeling and Programming Support**. 2014. Disponível em: <https://www.intorobotics.com/robotics-simulation-softwares-with-3d-modeling-and-programming-support/>. Acesso em: 23.06.2018.
- ROMERO, A. M. **Concepts**. 2014. Disponível em: <http://wiki.ros.org/ROS/Concepts>.
- ROS, W. **tu-darmstadt-ros-pkg**. 2013. Disponível em: <http://wiki.ros.org/tu-darmstadt-ros-pkg>. Acesso em: 24.06.2018.
- ROS, W. **hector_quadrotor**. 2014. Disponível em: http://wiki.ros.org/hector_quadrotor. Acesso em: 24.06.2018.
- ROS, W. **hector_slam**. 2014. Disponível em: http://wiki.ros.org/hector_slam. Acesso em: 24.06.2018.
- ROS, W. **EnvironmentVariables ROS**. 2016. Disponível em: <http://wiki.ros.org/ROS/EnvironmentVariables>. Acesso em: 24.06.2018.
- ROS, W. **geometry_msgs**. 2018. Disponível em: http://wiki.ros.org/geometry_msgs. Acesso em: 28.05.2018.
- ROJAVA, D. **android_core**. 2013. Disponível em: http://rosjava.github.io/android_core/latest/. Acesso em: 24.06.2018.
- SAITO, I. **roscpp**. 2013. Disponível em: <http://wiki.ros.org/roscpp>. Acesso em: 05.06.2018.
- SAITO, I. **rosmaster**. 2017. Disponível em: <http://wiki.ros.org/rosmaster>.
- SAITO, I. **srv**. 2017. Disponível em: <http://wiki.ros.org/srv>.
- SANTOS, A. G. N. C. dos. **Autonomous Mobile Robot Navigation using Smartphones**. Dissertação (Mestrado) — Universidade Técnica de Lisboa - Instituto Superior Técnico, 2008. Disponível em: <https://fenix.tecnico.ulisboa.pt/downloadFile/395137855053/dissertacao.pdf>. Acesso em: 22.06.2018.
- SANTOS, N. M. B. dos. **Atualização de Demonstrador Robótico para Utilização do "ROS"**. 2013. Disponível em: <https://repositorio-aberto.up.pt/bitstream/10216/67932/2/26737.pdf>. Acesso em: 02.06.2018.
- SILVA, C. R. de Sousa e; YEPES, I. **DESENVOLVIMENTO DE SISTEMA SLAM COM ODOMETRIA VISUAL PARA VANT DE INSPEÇÃO EM AMBIENTES INTERNOS**. 2016. Disponível em: <https://revista.unitins.br/index.php/humanidadeseinovacao/article/view/166>. Acesso em: 16.07.2018.
- SPRICKERHOF, J. **roscpp**. 2015. Disponível em: <http://wiki.ros.org/roscpp>.
- STRATOM. **rosjava, Android Studio and Windows**. 2016. Disponível em: http://www.stratom.com/blog/2016/02/23/rosjava-android-studio-and-windows/#codesyntax_2.
- THOMAS, D. **msg**. 2017. Disponível em: <http://wiki.ros.org/msg>.
- UDPROS. 2013. Disponível em: <http://wiki.ros.org/ROS/UDPROS>.

W3C. **Accelerometer**. 2018. Disponível em: <https://www.w3.org/TR/accelerometer/>. Acesso em: 17.06.2018.

WINKLER, C. **How Many Sensors are in a Drone, And What do they Do?** 2016. Disponível em: <https://www.sensorsmag.com/components/how-many-sensors-are-a-drone-and-what-do-they-do>. Acesso em: 23.06.2018.

WOODALL, W. **Distributions**. 2018. Disponível em: <http://wiki.ros.org/Distributions>.

WU, Y. **Master**. 2018. Disponível em: <http://wiki.ros.org/Master>.

Apêndice A – Configuração do ambiente de desenvolvimento Rosjava/Android Studio

Este apêndice apresenta como realizar a configuração da biblioteca Rosjava na IDE Android Studio. Será abordado de forma sintetizada o processo de configuração do ambiente, bem como os erros e soluções encontradas durante esses procedimentos. Esse apêndice tem por objetivo fornecer uma referência para a configuração desse ambiente de desenvolvimento, já que durante o desenvolvimento desse trabalho foram encontradas dificuldades na realização dessa tarefa a partir das documentações disponibilizadas.

A.1 Gradle

Para usar o Rosjava em um projeto no Android Studio, é preciso informar à IDE onde encontrar os pacotes. Isso é feito através do arquivo *build.gradle (Module: app)* do aplicativo. Nesse trabalho foram utilizadas dependências remotas. Desta forma, durante a compilação do projeto, é buscado remotamente as dependências pré-configuradas no arquivo *build.gradle* da aplicação.

Para configurar as dependências do aplicativo, é necessário primeiramente adicionar o repositório *maven* do Rosjava à lista de repositórios desejado para o projeto. Para isso, o [Código A.1](#) deve vir abaixo da seção *android{}*, no arquivo *build.gradle (Module: app)* (IRVS, 2016).

Código A.1 – Repositório Rosjava

```

1 repositories {
2     maven {
3         url 'https://github.com/rosjava/rosjava_mvn_repo/raw/master'
4     }
5     mavenCentral()
6 }

```

Com o repositório incluído ao arquivo *build.gradle* da aplicação, é possível especificar exatamente qual pacote dentro do repositório *maven* do Rosjava que pretendemos compilar. Isso é feito acrescentando na seção *dependencies{}* os pacotes esperados. O [Código A.2](#) exhibe os pacotes do repositório *maven* utilizados neste trabalho, que foram declarados na seção *dependencies{}* do arquivo *build.gradle (Module: app)*.

Código A.2 – Pacotes Rosjava utilizados

```

1 dependencies {
2     ...
3
4     compile('org.ros.android_core:android_15:[0.3, 0.4)') {
5         exclude group: 'junit'
6         exclude group: 'xml-apis'
7     }
8     ...
9 }

```

Por fim, é preciso editar o arquivo *META-INF* que contém informações do Android. Isso é necessário para evitar erros durante o empacotamento do APK, pois o pacote *android_core* contém arquivos com nomes duplicados que poderão gerar conflitos com arquivos criados durante a compilação do projeto (STRATOM, 2016). O Código A.3 exclui esses arquivos com nomes duplicados, que serão substituídos pelos pertencentes ao pacote *android_core*, e deve ser declarado no arquivo *build.gradle* (*Module: app*).

Código A.3 – Arquivos excluídos no Android

```

1 android.packagingOptions {
2     exclude 'META-INF/LICENSE.txt'
3     exclude 'META-INF/NOTICE.txt'
4 }

```

Como os arquivos do Gradle foram alterados, é necessário uma sincronização do projeto para a IDE funcionar corretamente. Isso pode ser feito clicando no comando *SyncNow* localizado na aba superior do arquivo.

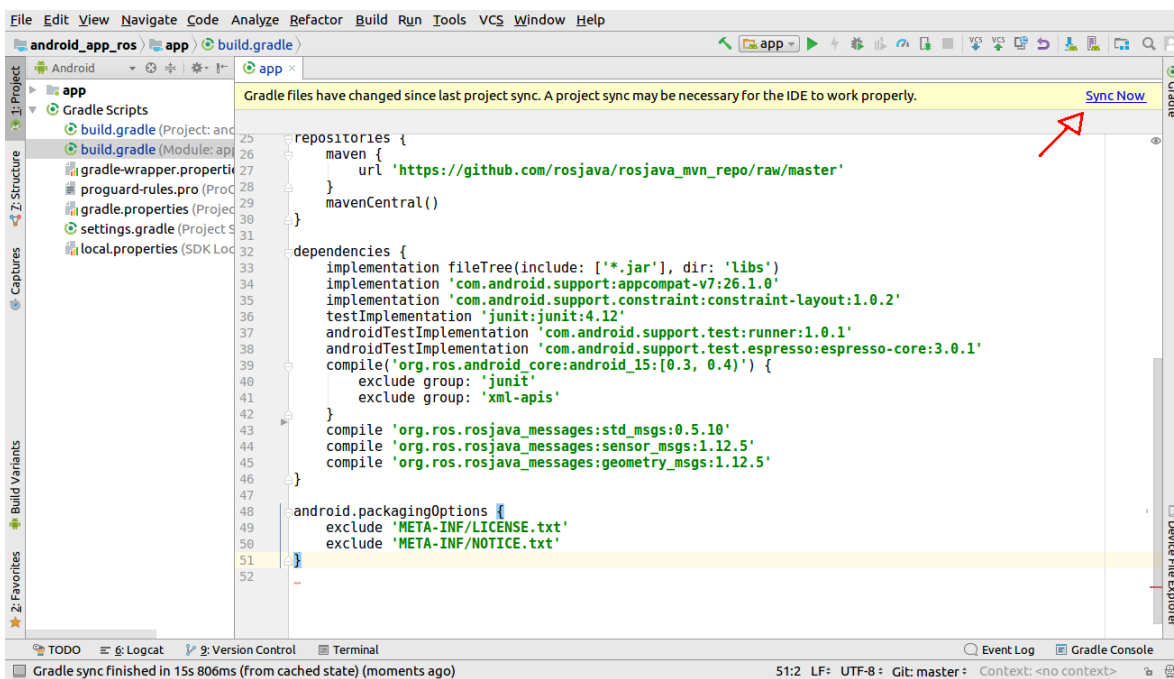


Figura 27 – SyncNow Android Studio

A.2 AndroidManifest

O pacote *android.core* contém seu próprio ícone de aplicativo (STRATOM, 2016), dessa forma o Android Studio acusará um erro de conflito após a sincronização do projeto. Para corrigir esse erro, é necessário editar o arquivo *AndroidManifest.xml*, adicionando as linhas 3 e 6 do Código A.4, para que então, a ferramenta *Rebuild Project* do Android Studio possa ser executada sem nenhum erro de conflito.

Código A.4 – Substituir ícone Android

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3     xmlns:tools="http://schemas.android.com/tools"
4     package="com.caiotf.android_app_ros">
5     <application
6         tools:replace="android:icon"
7         ...
```

Como já mencionado na subseção 2.1.2, é no arquivo *AndroidManifest.xml* que configuramos as permissões do aplicativo. Neste trabalho foram utilizadas as permissões *INTERNET* e *WAKE_LOCK*.

Para dar ao aplicativo a permissão de acesso a internet, devemos incluir dentro da seção *manifest* do arquivo *AndroidManifest.xml*, a linha 6 do Código A.5.

A permissão *WAKE_LOCK* permite o aplicativo utilizar um recurso do Android que impede que o processador entre em repouso ou que a tela do dispositivo diminua o brilho durante sua execução (ANDROID, 2018b). Para dar à aplicação essa permissão, adicionamos na seção *manifest* a linha 7 do Código A.5.

Código A.5 – Permissões do app

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3     xmlns:tools="http://schemas.android.com/tools"
4     package="com.caiotf.android_app_ros">
5
6     <uses-permission android:name="android.permission.INTERNET" />
7     <uses-permission android:name="android.permission.WAKE_LOCK" />
8
9     <application
10         tools:replace="android:icon"
11         ...
```


Apêndice B – Código fonte

B.1 Classe *MainActivity.java*

```

1 package com.caiotf.android_app_ros;
2
3 import android.os.Build;
4 import android.os.Bundle;
5 import android.support.constraint.ConstraintLayout;
6 import android.view.View;
7 import android.widget.LinearLayout;
8 import android.widget.Toast;
9 import android.widget.ToggleButton;
10
11 import com.google.common.base.Preconditions;
12 import com.google.common.collect.Lists;
13
14 import org.ros.address.InetAddressFactory;
15 import org.ros.android.BitmapFromCompressedImage;
16 import org.ros.android.RosActivity;
17 import org.ros.android.view.RosImageView;
18 import org.ros.android.view.visualization.VisualizationView;
19 import org.ros.android.view.visualization.layer.CameraControlLayer;
20 import org.ros.android.view.visualization.layer.CameraControlListener;
21 import org.ros.android.view.visualization.layer.LaserScanLayer;
22 import org.ros.android.view.visualization.layer.Layer;
23 import org.ros.android.view.visualization.layer.OccupancyGridLayer;
24 import org.ros.android.view.visualization.layer.RobotLayer;
25 import org.ros.node.NodeConfiguration;
26 import org.ros.node.NodeMainExecutor;
27
28 import sensor_msgs.CompressedImage;
29
30 public class MainActivity extends RosActivity {
31     private int currentApiVersion;
32     private RosImageView<CompressedImage> rosImageView;
33
34     private static final String MAP_FRAME = "map";
35     private static final String ROBOT_FRAME = "base_link";
36
37     private final SystemCommands systemCommands;
38
39     private VisualizationView visualizationView;
40     private ToggleButton followMeToggleButton;
41     private CameraControlLayer cameraControlLayer;
42
43     private ConstraintLayout constraintMap;

```

```

44     private LinearLayout linearLayoutMap;
45
46     public MainActivity() {
47         super("Controle", "Controle");
48         systemCommands = new SystemCommands();
49     }
50
51     @Override
52     protected void onCreate(Bundle savedInstanceState) {
53         super.onCreate(savedInstanceState);
54         setContentView(R.layout.activity_main);
55
56         currentApiVersion = android.os.Build.VERSION.SDK_INT;
57         final int flags = View.SYSTEM_UI_FLAG_LAYOUT_STABLE
58             | View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION
59             | View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN
60             | View.SYSTEM_UI_FLAG_HIDE_NAVIGATION
61             | View.SYSTEM_UI_FLAG_FULLSCREEN
62             | View.SYSTEM_UI_FLAG_IMMERSIVE_STICKY;
63
64         if (currentApiVersion >= Build.VERSION_CODES.KITKAT) {
65             getWindow().getDecorView().setSystemUiVisibility(flags);
66             final View decorView = getWindow().getDecorView();
67             decorView.setOnSystemUiVisibilityChangeListener(new View.
68                 OnSystemUiVisibilityChangeListener() {
69                 @Override
70                 public void onSystemUiVisibilityChange(int visibility) {
71                     if ((visibility & View.SYSTEM_UI_FLAG_FULLSCREEN) == 0) {
72                         decorView.setSystemUiVisibility(flags);
73                     }
74                 });
75         }
76
77         rosImageView = findViewById(R.id.image);
78         rosImageView.setTopicName("/front_cam/camera/image/compressed");
79         rosImageView.setMessageType("sensor_msgs/CompressedImage");
80         rosImageView.setMessageToBitmapCallable(new BitmapFromCompressedImage());
81
82         constraintMap = findViewById(R.id.constraintMapId);
83         linearLayoutMap = findViewById(R.id.linearMapId);
84
85         visualizationView = findViewById(R.id.visualization);
86         cameraControlLayer = new CameraControlLayer();
87         visualizationView.onCreate(Lists.newArrayList(cameraControlLayer,
88             new OccupancyGridLayer("map"), new LaserScanLayer("scan"), new
89                 RobotLayer(ROBOT_FRAME)));
90         followMeToggleButton = findViewById(R.id.follow_me_toggle_button);
91         enableFollowMe();
92     }
93
94     @Override
95     protected void init(NodeMainExecutor nodeMainExecutor) {
96
97         NodeControle nodeControle = new NodeControle(this);

```



```

98
99     NodeConfiguration nodeConfiguration = NodeConfiguration.newPublic(
        InetAddressFactory.newNonLoopback().getHostAddress(), getMasterUri()
        );
100
101     nodeMainExecutor.execute(nodeControle, nodeConfiguration);
102     nodeMainExecutor.execute(rosImageView, nodeConfiguration);
103
104     visualizationView.init(nodeMainExecutor);
105     cameraControlLayer.addListener(new CameraControlListener() {
106         @Override
107         public void onZoom(float focusX, float focusY, float factor) {
108             disableFollowMe();
109         }
110
111         @Override
112         public void onTranslate(float distanceX, float distanceY) {
113             disableFollowMe();
114         }
115
116         @Override
117         public void onRotate(float focusX, float focusY, double deltaAngle)
118             {
119             disableFollowMe();
120         }
121
122         @Override
123         public void onDoubleTap(float x, float y) {
124         }
125     });
126
127     nodeMainExecutor.execute(visualizationView, nodeConfiguration);
128     nodeMainExecutor.execute(systemCommands, nodeConfiguration);
129 }
130
131 public void onClearMapButtonClicked(View view) {
132     toast("Limpendo mapa...");
133     systemCommands.reset();
134     enableFollowMe();
135 }
136
137 public void onSaveMapButtonClicked(View view) {
138     toast("Salvando mapa...");
139     systemCommands.saveGeotiff();
140 }
141
142 private void toast(final String text) {
143     runOnUiThread(new Runnable() {
144         @Override
145         public void run() {
146             Toast toast = Toast.makeText(MainActivity.this, text, Toast.
                LENGTH_SHORT);
147             toast.show();
148         }
149     });
150 }

```

```
151
152 public void onFollowMeToggleButtonClicked(View view) {
153     boolean on = ((ToggleButton) view).isChecked();
154     if (on) {
155         enableFollowMe();
156     } else {
157         disableFollowMe();
158     }
159 }
160
161 private void enableFollowMe() {
162     Preconditions.checkNotNull(visualizationView);
163     Preconditions.checkNotNull(followMeToggleButton);
164     runOnUiThread(new Runnable() {
165         @Override
166         public void run() {
167             visualizationView.getCamera().jumpToFrame(ROBOT_FRAME);
168             followMeToggleButton.setChecked(true);
169         }
170     });
171 }
172
173 private void disableFollowMe() {
174     Preconditions.checkNotNull(visualizationView);
175     Preconditions.checkNotNull(followMeToggleButton);
176     runOnUiThread(new Runnable() {
177         @Override
178         public void run() {
179             visualizationView.getCamera().setFrame(MAP_FRAME);
180             followMeToggleButton.setChecked(false);
181         }
182     });
183 }
184
185 public void onMapButtonClicked(View view) {
186     boolean on = ((ToggleButton) view).isChecked();
187     if (on) {
188         constraintMap.setVisibility(View.VISIBLE);
189         linearLayoutMap.setVisibility(View.VISIBLE);
190     } else {
191         constraintMap.setVisibility(View.GONE);
192         linearLayoutMap.setVisibility(View.GONE);
193     }
194 }
195
196 }
```

B.2 Classe *NodeControle.java*

```

1 package com.caiotf.android_app_ros;
2
3 import android.content.Context;
4 import android.hardware.Sensor;
5 import android.hardware.SensorEvent;
6 import android.hardware.SensorEventListener;
7 import android.hardware.SensorManager;
8 import android.support.constraint.ConstraintLayout;
9 import android.view.MotionEvent;
10 import android.view.View;
11 import android.widget.CompoundButton;
12 import android.widget.ToggleButton;
13
14 import org.ros.namespace.GraphName;
15 import org.ros.node.ConnectedNode;
16 import org.ros.node.Node;
17 import org.ros.node.NodeMain;
18 import org.ros.node.topic.Publisher;
19
20 import java.util.Timer;
21 import java.util.TimerTask;
22
23 import geometry_msgs.Twist;
24
25 public class NodeControle implements NodeMain {
26
27     ConstraintLayout layout_joystick, layout_joystick2;
28
29     JoyStickClass js, js2;
30
31     MainActivity activity;
32
33     public NodeControle(MainActivity mainActivity) {
34         activity = mainActivity;
35     }
36
37     private ToggleButton botaoAcelerometro;
38
39     private Publisher<Twist> publisher;
40     private geometry_msgs.Twist cmdVelCorrente;
41     private Timer publisherTimer;
42
43     private SensorManager mSensorManager;
44     private Sensor mAccelerometer;
45     private SensorEventListener sensorEventListener;
46
47     @Override
48     public GraphName getDefaultNodeName() {
49         return GraphName.of("");
50     }
51
52     @Override
53     public void onStart(final ConnectedNode connectedNode) {
54

```

```
55     botaoAcelerometro = activity.findViewById(R.id.buttonAceler);
56
57     mSensorManager = (SensorManager) activity.getSystemService(Context.
58         SENSOR_SERVICE);
59     mAccelerometer = mSensorManager.getDefaultSensor(Sensor.
60         TYPE_ACCELEROMETER);
61
62     layout_joystick = activity.findViewById(R.id.layout_joystick);
63     layout_joystick2 = activity.findViewById(R.id.layout_joystick2);
64
65     String nomeTopico = "~cmd_vel";
66     publisher = connectedNode.newPublisher(nomeTopico, geometry_msgs.Twist.
67         _TYPE);
68     cmdVelCorrente = publisher.newMessage();
69
70     publisherTimer = new Timer();
71     publisherTimer.schedule(new TimerTask() {
72         @Override
73         public void run() {
74             publisher.publish(cmdVelCorrente);
75         }
76     }, 0, 80);
77
78     activity.runOnUiThread(new Runnable() {
79         @Override
80         public void run() {
81
82             sensorEventListener = new SensorEventListener() {
83
84                 @Override
85                 public void onSensorChanged(SensorEvent event) {
86                     Float y = event.values[1];
87                     Float z = event.values[2];
88
89                     if (z >= -2 && z <= 2) {
90                         cmdVelCorrente.getLinear().setX(0);
91                     } else {
92                         if (z > 2) {
93                             cmdVelCorrente.getLinear().setX(1.0);
94                         }
95                         if (z < -2) {
96                             cmdVelCorrente.getLinear().setX(-1.0);
97                         }
98                     }
99
100                     if (y >= -1 && y <= 1) {
101                         cmdVelCorrente.getAngular().setZ(0);
102                     } else {
103                         if (y > 1) {
104                             cmdVelCorrente.getAngular().setZ(-1.0);
105                         }
106                         if (y < -1) {
107                             cmdVelCorrente.getAngular().setZ(1.0);
108                         }
109                     }
110                 }
111             }
112         }
113     });
```



```

158         } else if (direction == JoyStickClass.
159             STICK_DOWNLEFT) {
160             cmdVelCorrente.getLinear().setZ(-1.0);
161             cmdVelCorrente.getAngular().setZ(1.0);
162         } else if (direction == JoyStickClass.STICK_LEFT
163             ) {
164             cmdVelCorrente.getAngular().setZ(1.0);
165             cmdVelCorrente.getLinear().setZ(0);
166         } else if (direction == JoyStickClass.
167             STICK_UPLEFT) {
168             cmdVelCorrente.getLinear().setZ(1.0);
169             cmdVelCorrente.getAngular().setZ(1.0);
170         }
171     } else if (arg1.getAction() == MotionEvent.ACTION_UP
172         ) {
173         cmdVelCorrente.getLinear().setZ(0);
174         cmdVelCorrente.getAngular().setZ(0);
175     }
176     return true;
177 }
178 });
179
180 layout_joystick2.setOnTouchListener(new View.OnTouchListener
181     () {
182     public boolean onTouch(View arg0, MotionEvent arg1) {
183         js2.drawStick(arg1);
184         if (arg1.getAction() == MotionEvent.ACTION_DOWN
185             || arg1.getAction() == MotionEvent.
186                 ACTION_MOVE) {
187             int direction = js2.get8Direction();
188             if (direction == JoyStickClass.STICK_UP) {
189                 cmdVelCorrente.getLinear().setX(1.0);
190                 cmdVelCorrente.getLinear().setY(0);
191             } else if (direction == JoyStickClass.
192                 STICK_UPRIGHT) {
193                 cmdVelCorrente.getLinear().setX(1.0);
194                 cmdVelCorrente.getLinear().setY(-1.0);
195             } else if (direction == JoyStickClass.
196                 STICK_RIGHT) {
197                 cmdVelCorrente.getLinear().setY(-1.0);
198                 cmdVelCorrente.getLinear().setX(0);
199             } else if (direction == JoyStickClass.
200                 STICK_DOWNRIGHT) {
201                 cmdVelCorrente.getLinear().setX(-1.0);
202                 cmdVelCorrente.getLinear().setY(-1.0);
203             } else if (direction == JoyStickClass.STICK_DOWN
204                 ) {
205                 cmdVelCorrente.getLinear().setX(-1.0);
206                 cmdVelCorrente.getLinear().setY(0);
207             } else if (direction == JoyStickClass.
208                 STICK_DOWNLEFT) {
209                 cmdVelCorrente.getLinear().setX(-1.0);
210                 cmdVelCorrente.getLinear().setY(1.0);
211             } else if (direction == JoyStickClass.STICK_LEFT
212                 ) {
213                 cmdVelCorrente.getLinear().setY(1.0);
214                 cmdVelCorrente.getLinear().setX(0);

```

```

203         } else if (direction == JoyStickClass.
204             STICK_UPLEFT) {
205             cmdVelCorrente.getLinear().setX(1.0);
206             cmdVelCorrente.getLinear().setY(1.0);
207         }
208     } else if (arg1.getAction() == MotionEvent.ACTION_UP
209 ) {
210         cmdVelCorrente.getLinear().setX(0);
211         cmdVelCorrente.getLinear().setY(0);
212     }
213     return true;
214 }
215 }
216 }
217 }
218 }
219 }
220 }
221 }
222 }
223 }
224 }
225 }
226 }
227 }
228 }
229 }
230 }
231 }
232 }
233 }
234 }
235 }
236 }
237 }
238 }
239 }
240 }
241 }

```

```

242         if (arg1.getAction() == MotionEvent.
                ACTION_DOWN
243             || arg1.getAction() == MotionEvent.
                ACTION_MOVE) {
244             int direction = js2.get8Direction();
245             if (direction == JoyStickClass.
                STICK_RIGHT) {
246                 cmdVelCorrente.getLinear().setY
                    (-1.0);
247             } else if (direction == JoyStickClass.
                STICK_LEFT) {
248                 cmdVelCorrente.getLinear().setY(1.0)
                    ;
249                 cmdVelCorrente.getLinear().setX(0);
250             }
251         } else if (arg1.getAction() == MotionEvent.
                ACTION_UP) {
252             cmdVelCorrente.getLinear().setY(0);
253         }
254         return true;
255     }
256 });
257 }
258
259 if(!botaoAcelerometro.isChecked()) {
260     mSensorManager.unregisterListener(
        sensorEventListener, mAccelerometer);
261
262     layout_joystick.setOnTouchListener(new View.
        OnTouchListener() {
263         public boolean onTouch(View arg0, MotionEvent
            arg1) {
264             js.drawStick(arg1);
265             if (arg1.getAction() == MotionEvent.
                ACTION_DOWN
266                 || arg1.getAction() == MotionEvent.
                ACTION_MOVE) {
267                 int direction = js.get8Direction();
268                 if (direction == JoyStickClass.STICK_UP)
                {
269                     cmdVelCorrente.getLinear().setZ(1.0)
                        ;
270                     cmdVelCorrente.getAngular().setZ(0);
271                 } else if (direction == JoyStickClass.
                STICK_UPRIGHT) {
272                     cmdVelCorrente.getLinear().setZ(1.0)
                        ;
273                     cmdVelCorrente.getAngular().setZ
                        (-1.0);
274                 } else if (direction == JoyStickClass.
                STICK_RIGHT) {
275                     cmdVelCorrente.getAngular().setZ
                        (-1.0);
276                     cmdVelCorrente.getLinear().setZ(0);
277                 } else if (direction == JoyStickClass.
                STICK_DOWNRIGHT) {

```



```

278         cmdVelCorrente.getLinear().setZ
279             (-1.0);
280     } else if (direction == JoyStickClass.
281         STICK_DOWN) {
282         cmdVelCorrente.getLinear().setZ
283             (-1.0);
284         cmdVelCorrente.getAngular().setZ(0);
285     } else if (direction == JoyStickClass.
286         STICK_DOWNLEFT) {
287         cmdVelCorrente.getLinear().setZ
288             (-1.0);
289         cmdVelCorrente.getAngular().setZ
290             (1.0);
291     } else if (direction == JoyStickClass.
292         STICK_LEFT) {
293         cmdVelCorrente.getAngular().setZ
294             (1.0);
295         cmdVelCorrente.getLinear().setZ(0);
296     } else if (direction == JoyStickClass.
297         STICK_UPLEFT) {
298         cmdVelCorrente.getLinear().setZ(1.0)
299             ;
300         cmdVelCorrente.getAngular().setZ
301             (1.0);
302     }
303 } else if (arg1.getAction() == MotionEvent.
304     ACTION_UP) {
305     cmdVelCorrente.getLinear().setZ(0);
306     cmdVelCorrente.getAngular().setZ(0);
307 }
308 return true;
309 }
310 });
311
312 layout_joystick2.setOnTouchListener(new View.
313     OnTouchListener() {
314     public boolean onTouch(View arg0, MotionEvent
315         arg1) {
316         js2.drawStick(arg1);
317         if (arg1.getAction() == MotionEvent.
318             ACTION_DOWN
319             || arg1.getAction() == MotionEvent.
320                 ACTION_MOVE) {
321             int direction = js2.get8Direction();
322             if (direction == JoyStickClass.STICK_UP)
323             {
324                 cmdVelCorrente.getLinear().setX(1.0)
325                     ;
326                 cmdVelCorrente.getLinear().setY(0);
327             } else if (direction == JoyStickClass.
328                 STICK_UPRIGHT) {
329                 cmdVelCorrente.getLinear().setX(1.0)
330                     ;
331                 cmdVelCorrente.getLinear().setY
332                     (-1.0);

```

```

313         } else if (direction == JoyStickClass.
314             STICK_RIGHT) {
315             cmdVelCorrente.getLinear().setY
316                 (-1.0);
317             cmdVelCorrente.getLinear().setX(0);
318         } else if (direction == JoyStickClass.
319             STICK_DOWNRIGHT) {
320             cmdVelCorrente.getLinear().setX
321                 (-1.0);
322             cmdVelCorrente.getLinear().setY
323                 (-1.0);
324         } else if (direction == JoyStickClass.
325             STICK_DOWN) {
326             cmdVelCorrente.getLinear().setX
327                 (-1.0);
328             cmdVelCorrente.getLinear().setY(0);
329         } else if (direction == JoyStickClass.
330             STICK_DOWNLEFT) {
331             cmdVelCorrente.getLinear().setX
332                 (-1.0);
333             cmdVelCorrente.getLinear().setY(1.0)
334                 ;
335         } else if (direction == JoyStickClass.
336             STICK_LEFT) {
337             cmdVelCorrente.getLinear().setY(1.0)
338                 ;
339             cmdVelCorrente.getLinear().setX(0);
340         } else if (direction == JoyStickClass.
341             STICK_UPLEFT) {
342             cmdVelCorrente.getLinear().setX(1.0)
343                 ;
344             cmdVelCorrente.getLinear().setY(1.0)
345                 ;
346         }
347     } else if (arg1.getAction() == MotionEvent.
348         ACTION_UP) {
349         cmdVelCorrente.getLinear().setX(0);
350         cmdVelCorrente.getLinear().setY(0);
351     }
352     return true;
353 }
354 }
355 }
356 }
357 }
358 }
359 }
360 }
361 }
362 }
363 }
364 }
365 }
366 }
367 }
368 }
369 }
370 }
371 }
372 }
373 }
374 }
375 }
376 }
377 }
378 }
379 }
380 }
381 }
382 }
383 }
384 }
385 }
386 }
387 }
388 }
389 }
390 }
391 }
392 }
393 }
394 }
395 }
396 }
397 }
398 }
399 }
400 }
401 }
402 }
403 }
404 }
405 }
406 }
407 }
408 }
409 }
410 }
411 }
412 }
413 }
414 }
415 }
416 }
417 }
418 }
419 }
420 }
421 }
422 }
423 }
424 }
425 }
426 }
427 }
428 }
429 }
430 }
431 }
432 }
433 }
434 }
435 }
436 }
437 }
438 }
439 }
440 }
441 }
442 }
443 }
444 }
445 }
446 }
447 }
448 }
449 }
450 }
451 }
452 }
453 }
454 }
455 }
456 }
457 }
458 }
459 }
460 }
461 }
462 }
463 }
464 }
465 }
466 }
467 }
468 }
469 }
470 }
471 }
472 }
473 }
474 }
475 }
476 }
477 }
478 }
479 }
480 }
481 }
482 }
483 }
484 }
485 }
486 }
487 }
488 }
489 }
490 }
491 }
492 }
493 }
494 }
495 }
496 }
497 }
498 }
499 }
500 }
501 }
502 }
503 }
504 }
505 }
506 }
507 }
508 }
509 }
510 }
511 }
512 }
513 }
514 }
515 }
516 }
517 }
518 }
519 }
520 }
521 }
522 }
523 }
524 }
525 }
526 }
527 }
528 }
529 }
530 }
531 }
532 }
533 }
534 }
535 }
536 }
537 }
538 }
539 }
540 }
541 }
542 }
543 }
544 }
545 }
546 }
547 }
548 }
549 }
550 }
551 }
552 }
553 }
554 }
555 }
556 }
557 }
558 }
559 }
560 }
561 }
562 }
563 }
564 }
565 }
566 }
567 }
568 }
569 }
570 }
571 }
572 }
573 }
574 }
575 }
576 }
577 }
578 }
579 }
580 }
581 }
582 }
583 }
584 }
585 }
586 }
587 }
588 }
589 }
590 }
591 }
592 }
593 }
594 }
595 }
596 }
597 }
598 }
599 }
600 }
601 }
602 }
603 }
604 }
605 }
606 }
607 }
608 }
609 }
610 }
611 }
612 }
613 }
614 }
615 }
616 }
617 }
618 }
619 }
620 }
621 }
622 }
623 }
624 }
625 }
626 }
627 }
628 }
629 }
630 }
631 }
632 }
633 }
634 }
635 }
636 }
637 }
638 }
639 }
640 }
641 }
642 }
643 }
644 }
645 }
646 }
647 }
648 }
649 }
650 }
651 }
652 }
653 }
654 }
655 }
656 }
657 }
658 }
659 }
660 }
661 }
662 }
663 }
664 }
665 }
666 }
667 }
668 }
669 }
670 }
671 }
672 }
673 }
674 }
675 }
676 }
677 }
678 }
679 }
680 }
681 }
682 }
683 }
684 }
685 }
686 }
687 }
688 }
689 }
690 }
691 }
692 }
693 }
694 }
695 }
696 }
697 }
698 }
699 }
700 }
701 }
702 }
703 }
704 }
705 }
706 }
707 }
708 }
709 }
710 }
711 }
712 }
713 }
714 }
715 }
716 }
717 }
718 }
719 }
720 }
721 }
722 }
723 }
724 }
725 }
726 }
727 }
728 }
729 }
730 }
731 }
732 }
733 }
734 }
735 }
736 }
737 }
738 }
739 }
740 }
741 }
742 }
743 }
744 }
745 }
746 }
747 }
748 }
749 }
750 }
751 }
752 }
753 }
754 }
755 }
756 }
757 }
758 }
759 }
760 }
761 }
762 }
763 }
764 }
765 }
766 }
767 }
768 }
769 }
770 }
771 }
772 }
773 }
774 }
775 }
776 }
777 }
778 }
779 }
780 }
781 }
782 }
783 }
784 }
785 }
786 }
787 }
788 }
789 }
790 }
791 }
792 }
793 }
794 }
795 }
796 }
797 }
798 }
799 }
800 }
801 }
802 }
803 }
804 }
805 }
806 }
807 }
808 }
809 }
810 }
811 }
812 }
813 }
814 }
815 }
816 }
817 }
818 }
819 }
820 }
821 }
822 }
823 }
824 }
825 }
826 }
827 }
828 }
829 }
830 }
831 }
832 }
833 }
834 }
835 }
836 }
837 }
838 }
839 }
840 }
841 }
842 }
843 }
844 }
845 }
846 }
847 }
848 }
849 }
850 }
851 }
852 }
853 }
854 }
855 }
856 }
857 }
858 }
859 }
860 }
861 }
862 }
863 }
864 }
865 }
866 }
867 }
868 }
869 }
870 }
871 }
872 }
873 }
874 }
875 }
876 }
877 }
878 }
879 }
880 }
881 }
882 }
883 }
884 }
885 }
886 }
887 }
888 }
889 }
890 }
891 }
892 }
893 }
894 }
895 }
896 }
897 }
898 }
899 }
900 }
901 }
902 }
903 }
904 }
905 }
906 }
907 }
908 }
909 }
910 }
911 }
912 }
913 }
914 }
915 }
916 }
917 }
918 }
919 }
920 }
921 }
922 }
923 }
924 }
925 }
926 }
927 }
928 }
929 }
930 }
931 }
932 }
933 }
934 }
935 }
936 }
937 }
938 }
939 }
940 }
941 }
942 }
943 }
944 }
945 }
946 }
947 }
948 }
949 }
950 }
951 }
952 }
953 }
954 }
955 }
956 }
957 }
958 }
959 }
960 }
961 }
962 }
963 }
964 }
965 }
966 }
967 }
968 }
969 }
970 }
971 }
972 }
973 }
974 }
975 }
976 }
977 }
978 }
979 }
980 }
981 }
982 }
983 }
984 }
985 }
986 }
987 }
988 }
989 }
990 }
991 }
992 }
993 }
994 }
995 }
996 }
997 }
998 }
999 }
1000 }

```

```
354     publisherTimer.purge();
355 }
356
357
358     @Override
359     public void onError(Node node, Throwable throwable) {
360
361     }
362
363 }
```

B.3 Classe *SystemCommands.java*

```
1 package com.caiotf.android_app_ros;
2
3 import org.ros.namespace.GraphName;
4 import org.ros.node.AbstractNodeMain;
5 import org.ros.node.ConnectedNode;
6 import org.ros.node.Node;
7 import org.ros.node.topic.Publisher;
8
9 class SystemCommands extends AbstractNodeMain{
10     private Publisher<std_msgs.String> publisher;
11
12     @Override
13     public GraphName getDefaultNodeName() {
14         return GraphName.of("system_commands");
15     }
16
17     @Override
18     public void onStart(ConnectedNode connectedNode) {
19         publisher = connectedNode.newPublisher("syscommand", std_msgs.String.
20             _TYPE);
21     }
22
23     public void reset() {
24         publish("reset");
25     }
26
27     public void saveGeotiff() {
28         publish("savegeotiff");
29     }
30
31     private void publish(java.lang.String command) {
32         if (publisher != null) {
33             std_msgs.String message = publisher.newMessage();
34             message.setData( command);
35             publisher.publish(message);
36         }
37     }
38
39     @Override
40     public void onShutdown(Node arg0) {
41         publisher = null;
42     }
43 }
```

