UMA BIBLIOTECA DE AUXÍLIO AO DESENVOLVIMENTO DE APLICAÇÕES DE REDES MÓVEIS

Eduardo Gomes de Oliveira

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia Elétrica.

Orientadores: Miguel Elias Mitre Campista
Luís Henrique Maciel Kosmalski Costa

Rio de Janeiro
Março de 2016

UMA BIBLIOTECA DE AUXÍLIO AO DESENVOLVIMENTO DE APLICAÇÕES DE REDES MÓVEIS

Eduardo Gomes de Oliveira

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Examinada por:

Prof. Miguel Elias Mitre Campista, D.Sc.

Prof. Luís Henrique Maciel Kosmalski Costa, Dr.

Prof. Débora Christina Muchaluat Saade, D.Sc.

Prof. Igor Monteiro Moraes, D.Sc.

Prof. José Geraldo Ribeiro Júnior, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2016

Oliveira, Eduardo Gomes de

Uma Biblioteca de Auxílio ao Desenvolvimento de Aplicações de Redes Móveis/Eduardo Gomes de Oliveira. – Rio de Janeiro: UFRJ/COPPE, 2016.

XI, 49 p.: il.; 29, 7cm.

Orientadores: Miguel Elias Mitre Campista

Luís Henrique Maciel Kosmalski Costa

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia Elétrica, 2016.

Referências Bibliográficas: p. 47 – 49.

1. Bibliotecas de Programação. 2. Redes Móveis. 3. Android. 4. Redes Veiculares. 5. Smartphones. I. Campista, Miguel Elias Mitre *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

Dedico esta conquista
primeiramente a Deus, a minha
família, em especial a minha
esposa Gilmara.

Agradecimentos

Dedico esta conquista primeiramente a Deus e a minha família, em especial a minha esposa Gilmara.

Muitas foram as pessoas que me apoiaram e estiveram ao meu lado durante a elaboração deste trabalho. Assim, expresso a vocês os meus sinceros agradecimentos.

Agradeço aos professores Miguel Elias Mitre Campista, Luís Henrique Maciel Kosmalski Costa e José Geraldo Ribeiro Júnior pelo apoio prestado na elaboração da proposta do trabalho, pelas trocas de idéias e orientações.

Aos Coordenadores, professores e secretárias do curso por todo apoio dedicado durante os últimos dois anos.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

UMA BIBLIOTECA DE AUXÍLIO AO DESENVOLVIMENTO DE APLICAÇÕES DE REDES MÓVEIS

Eduardo Gomes de Oliveira

Março/2016

Orientadores: Miguel Elias Mitre Campista
Luís Henrique Maciel Kosmalski Costa

Programa: Engenharia Elétrica

As redes móveis são formadas a partir de tecnologias sem fio, como o Bluetooth, o Zigbee ou ainda o emergente WiFi Direct, e outras existentes em redes locais sem-fio e em redes celulares. Existe ainda uma abundância de dispositivos móveis cada vez mais equipados com sensores como GPS, acelerômetro e orientação. Toda essa diversidade de tecnologias e sensores representa uma complexidade adicional para o desenvolvimento de aplicações específicas, pois se traduzem em diferentes interfaces de programação. Assim, esta dissertação propõe uma biblioteca denominada ComuniCAR, projetado para facilitar o desenvolvimento de aplicações de redes móveis para smartphones. A biblioteca foi desenvolvida para o Android, um dos sistemas operacionais mais difundidos atualmente no mercado de smartphones e tablets. A biblioteca fornece uma interface de programação que abstrai os detalhes de acesso aos módulos de sensores internos e dos componentes de software do sistema Android. Assim, a extração de informação contextual, como velocidade, localização e sentido de movimento, a comunicação e o compartilhamento dessas informações entre os vizinhos se torna bem mais simples para o programador, exigindo deste a produção de menos linhas de código quando comparada à utilização de todas as interfaces de programação de aplicação específicas. Nesta dissertação, além da implementação da biblioteca, realizou-se uma análise qualitativa e quantitativa do processo de desenvolvimento dessas aplicações ao utilizá-lo.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

LIBRARY FOR THE DEVELOPMENT OF MOBILE NETWORK APPLICATIONS

Eduardo Gomes de Oliveira

March/2016

Advisors: Miguel Elias Mitre Campista
Luís Henrique Maciel Kosmalski Costa

Department: Electrical Engineering

Mobile networks are constructed from specific technologies, such as Bluetooth, Zigbee or the emerging WiFi Direct, and other existing ones from wireless local area networks and cellular networks. In addition, there are plenty of mobile devices increasingly rich in sensors such as GPS, accelerometer, and orientation. All this diversity sensor technology and represents additional complexity to the development of specific applications as translate into different programming interfaces. This dissertation proposes a ComuniCAR library designed to facilitate the development of mobile networking applications for smartphones. The proposed library was developed for Android, one of the most popular operating systems currently in the smartphones and tablet market. The ComuniCAR provides a programming interface that abstracts the details of access to the internal sensor module and the Android system software components. Thus, the contextual information extraction, such as speed, location and direction of movement, communication and sharing of such information between neighbors becomes simpler for the programmer, requiring this production of fewer lines of code compared to using all specific application programming interfaces. In this dissertation, and the implementation of the ComuniCAR library was held a qualitative and quantitative analysis of these application development process to use it.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Atualmente existe uma grande pluralidade de dispositivos móveis, com tamanhos, preços e tipos diferentes. Os tipos variam entre *smartphones*, *tablets*, *smartwatchs*, entre outros. Com a diminuição dos preços e aumento acelerado das suas capacidades, houve um crescimento acentuado de utilizadores, demandando a criação de aplicações de diferentes tipos para esta plataforma. Algumas aplicações necessitam de conectividade com outros dispositivos e os mesmos incorporam tecnologias de comunicação *device-to-device* (D2D), como é o caso do *Bluetooth*, e o mais emergente o *Wi-Fi Direct*, desenvolvido pela *Wi-Fi Alliance*.

Para facilitar o desenvolvimento de aplicações nesta área, interfaces de programação de aplicações (*Application Programming Interface* – API) são oferecidas de maneira isolada, frequentemente por sistemas operacionais móveis, como o *Android*. Tais interfaces, apesar de já ocultarem detalhes do dispositivo, ainda não permitem um nível de abstração mais elevado, fazendo com que os utilizadores se deparem com procedimentos pouco amigáveis. Sendo assim, esta dissertação propõe uma biblioteca denominada ComuniCAR que tem como objetivo facilitar o desenvolvimento de aplicações no contexto de redes móveis. A biblioteca abstrai detalhes de programação, notadamente os que se referem às APIs oferecidas pelo sistema operacional Android dos dispositivos.

A proposta de uma biblioteca é motivada por duas razões distintas. A primeira é que outras propostas da literatura não possuem versões de código aberto [3–5]. A segunda é que apesar do alto grau de complexidade no desenvolvimento das aplicações de redes móveis, a maioria delas utiliza um grande número de procedimentos semelhantes, que poderiam ser encapsuladas e oferecidas para o desenvolvedor.

Neste cenário, o ComuniCAR surge como uma contribuição no desenvolvimento de aplicações que auxiliem a execução de experimentos em redes móveis, facilitando a vida do programador. Esta biblioteca possui uma arquitetura e está dividida em três camadas: aplicação, desenvolvimento e dispositivo. A primeira, camada de aplicação, é a mais elevada e nela estão representadas as aplicações desenvolvidas por

terceiros que utilizam o ComuniCAR. Já a camada de desenvolvimento encapsula os procedimentos mais comuns a aplicações típicas de redes móveis, aumentando o grau de abstração das APIs do Android oferecida no nível mais inferior, nível de dispositivo.

A biblioteca proposta é específica para redes ad hoc e foi avaliada em termos de coesão e acoplamento, demonstrando alta coesão e baixo acoplamento. Adicionalmente, o ComuniCAR é avaliado a partir da construção de aplicações. Para tal, dois casos de uso foram avaliados, cada um utilizando uma aplicação específica. O primeiro caso de uso apresenta uma aplicação para obter os dados de localização provenientes do GPS nativo de um dispositivo baseado em Android. A segunda aplicação descobre quais vizinhos estão no entorno do dispositivo em um dado momento utilizando o WiFi Direct. Quando a comunicação entre os nós é realizada, os dados de localização são compartilhados entre eles. Considerando todos os casos de uso, a redução da quantidade de esforço do programador pode chegar até 70%.

De maneira resumida, as contribuições desta dissertação são as seguintes: (1) Apresentar as tecnologias utilizadas para o desenvolvimento do ComuniCAR, (2) Especificar o ComuniCAR, (3) Implementar o ComuniCAR (4) Validar o ComuniCAR.

O restante desta dissertação está organizado da seguinte forma. O Capítulo 2 apresenta as tecnologias utilizadas no desenvolvimento do ComuniCAR. O Capítulo 3 introduz os trabalhos relacionados a esta dissertação. Já o Capítulo 4 apresenta a implementação do ComuniCAR. No Capítulo 5 são apresentados dois casos de uso utilizando os recursos do ComuniCAR. Por fim, o Capítulo 6 apresenta as conclusões obtidas e tópicos para futuras investigações.

Capítulo 2

Fundamentação Teórica

Neste capítulo estão expostos os principais conceitos teóricos necessários ao desenvolvimento desta dissertação. Dentre esses conceitos estão: Bibliotecas de classes, Android, JSON, Redes ad hoc móveis e Wi-Fi Direct.

2.1 Desenvolvimento

2.1.1 Biblioteca de classes

Uma interface de programação de aplicativos (*Application Programming Interface* – API) é um conjunto de regras e especificações que permite a comunicação entre o software. As API's são vistas como as especificações de como utilizar uma biblioteca de classes, afim de realizar uma tarefa, por meio da interação com a biblioteca [6].

As bibliotecas de classes podem ser definidas como sendo a implementação de regras de uma API. Elas oferecem um conjunto de funções pré definidas, responsáveis por suprir as necessidades comuns de programadores em um dado domínio. As bibliotecas são auto-suficientes e abstraem um conjunto de funções comuns a um contexto, como exemplos, bibliotecas de *strings*, funções matemáticas, algoritmos de ordenação, manipulação de imagens e de estrutura de dados. Essas características concedem as programador a facilidade de escrever códigos menores e modularizados, garantindo maior manutenibilidade e legibilidade do código [6]. As bibliotecas ainda podem ser classificadas como bibliotecas estáticas ou blibliotecas compartilhadas, sendo que a diferença é na etapa de compilação.

- Bibliotecas estáticas: Esse tipo de biblioteca é conectada ao programa que faz uso dela durante a compilação, sem a necessidade de recompilar o arquivo da biblioteca. O principal motivo para o uso desse tipo de biblioteca é a redução de tempo de compilação [6]. O ComuniCAR é uma biblioteca estática que é compilada antes de ser utilizada nas aplicações.

- Bibliotecas compartilhadas: É um tipo de biblioteca que a partir da ativação do programa, é inserida, o que exige a sua recompilação. O programa não necessariamente inclui código da biblioteca, mas referências às funções contidas nela [6].

Os códigos desenvolvidos na biblioteca podem ser utilizados por vários programas sem a necessidade de detalhes de sua implementação. A grande vantagem de utilizar uma biblioteca, é que uma vez desenvolvida, não será mais preciso compilar, necessitando simplesmente de integrá-la ao programa desejado. Dessa forma, existe uma grande vantagem em utilizar bibliotecas, pois uma vez implementada, o desenvolvedor pode-se abstrair dos detalhes de programação e concentrar-se somente no problema principal.

Na documentação do Android é muito enfatizada a criação de módulos para conter o código comum do projeto, ou seja, as bibliotecas ou classes que o desenvolvedor pode reutilizar entre vários módulos [2]. Segundo Glauber[7], um módulo pode ser:

- `Java Library` - Contém apenas código Java, por meio do qual é gerado um arquivo JAR.

- `Android Library` - Além de código Java, pode ter recursos específicos do Android como *layouts*, imagens, entre outros. Ao ser compilado um arquivo ARR (*Android ARchive*) é gerado. O ComuniCAR é uma biblioteca que se enquadra nesse módulo.

- `Android Application` - É uma aplicação que será executada no aparelho e que pode depender de bibliotecas (locais ou externas) ou outros módulos. A maioria dos projetos só vai ter um desses módulos, e ao final será gerado o APK (*Android application*).

2.1.2 Android

Os recentes avanços na área de telecomunicações têm possibilitado um aumento exponencial na venda de *smartphones* e *tablets* nos últimos anos [8]. Segundo dados da *International Data Corporation* – (IDC), empresa global especializada em análise do mercado tecnológico publicado pela Revista Exame[9], somente no ano de 2013, foram vendidos mais de um bilhão de *smartphones* em todo mundo. Uma pesquisa foi realizada recentemente e a mesma empresa apontou que em 2015 o mercado de *smartphones* em todo o mundo cresceu 13%, com 341,5 milhões de unidades vendidas em um ano. Esta pesquisa aponta ainda que 82,8% desses aparelhos comercializados utilizam o sistema operacional Android. A Figura 2.1 apresenta o *ranking* das plataformas móveis mais utilizadas nos últimos quatro anos:

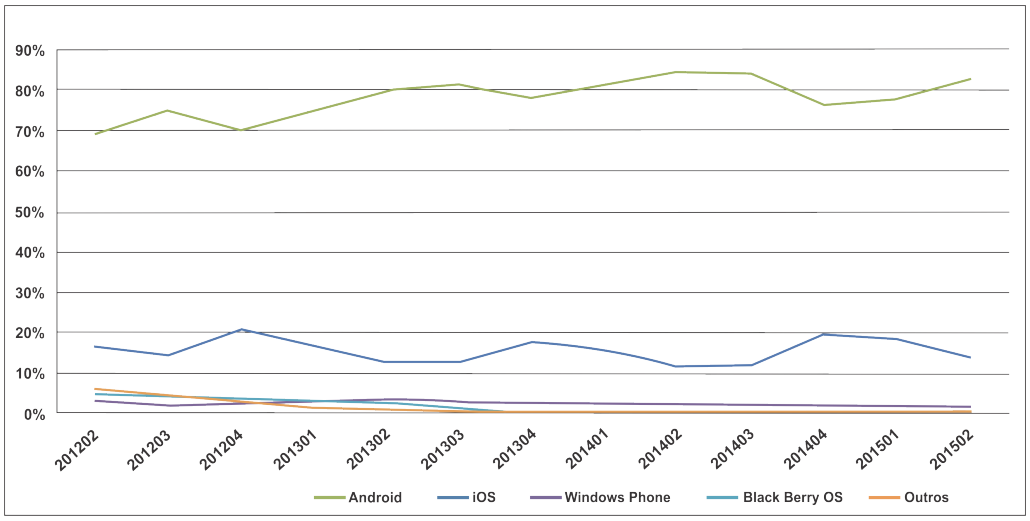

Figura 2.1: Ranking das plataformas móveis (Figura adaptada de [1]).

Em consequência disso, é crescente a demanda no desenvolvimento de aplicações para dispositivos móveis. A popularidade é tanta que o número de aplicativos disponíveis somente na *Google Play*, loja de aplicativos do Android já ultrapassa a marca de dois milhões de aplicativos [10]. Contudo, apesar do crescente número de aplicativos desenvolvidos diariamente, há uma carência no desenvolvimento de bibliotecas de programação para o Android que facilitam o desenvolvimento de aplicações no contexto de redes móveis. Estas bibliotecas poderiam ser exploradas no meio científico para a realização de experimentos reais nesta área.

A história do Android começou por um projeto inicialmente desenvolvido por uma *startup* americana em Outubro de 2003 chamada Android Inc. Essa empresa foi adquirida pela Google em Agosto de 2005, que por sua vez amadureceu o projeto e o tornou público em 2007 com o objetivo de apresentar a primeira plataforma *open source* de desenvolvimento para dispositivos móveis [11]. Para os fabricantes de *smartphones*, o fato de existir uma plataforma única, consolidada, livre e *open source* é uma grande vantagem para criar novos aparelhos. E foi assim que surgiu o (*Open Handset Alliance* - OHA), um grupo constituído por aproximadamente 84 empresas líderes em tecnologia móvel liderados pelo Google. Entre alguns integrantes do grupo estão nomes como Intel, HTC, LG, Motorola, Samsung, Sony Ericsson, Toshiba, Huawei, Sprint Nextel, China Mobile, T-Mobile, ASUS, Acer, Dell, Garmin, entre outros. Os objetivos principais do grupo é definir uma plataforma única e aberta para *smartphones* para deixar os consumidores mais satisfeitos com o produto final e a criação de uma plataforma moderna e flexível para o desenvolvimento de aplicações coorporativas. O resultado dessa união foi o nascimento do Android. [2].

Conforme mencionado, o Android é a primeira plataforma para aplicações móveis livre e *open source*, o que representa uma grande vantagem competitiva para sua evolução, uma vez que diversas empresas e desenvolvedores do mundo podem contri-

buir para melhorar a plataforma. Para os fabricantes de *smartphones*, isso também é uma grande vantagem, pois é possível utilizar o sistema operacional do Android em seus *smartphones* sem ter que pagar por isso. Além disso, a licença (*Apache Software Foundation* - ASF) permite que alterações sejam efetuadas no código-fonte para criar produtos customizados sem precisar compartilhar as alterações com ninguém.

Em relação à arquitetura, o Android é uma plataforma que apresenta desde sistema operacional, até *middleware* e aplicativos, conforme pode ser observado na Figura 2.2. Sua arquitetura é dividida em camadas que são: aplicativos, arcabouços, bibliotecas e núcleo do sistema operacional [12]. Abaixo, será descrito cada uma delas:

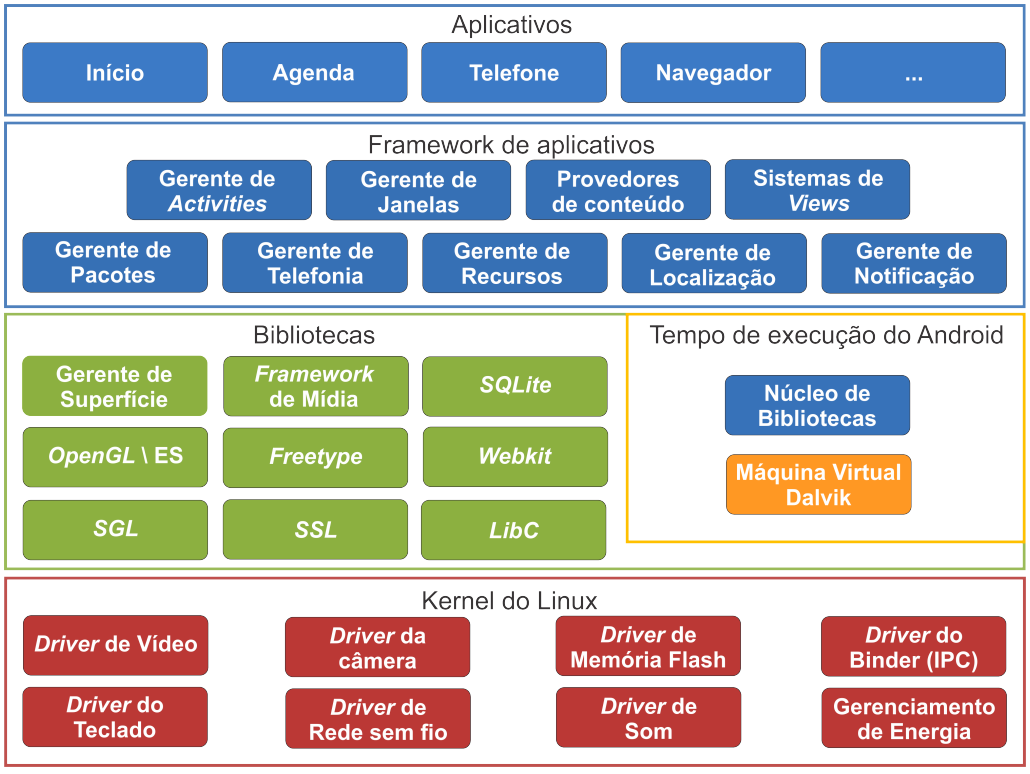

Figura 2.2: Arquitetura do Android (Figura adaptada de [2]).

Na camada de aplicativos (*Applications*) da arquitetura Android, estão representadas as aplicações que são executadas sobre a plataforma, sejam elas nativas como o caso da calculadora, do gerenciador de contatos, do calendário, entre outros, ou aplicações desenvolvidas por terceiros como é o caso da aplicação SmartCAR que será apresentada neste trabalho utilizando os recursos do ComuniCAR. Para a camada de aplicativos, não existe diferença entre aplicações nativas e aplicações de terceiros, todas são escritas com as mesmas interfaces de programação e executadas em paralelo, inclusive tendo a possibilidade da troca de uma aplicação nativa por outra que tenha a mesma finalidade e seja desenvolvida por um terceiro ou pelo próprio usuário [12].

Na camada de *framework de aplicativos* (*Application Framework*), estão localizadas as interfaces de programação e os recursos que são utilizados pelos aplicativos que executam sobre a plataforma do Android, como por exemplo, cliente de email, despertador, jogos, mapas, calendários, Internet, browser, entre outros. Na camada de bibliotecas (Libraries), ficam as interfaces de programação desenvolvidas em C/C++ e que dão suporte dentre outros recursos a renderização 3D (Open GL ES), suporte aos diversos formatos de vídeo e áudio e gerenciamento de base de dados (SQLite).

Na pequena camada do ambiente de execução conhecida como tempo de execução do Android (Android *Runtime*), encontra-se componentes como as *core libraries*, que disponibilizam a API Java necessária para a escrita do código de programação das aplicações, bem como a DVM (*Dalvik Virtual Machine*), que é a máquina virtual com melhor desempenho, maior integração com a nova geração de hardware e projetada para executar várias máquinas virtuais paralelamente. Ela foi projetada para funcionar em sistemas com baixa frequência de CPU, pouca memória RAM disponível e sistema operacional sem espaço de *Swap*, oferecendo condições para que a aplicação Java desenvolvida possa ser executada.

Na camada do núcleo (Linux *Kernel*), baseada em Linux, localiza-se o sistema operacional da plataforma, responsável por serviços denominados de baixo nível como gerenciamento de processos, gerenciamento de memória, segurança, entre outros e ainda possui vários *drivers* de hardware.

Atualmente os dispositivos Android incluem *smartphones*, *tablets*, *e-readers*, robôs, motores a jato, satélites da NASA, console de jogos, televisões, câmeras, geladeiras, sistemas automotivos de *infotainment* que servem para controlar rádio, GPS, ligações telefônicas, termostato, entre outros, equipamentos voltados para a saúde, relógios inteligentes (*smartwatches*) e muitos outros [13].

2.1.3 Ambiente de Desenvolvimento

O Android Studio é um ambiente de desenvolvimento integrado (*Integrated Development Environment* – IDE) oficial de desenvolvimento de aplicações para a plataforma Android. Foi anunciado em 16 de Maio de 2013, na conferência Google I/O e é disponibilizado gratuitamente sobre a Licença Apache 2.0 [2]. O Android SDK é o software utilizado para desenvolver aplicações no Android, que tem um emulador para simular o dispositivo, ferramentas utilitárias e uma API completa para a linguagem Java. Tal API é instalada juntamente com o Android Studio [2], contendo todas as classes necessárias para o desenvolvimento de aplicações. As ferramentas do Android SDK compilam o código, em conjunto com todos os arquivos de dados e recursos, em um APK, que é o pacote do Android, com sufixo .apk. Os arquivos

de APK contêm todo o conteúdo de um aplicativo do Android e são os arquivos que os dispositivos desenvolvidos para Android usam para instalar o aplicativo [14].

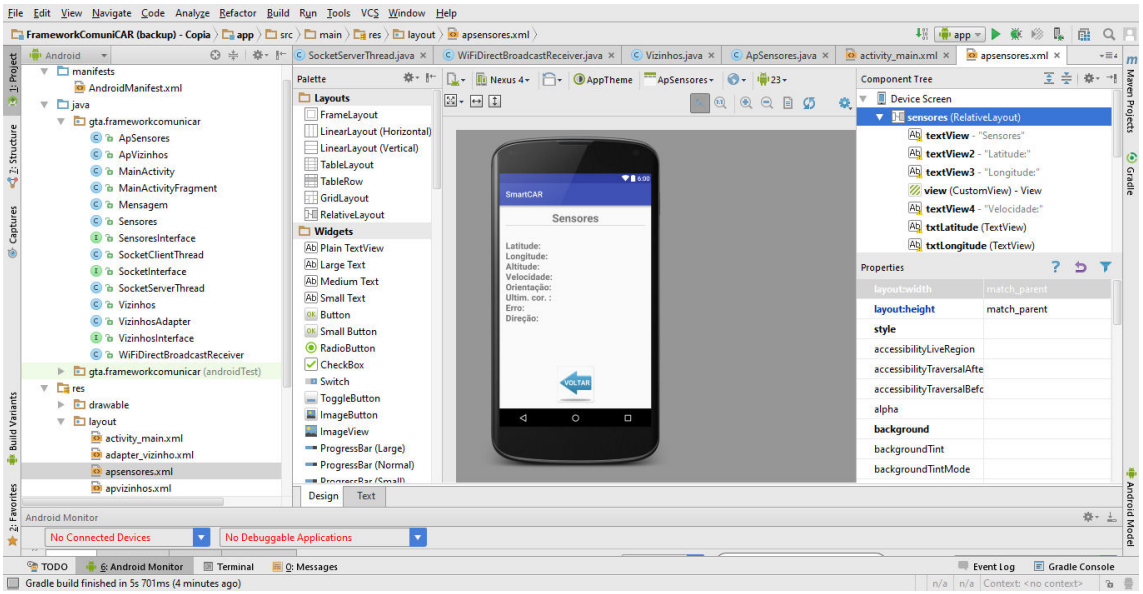

Figura 2.3: Captura de tela do Android Studio 1.5.1.

A implementação do ComuniCAR foi realizada através do Android Studio. Além disso, é importante lembrar que o desenvolvimento de aplicações para a plataforma Android é feita através da linguagem de programação Java.

Elementos de uma aplicação

Segundo Pereira et al. [12], o Android possui um grupo de componentes essenciais que o sistema pode instanciar e executar sempre que for necessário, constituído pelos seguintes elementos:

- `Activities`: Activity ou Atividade é o mais utilizado de todos os componentes. Geralmente representa uma tela de aplicação e é responsável por controlar os eventos da tela e definir qual "Vista"é responsável por desenhar a interface gráfica do usuário [2]. Segundo o mesmo autor, é importante entender o ciclo de vida de uma activity, com o intuito de desenvolver aplicações mais robustas. A Figura 2.4 apresenta os métodos do ciclo de vida da activity, descritos logo a seguir [12]:
  - `onCreate()`: Este método é chamado uma única vez. O objetivo desse método é fazer a inicialização necessária para executar a aplicação.
  - `onStart()`: Este método é chamado quando a atividade está ficando visível aos usuários.

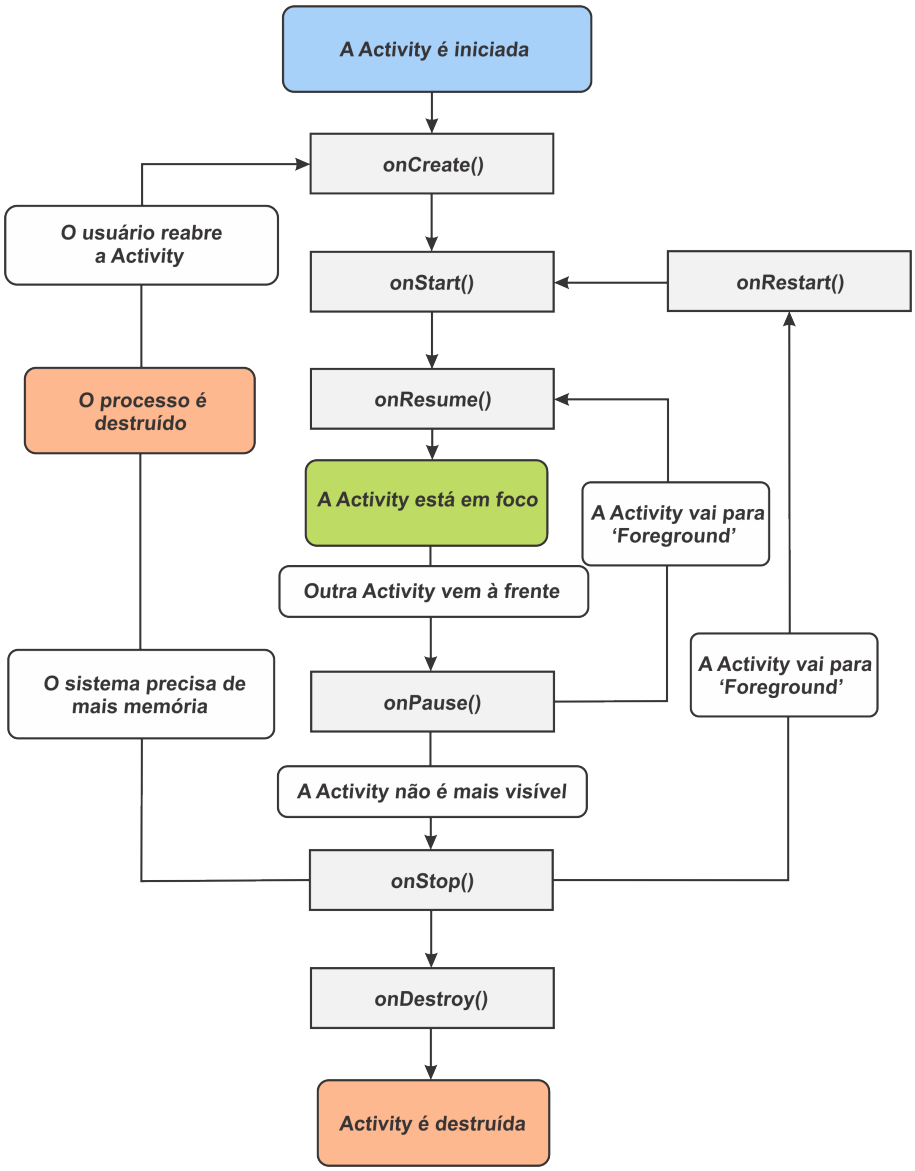

Figura 2.4: Ciclo de vida de uma activity (Figura adaptada de [2]).

– `onRestart()`: Este método é chamado quando quando uma atividade foi parada temporariamente e está sendo iniciada outra vez.

– `onResume()`: É o topo da pilha de atividade, chamado quando vai iniciar a interação com o usuário. Pode-se dizer que este método representa o estado de que a activity está executando.

– `onPause()`: Este método será chamado sempre que a tela da atividade fechar. Isso pode acontecer se o usuário pressionar o botão "Home", ou o botão "Sai" da aplicação, ou ainda quando o usuário receber uma ligação telefônica. Nesse momento, esse método é chamado para salvar o estado da aplicação, para que posteriormente, quando a atividade voltar e executar, tudo possa ser recuperado, no método `onResume()` citado anteriormente [2].

  - `onStop()`: Este método é chamado quando a atividade não estiver mais sendo utilizada pelo usuário e perdeu o foco para outra atividade.
  - `onDestroy()`: Este método pode ser chamado quando a atividade terminou, ou quando o sistema precisa finalizar atividades para liberação de recursos.
  - `onFreeze()`: Este método possibilita salvar o estado atual da atividade.

- `Services`: São códigos sem interfaces de usuário, que rodam em background, não sendo interrompidos quando da troca de atividade pelo usuário. Ao contrário da *Activity*, que tem um ciclo de vida próprio e uma vida curta, os *Services* mantém o serviço ativo até que seja recebida outra ordem. Uma vez conectado ao serviço, pode-se comunicar com este através de uma interface apresentada para o usuário [12].

- `Broadcast Services`: Quando a aplicação recebe um evento externo através de uma intenção, é este componente que trata a reação deste evento. Ele pode ser acionado pelo tocar do telefone, quando existirem redes disponíveis, um determinado horário, entre outros eventos previamente programados. Não possui uma interface de usuário, mas pode fazer uso de uma *Notification Manager*, como forma de interface para o usuário, que é um mecanismo de alerta que na sua forma mais comum, aparece como pequeno ícone na barra de status, para notificar sobre alguma ocorrência.

- `Content Provider`: É utilizado quando há necessidade de compartilhar os dados entre as aplicações.

- `Intents e Intent Filters`: *Activities*, *services* e *Broadcast Receivers* são ativados por mensagens assíncronas, denominadas *Intents*, diferentemente dos *Content Providers*, que são ativados por requisições de um *Content Resolver*. Uma *intent* é um objeto que detém o conteúdo de uma mensagem. Por exemplo, uma *activity*, poderia transmitir o pedido de mostrar uma foto, e para um *Broadcast*, poderia anunciar que a bateria está descarregada. As *Intent Filters* servem para descrever quais *Intents* uma *Activity* ou *Broadcast Receiver* são capazes de tratar.

- `AndroidManifest.xml`: Toda aplicação deve ter um arquivo *AndroidManifest.xml* em seu diretório raiz. Ele é um arquivo de configuração que descreve os elementos da aplicação, as classes de cada componente utilizado, o tipo de dado que ele pode tratar, ou seja, serve para definir os dados de cada elemento. As especificações de permissões também podem ser definidas neste arquivo.

As interfaces de programação estabelecidas para o Android permitem total modificação por meio de programação do seu conteúdo. No entanto, programas que não necessitam envolver-se em detalhes de implementação do software podem apenas utilizar os serviços, sem a preocupação de como funciona, utilizando apenas características menos evidentes ao usuário padrão. Um ponto forte das interfaces básicas do Android é a otimização que estas possuem, focando a utilidade dos pacotes, em conjunto com um bom aproveitamento, excluindo pacotes pesados e poucos evoluídos. Com essas interfaces, podem ser criadas todas as interfaces com usuário, permitindo a criação de telas, acesso a arquivos, criptografar dados, ou seja, utilizar a funcionalidade definida pelo utilizador [12]. Segue abaixo as principais interfaces utilizadas neste trabalho [14]:

- `android.location:` Oferece serviços de geolocalização para a aplicação Android. O componente central de localização é o serviço de sistema `LocationManager` que fornece a API responsável por determinar a localização geográfica. Essa API tem como vantagem o baixo consumo de energia do dispositivo.

- `android.net.wifi.p2p:` Essa API fornece classes para interagir com o hardware WiFi do dispositivo para descobrir e conectar com seus pares. O componente central dessa API é o *WifiP2pManager*.

- `android.util:` Essa API contém todos os recursos que uma aplicação poderá usar como áudio, vídeo, arquivos XML, entre outros. São separados da aplicação com o objetivo de ocupar menos espaço e demorar menos tempo para que sejam carregados.

- `android.os:` Contém serviços referentes ao sistema operacional, passagem de parâmetros e comunicação entre processos.

- `android.provider:` API que contém os padrões de provedores de conteúdos, conhecidos como (*Content Providers*. É responsável pela disponibilização dos dados através das aplicações tornando esses dados públicos. Quase todo o tipo de dado é compartilhável, como áudio, vídeo, texto e imagens.

- `android.content:` Contém as APIs de acesso a dados no dispositivo, como as aplicações instaladas e seus recursos.

- `android.view:` Pacote que contém as principais funções e componentes de interface gráfica.

- `android.widget:` Contém *widgets* prontos (botões, listas, grades, entre outros) para serem utilizadas nas aplicações.

- `android.app`: Possui APIs de alto nível referente ao modelo da aplicação. O *Activity Manager* que está incluído nessa API, é responsável pelo gerenciamento de cada atividade do sistema.

- `android.telephony`: API para interagir com funcionalidades de telefonia e telecomunicação.

2.1.4 JSON

No contexto da implementação do ComuniCAR, o JSON foi utilizado para formatar os dados obtidos dos sensores, antes de enviá-los para outros dispositivos. A transferência de dados, dessa forma, se torna mais simples. O JSON (*JavaScript Object Notation*) é uma formatação leve, baseado em texto, que facilita o intercâmbio de dados estruturados [15]. O JSON pode ser visto como um formato "universal" que é muito conveniente para troca de informações entre aplicações através de diversos protocolos. O JSON está baseado em um subconjunto da linguagem de programação JavaScript, mas é completamente independente de linguagem, pois usa convenções que são familiares às linguagens C e familiares, incluindo C++, Java, JavaScript, Perl, Python, entre outras. Essas propriedades fazem com que JSON seja um formato ideal para a troca de dados entre as aplicaçõe [16]. O JSON está constituído em duas estruturas:

- `Uma coleção de pares nome/valor`. Em várias linguagens, isto é caracterizado como um *object*, *record*, *struct*, dicionário, *hash*, *table*, *keyed list*, ou *arrays* associativas.

- `Uma lista ordenada de valores`. Na maioria das linguagens, esse tipo é caracterizado como um *array*, vetor, lista ou sequência.

Todas as estruturas de dados consideradas pelo JSON são ditas universais. Virtualmente todas as linguagens de programação modernas as suportam, de uma forma ou de outra. É aceito inclusive que um formato de troca de dados que seja independente de linguagem de programação se baseia nessas estruturas. Abaixo, segue um exemplo de código em JSON.

Código 1. Array de Objetos em JSON.

```
1 [
2       {
3           "Latitude":  -21.5228,
4           "Longitude": -42.6357,
5           "Altitude": 110,
6           "Velocidade": 40,
7           "Orientacao": "N",
```

```
            "Erro": 6,
            "Direcao": 10
        },
        {
            "Latitude":  -21.5678,
            "Longitude": -42.9876,
            "Altitude": 98,
            "Velocidade": 50,
            "Orientacao": "N",
            "Erro": 4,
            "Direcao": 14
        }
]
```

2.2 Redes Móveis

Uma das principais características dos dispositivos móveis está na possibilidade do usuário ter acesso a qualquer momento e em qualquer lugar. Para que isso ocorra, esses equipamentos fazem uso de um tipo especial de rede, onde não existe a necessidade do uso de cabos, a qual é conhecida como rede sem fio. Apesar de existirem muitos padrões de redes sem fio na literatura como o *Bluetooth*, o Wi-Fi, o LTE e o 3G/4G, esta seção foca na tecnologia que foi utilizada no desenvolvimento do ComuniCAR, ou seja, no emergente Wi-Fi Direct.

2.2.1 Wi-Fi Direct

O Wi-Fi Direct é uma especificação recentemente desenvolvida pela Wi-Fi Alliance que opera em dispositivos IEEE 802.11, na faixa de 2,4 e 5 GHz. Essa especificação possui alcance de transmissão de aproximadamente 200 metros, podendo alcançar a velocidade de transmissão de até 250 Mb/s. O Wi-Fi Direct visa introduzi a capacidade de conexão direta a milhares de dispositivos que já possuem Wi-Fi implantado [17]. Segundo Casetti *et al.* [18], essa especificação tem como objetivo permitir a comunicação entre os dispositivos com Wi-Fi Direct sem a necessidade de um ponto de acesso.

A comunicação entre os dispositivos ocorre dentro de um único grupo. Um dos dispositivos no grupo atua como proprietário (*Grupo Owner* - GO) e os outros dispositivos são chamados de clientes. A Figura 2.5 ilustra como é realizada a comunicação entre os dispositivos.

Alguns dos benefícios dos dispositivos Wi-Fi Direct são [17]:

- `Mobilidade e Portabilidade:` Os dispositivos com Wi-Fi Direct podem se

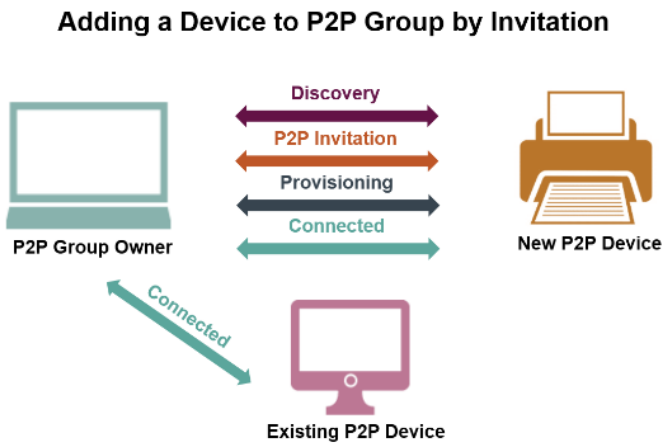

Figura 2.5: Comunicação entre os dispositivos.

conectar a qualquer momento e de qualquer lugar, pois não necessitam de um roteador Wi-Fi.

- `Uso Imediato:` Os usuários têm a possibilidade de criar conexões com outros dispositivos certificados para Wi-Fi Direct que eles levem para a casa.
- `Facilidade de Uso:` Os serviços de descoberta de dispositivos do Wi-Fi Direct permitem que usuários identifiquem dispositivos e serviços disponíveis antes de estabelecer uma conexão.
- `Conexões Seguras:` Os dispositivos com Wi-Fi Direct utilizam o Wi-Fi *Protected Setup* para a criação de conexões seguras entre os dispositivos.
- `Desenvolvimento de aplicações:` Wi-Fi Direct fornece uma API que torna mais simples o desenvolvimento de aplicações inovadoras para os usuários.

O Wi-Fi Direct foi escolhido nesta dissertação por ser uma tecnologia de rede P2P ainda pouco explorada no meio científico e que possui grandes possibilidades de crescimento no mercado de dispositivos móveis nos próximos anos. Como o Wi-Fi Direct é uma extensão do Wi-Fi, a implementação dele junto ao ComuniCAR ocasiona ganhos com relação à area de cobertura e taxa de transmissão em relação ao *Bluetooth* que são menores.

2.2.2 Redes *ad hoc*

As redes ad hoc móveis (*Mobile Ad hoc NETworks - MANETs*) são constituídas por dispositivos móveis que utilizam comunicação sem fio. A principal característica dessas redes é a ausência de um elemento centralizador, como pontos de acesso ou estações-base existentes em outras redes locais sem fio ou ainda nas redes de telefonia

celular. A comunicação entre nós que estão fora do alcance de transmissão do rádio é feita em múltiplos saltos através da colaboração de nós intermediários. Além disso, a topologia da rede pode mudar dinamicamente devido à mobilidade dos nós. [19].

Estas redes podem ser montadas de maneira rápida e fácil e em praticamente qualquer lugar. Por dispensar um dispositivo que atue como ponto de acesso faz com que a rede se torne mais viável em relação ao custo de instalação e mais fácil de configurar [20].

Estas redes são úteis em cenários onde não é possível se instalar uma infraestrutura, ou também onde não há necessidade de infraestrutura devido ao caráter temporário da rede. As aplicações podem ser na área militar, conferências e seminários, VANETs (*Vehicular Ad hoc NETwork*), serviços de emergência em desastres naturais, salas de aula ou simples necessidade de troca de arquivos [20].

Como foi dito anteriormente, o ComuniCAR poderia ser utilizado no desenvolvimento de aplicações para VANETs. Por exemplo, uma aplicação de alerta de colisão poderia ser implementada obtendo os dados de localização, sentido e velocidade que são oferecidos pelos sensores do dispositivo. Ao descobrir e estabeler uma conexão entre os dispositivos utilizando o Wi-Fi Direct, esses dados poderiam ser compartilhados para todos os dispositivos desse grupo. Após obter estas informações, poderia ser calculado a distância segura de acordo com a velocidade relativa de cada dispositivo. Se a velocidade final oferecer risco de colisão iminente, o condutor seria avisado por um alerta. Outro exemplo de aplicação na área de entretenimento poderia ser desenvolvida utilizando os recursos do ComuniCAR. Com os mesmos recursos utilizados de localização, sentido e velocidade e de conectividade citados no exemplo anterior, os dispositivos poderiam compartilhar arquivos como áudio e vídeo.

Capítulo 3

Trabalhos Relacionados

Neste capítulo estão catalogados os trabalhos correlatos de bibliotecas que facilitam o desenvolvimento de aplicações para redes móveis. Neste capítulo também é apresentado um comparativo entre os trabalhos correlatos e o ComuniCAR.

Park *et al.* [3] propuseram o VoCell, um arcabouço de software que fornece uma biblioteca que visa simplificar o desenvolvimento de aplicações para *smartphones* baseados no sistema Android no contexto de redes veiculares. Utilizando o VoCell, os desenvolvedores podem acessar facilmente uma ampla variedade de informações de sensores embarcados nos *smartphones* e compartilhar esses dados através de servidores na nuvem como o *Amazon Cloud Service*, *Microsoft Azure* ou *Google App Engine*. Um desenvolvedor de aplicativos, mesmo sem grande conhecimento em programação como a linguagem Java, é capaz de criar aplicações no contexto de redes veiculares de forma simplificada, incluindo VoCell como uma biblioteca nos seus projetos. Este arcabouço oferece recursos para a criação de aplicações do tipo infraestruturada. O VoCell foi avaliado no desenvolvimento de aplicações na área de segurança veicular como monitoramento do trânsito em tempo real, alerta de colisão, entre outras, demonstrando bons resultados. Diferente do VoCell, o ComuniCAR tem como proposta oferecer recursos para o desenvolvimento de aplicações mais genéricas no contexto de redes móveis.

Outro arcabouço para a plataforma Android, denominado Reivent, foi proposto por [4]. O Reivent oferece uma biblioteca e foi utilizada no desenvolvimento de duas aplicações na área de entretenimento para redes veiculares: *VNChat* e *iThere*. O *VNChat* é uma aplicação de troca de mensagens (*Chat*) onde o usuário pode enviar mensagem de texto através da rede para outro usuário. Já o *iThere* é um aplicativo que tem como objetivo informar ao usuário sobre os amigos (vizinhos) que estão próximos. Ele funciona como uma rede social para redes veiculares, transmitindo a localização em tempo real do usuário e de seus vizinhos próximos, utilizando o serviço do Google Maps. O ComuniCAR oferece recursos para desenvolvimento de aplicações mais genéricas no contexto de redes móveis.

Um fator importante observado nas pesquisas é que nenhum dos trabalhos citados anteriormente oferecem de forma gratuita as bibliotecas que foram implementadas para o estudo e utilização para o desenvolvimento de outras aplicações. Assim, percebeu-se uma oportunidade de desenvolver uma biblioteca de software voltado para redes móveis que seja livre e extensível. Como consequência, outros pesquisadores da área poderiam utilizá-la e ao mesmo tempo contribuir, desenvolvendo novas funcionalidades para a criação de diversas aplicações na área de redes móveis utilizando *smartphones* baseados em Android.

O objetivo é que com o uso do ComuniCAR, as pesquisas nessa área possam se intensificar.

Capítulo 4

O ComuniCAR

Neste capítulo é descrito a biblioteca proposta para esta dissertação, o ComuniCAR. Inicialmente, é feita uma breve introdução a respeito dessa biblioteca, o motivo de seu desenvolvimento e as vantagens de sua utilização. Em seguida, a arquitetura do ComuniCAR é apresentada, evidenciando cada camada que a compõe. Além disso, neste capítulo, também é exposto o funcionamento dessa biblioteca e como a sua implementação foi realizada.

O objetivo principal do ComuniCAR é facilitar o desenvolvimento de aplicações para redes móveis através do encapsulamento de procedimentos dependentes de *drivers* ou sistemas operacionais móveis. Nesta dissertação esta biblioteca foi utilizada para criar uma aplicação chamada SmartCAR que será tratado no capítulo 5. Esta biblioteca foi proposta com vistas à extensão, permitindo que os desenvolvedores possam estendê-la a fim de atender as necessidades dos aplicativos que desejarem implementar. Assim, torna-se mais simples criar novas funcionalidades não pensadas em um primeiro momento ou melhorar as funcionalidades existentes.

A idéia é que o ComuniCAR seja disponibilizado de forma gratuita. Assim, quando o usuário for utilizá-lo, deverá incluir os fontes em seu projeto e estender os trechos de código que achar necessário para o desenvolvimento de sua aplicação. A Figura 4.1 apresenta a arquitetura do ComuniCAR.

Ao fornecer aos desenvolvedores um conjunto de funcionalidades já implementadas no segmento de redes móveis, espera-se facilitar a investigação e estimular o desenvolvimento de aplicações nesta área.

4.1 Arquitetura

A arquitetura do ComuniCAR possui três camadas de implementação: *Interfaces de Dispositivo*, *Interfaces de Desenvolvimento* e *Aplicação*, como apresentado na Figura 4.1. De maneira resumida, a camada mais baixa da arquitetura, chamada de *Interfaces de Dispositivo*, oferece as APIs para acesso às funcionalidades

dos dispositivos (*smartphones*). Tais interfaces são disponibilizadas por bibliotecas do sistema operacional Android, ou por drivers instalados no dispositivo pelos fabricantes. A camada de Interfaces de desenvolvimento é responsável por simplificar o desenvolvimento das aplicações, através da ocultação de detalhes das interfaces dos dispositivos oferecidos pelo ComuniCAR. Por fim, a camada mais elevada da biblioteca são as aplicações desenvolvidas por terceiros que poderão utilizar o ComuniCAR. A seguir, cada uma das camadas do ComuniCAR é apresentada em maiores detalhes.

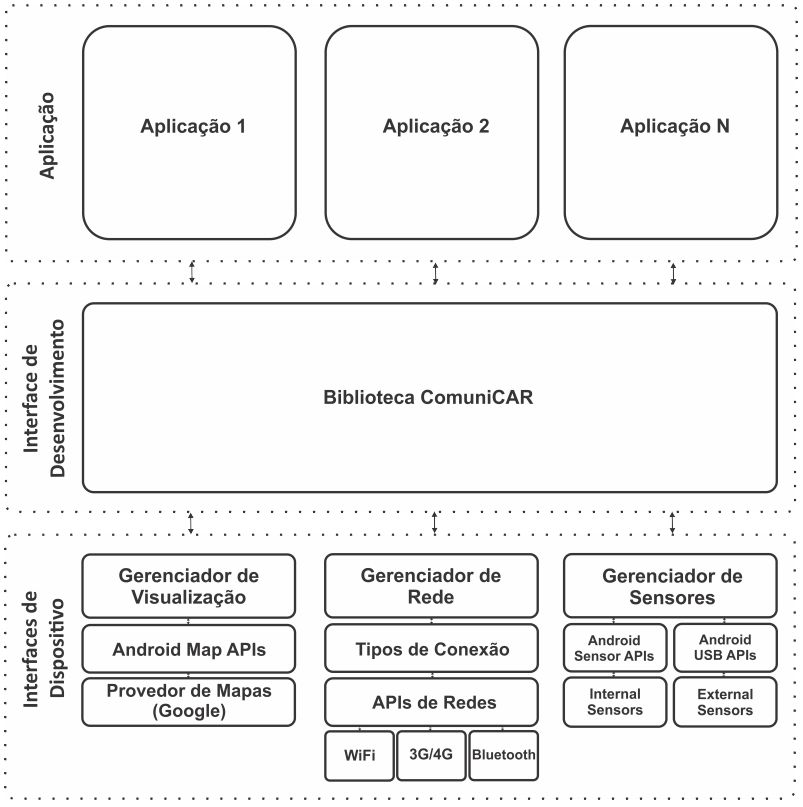

Figura 4.1: Arquitetura da biblioteca proposta ComuniCAR composta por três camadas: Interfaces de Dispositivo, Interfaces de Desenvolvimento e Aplicação.

4.1.1 Camada de Interfaces de Dispositivo

A camada Interfaces de Dispositivo é responsável por realizar o acesso às funcionalidades dos dispositivos, seja para extrair informações de interesse, interagir com outros dispositivos ou acessar funções externas. Esta camada pode ser subdividida em três componentes principais que são o `Gerenciador de Sensores`, o `Gerenciador de Rede` e o `Gerenciador de Visualização`. O `Gerenciador de Sensores` define uma classe responsável por criar uma instância do serviço de sensor. Essa classe fornece vários métodos para acessar e listar sensores, fornecendo um conjunto de funções que executam as operações necessárias para obter dados, como

velocidade, sentido, localização. Já o `Gerenciador de Rede` fornece um conjunto de funções para criações de conexões e transferência de dados entre os dispositivos utilizando tecnologias de redeS sem fio como Wi-Fi, 3G/4G e *Bluetooth*. Por último, o `Gerenciador de Visualização` é utilizado para fornecer suporte a visualização de informações em interfaces gráficas, como por exemplo, em mapas geográficos.

4.1.2 Camada de Interfaces de Desenvolvimento

Apesar dos dispositivos já oferecerem interfaces de programação a partir do sistema operacional móvel do Android ou de *drivers*, o programador ao criar aplicações específicas no contexto de redes móveis poderá encontrar dificuldades durante o desenvolvimento de aplicações. Essas dificuldades são resultado do grande número de funções oferecidas pelas API's do Android para a criação de aplicações móveis. Repare que para quem não está ambientado em plataformas de desenvolvimento para programação de dispositivos móveis, descobrir quais funções são necessárias e importantes para cada uma das possíveis aplicações pode resultar em um tempo grande de estudo e possíveis frustações.

Como o objetivo da biblioteca é oferecer funcionalidades comuns entre um conjunto de aplicações, nota-se uma oportunidade de oferecer essas facilidades para quem deseja desenvolver esse tipo de aplicação. Pensando nisso, foi proposta uma camada superior à de Interfaces de Dispositivo, chamada Interfaces de Desenvolvimento, para que todas essas particularidades sejam encapsuladas, oferecendo funções de mais alto nível para o desenvolvimento de aplicações no contexto de redes móveis. O encapsulamento de procedimentos utilizados com frequêcia em métodos de mais alto nível torna ao mesmo tempo a programação mais simples e as aplicações menos vulneráveis a alterações nas interfaces de programação de nível mais baixo.

Quando ocorre alguma atualização das API's do Android e se alguma funcionalidade precisa ser reescrita ou atualizada, essas atualizações serão realizadas nesta camada. Uma das vantagens é que esse processo é transparente para a camada de aplicação que será tratada posteriormente, pois não afeta de imediato no funcionamento da aplicação. As atualizações só terão efeito depois de compilar novamente o ComuniCAR e disponibilizá-lo para a aplicação. Como o mesmo é uma biblioteca estática, o programador poderia atualizar as API's, verificar se existe alguma incompatibilidade com as funcionalidades já criadas e se for o caso, realizar as atualizações e testes nos códigos desenvolvidos antes de compilar e disponibilizá-lo para o desenvolvimento de outras aplicações. Isso é importante quando a aplicação já está em ambiente de produção e as atualizações poderiam comprometer no seu funcionamento.

4.1.3 Camada de Aplicação

Esta camada representa o desenvolvimento das aplicações de redes móveis por terceiros que utiliza as funcionalidades da camada inferior, Interface de Desenvolvimento. Um dos grandes desafios em desenvolver aplicações para o Android é acompanhar a evolução desenfreada da plataforma. E a cada novidade, surgem mais API's, novos arcabouços, novos *widgets*, entre outros. Porém, as atualizações podem comprometer o funcionamento das aplicações surpreendendo os desenvolvedores. Esse problema pode ocorrer devido as novidades que a cada atualização pode conter. Por exemplo, a partir do Android 6.0 (API 23) os aplicativos precisam pedir permissões em tempo de execução para utilizar determinadas API's, como por exemplo: localização, câmera, escrever no *sdcard*, entre outros [14].

4.2 Implementação do ComuniCAR

O ComuniCAR foi desenvolvido para oferecer funções de alto nível para o desenvolvimento de aplicações, propósito da Camada de Interfaces de Desenvolvimento, e nas possíveis facilidades a serem inseridas na Camada de Aplicação.

O Android fornece os componentes de Gerenciamento de Visualização, Sensores e Rede utilizados. O Gerenciamento de Visualização dispõe de uma interface de programação para o Google Maps, o Gerenciamento de Sensores dispõe de interfaces com os sensores dos dispositivos e porta USB e o Gerenciamento de Rede possui interfaces para comunicações através de tecnologias de rede como o WiFi Direct, utilizado nesta implementação.

Ainda, a partir do uso do Android, é possível implementar as funções da Camada de Interfaces de Desenvolvimento usando um ambiente de programação específico. Tal ambiente facilita a programação, pois já possui as funções dos dispositivos embutidas. A seguir, a implementação da camada de Interfaces de Desenvolvimento é descrita apresentando o diagrama de classe da 4.2, e em seguida, as funções criadas para esta camada são listadas e apresentadas. A camada de Aplicação define o caso de uso que é o objeto de estudo nesta dissertação e, portanto, é apresentada no Capítulo 5 seguinte.

4.2.1 Implementação da Camada de Interfaces de Dispositivo

A implementação da Camada de Interfaces de Dispositivos do ComuniCAR envolve basicamente o Android.

4.2.2 Implementação da Camada de Interfaces de Desenvolvimento

A implementação da camada de Interfaces de Desenvolvimento implicou a criação de funções de mais alto nível, levando em conta as funções disponíveis através da API do Android.

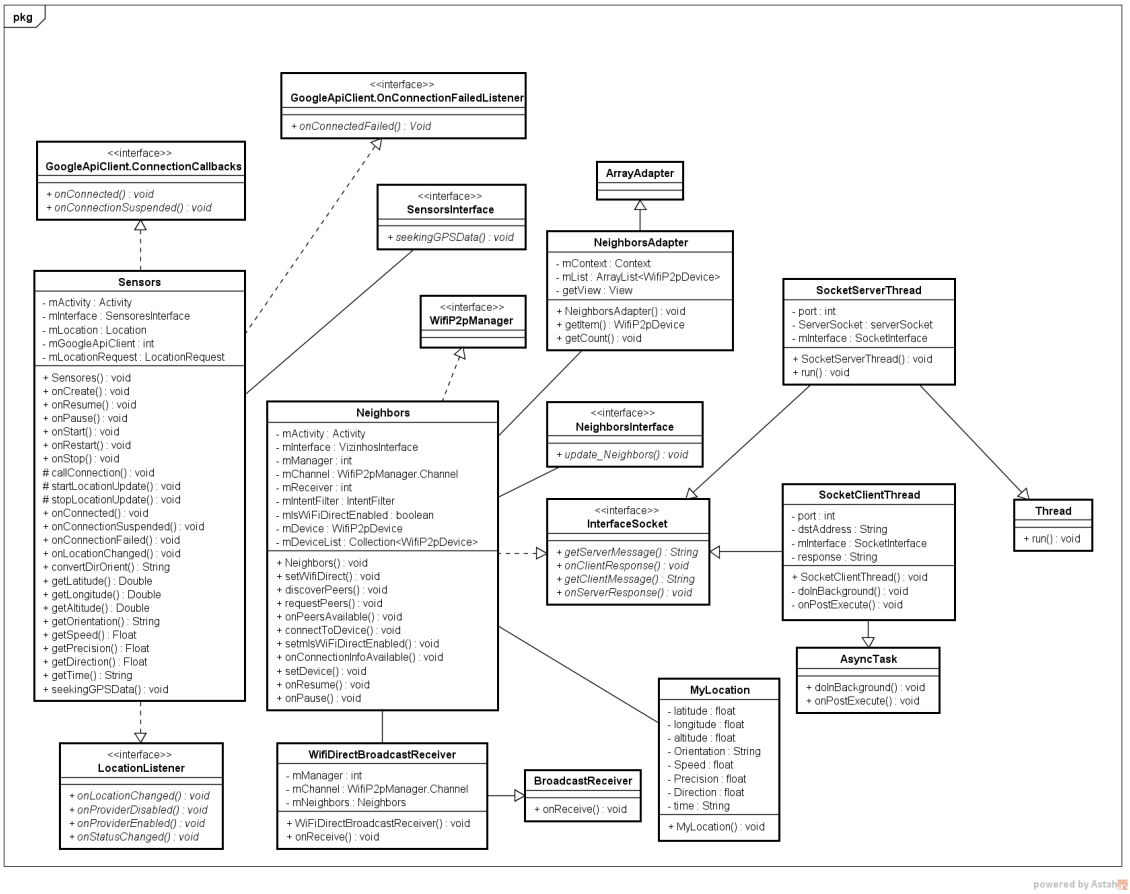

Figura 4.2: Diagrama de classe da biblioteca.

Serviço de localização

Normalmente, a maioria dos dispositivos *Android* permitem determinar a sua localização geográfica atual. Isto pode ser realizado através da utilização de um módulo GPS, normalmente interno, rede GSM ou através de redes Wi-Fi [2].

O Google Play Services facilita a conexão com os serviços do Google, além de várias API's como localização, Google Fit, Google+. GoogleDrive, Games, entre outros [2]. A API de localização que foi utilizada nesta dissertação é conhecida como *Fused Location Provider*. O Google Play Services precisou ser declarado como dependência no projeto localizado no ficheiro *AndroidManifest.xml*.

Código 1. Trecho de código para declaração de dependências no projeto

```
dependencies {
    compile 'com.google.android.gms:play-services:8.4.0'
}
```

Foi desenvolvida uma classe chamada de *Sensors* que é responsável por facilitar o desenvolvimento de uma aplicação que necessita dos dados de localização do GPS em tempo real. Abaixo, segue a explicação de todos os métodos que foram implementados nesta classe.

Para utilizar qualquer uma das API's gerenciadas pelo Google Play Services, foi necessário se conectar aos serviços do Google. A classe responsável por fazer esta conexão é a *GoogleApiClient*, que encapsula toda a comunicação com os servidores do Google. O Código 2 apresenta o método *void callConnection()* que foi implementado para construir um objeto do tipo *GoogleApiClient*.

Código 2. Trecho de código utilizado para conectar com os servidores do Google

```

public GoogleApiClient mGoogleApiClient;

protected synchronized void callConnection() {
        Log.i("LOG", "callConnection()");
        mGoogleApiClient = new GoogleApiClient.Builder(mActivity)
            .addOnConnectionFailedListener(this)
            .addConnectionCallbacks(this)
            .addApi(LocationServices.API)
            .build();
}
```

Depois de construir o objeto *GoogleApiClient* no método *callConnection()*, o método *onCreate()* da *Activity* é responsável por chamar o método *callConnection()*.

Código 3. Trecho de código que invoca o método callConnection()

```
    public void onCreate() {
        callConnection();
    }
```

Depois dessa etapa, foi chamado o método *GoogleApiClient.connect()* que é responsável para realizar a conexão no método *onStart()* e *onRestart()* da *activity* e fechar a conexão no método *onStop()*, conforme demonstrado no Código 4.

Código 4. Trecho de código que invoca o método GoogleApiClient.connect()

```
    public void onStart() {
        mGoogleApiClient.connect();
    }

    public void onRestart() {
        mGoogleApiClient.connect();
```

```
7       }
8
9       public void onStop() {
10          stopLocationUpdate();
11          mGoogleApiClient.disconnect();
12      }
```

Para receber o resultado da conexão, sendo com sucesso ou falha, foi implementado os métodos das interfaces *ConnectionCallbacks* e *OnConnectionFailedListener*.

Código 5. Métodos da ConnectionCallbacks e OnConnectionFailedListener

```
1       public void onConnected(Bundle bundle) {
2         // Conectado ao Google Play Services!
3           // Pode utilizar qualquer API agora.
4           startLocationUpdate(); // Inicia o GPS
5       }
6
7       public void onConnectionSuspended(int i) {
8       // A conexao foi interrompida.
9       // A aplicacao precisa aguardar ate a conexao ser restabelecida
10          Toast.makeText(mActivity.getApplicationContext(), "Conexao
                interrompida!", Toast.LENGTH_SHORT).show();
11      }
12
13      public void onConnectionFailed(ConnectionResult connectionResult) {
14      // Erro na conexao.
15      // Pode ser uma configuracao invalida ou falta de conectividade no
            dispositivo.
16          Toast.makeText(mActivity.getApplicationContext(), "Erro ao
                conectar: " + connectionResult, Toast.LENGTH_SHORT).show();
17      }
```

Depois que a aplicação está conectada aos serviços do Google, foi implementado o método *startLocationUpdate()*. Este método foi criado para configurar e iniciar o GPS. Neste método foi criado um ojeto *LocationRequest*, que contém as configurações referentes a precisão e ao intervalo de tempo com que o utilizador deseja receber as coordenadas. O método *setPriority(int)* define a precisão do GPS. As constantes mais utilizadas são *PRIORITY HIGH ACCURACY* e *PRIORITY LOW POWER*. A primeira tem como objetivo retornar a localização mais precisa possível. A segunda visa obter uma localização aproximada a fim de economizar a bateria. O método setInterval(long) recebe o intervalo de tempo em milissegundos em que a aplicação deseja receber atualizações do GPS. O método setFastestInterval(long) recebe o intervalo mínimo no qual a aplicação consegue receber e tratar os eventos corretamente, sem afetar a performance da aplicação. Outro método importante é O *FusedLocationApi* que foi utilizado para recuperar a última localização conhecida

pelo sistema.

Código 6. Implementação do método startLocationUpdate()

```
1     protected void startLocationUpdate() {
2         mLocationRequest = new LocationRequest();
3         mLocationRequest.setInterval(0);
4         mLocationRequest.setFastestInterval(0);
5         mLocationRequest.setPriority(LocationRequest.
              PRIORITY_HIGH_ACCURACY);
6         //region foo
7         if (ActivityCompat.checkSelfPermission(mActivity, Manifest.
              permission.ACCESS_FINE_LOCATION)
8                 != PackageManager.PERMISSION_GRANTED &&
9                 ActivityCompat.checkSelfPermission(mActivity, Manifest.
                      permission.ACCESS_COARSE_LOCATION)
10                        != PackageManager.PERMISSION_GRANTED) {
11            return;
12        }
13        LocationServices.FusedLocationApi.requestLocationUpdates(
              mGoogleApiClient, mLocationRequest, Sensores.this);
14    }
```

O código 7 abaixo apresenta o método *stopLocationUpdate()* que é responsável por parar o GPS.

Código 7. Implementação do método stopLocationUpdate()

```
1 protected void stopLocationUpdate(){
2 LocationServices.FusedLocationApi.removeLocationUpdates(
      mGoogleApiClient, Sensores.this);
3 }
```

Segundo Lecheta[2], o momento correto de iniciar o GPS é logo depois de a conexão com o Google Play Services é realizada, portanto isso deve ser feito no método *onConnected()* citado anteriormente.

Para parar o GPS, pode-se utilizar os métodos do ciclo de vida da *activity*, como o *onPause()* ou *onStop()*.

Código 8. Implementação dos métodos onPause() e onStop()

```
1     public void onPause() {
2         if (mGoogleApiClient != null) {
3             stopLocationUpdate(); // Para o GPS
4         }
5     }
6
7     public void onStop() {
8         stopLocationUpdate(); // Para o GPS
9         mGoogleApiClient.disconnect(); // Desconexao da API
```

```
10      }
```

Foi implementado também o método *onLocationChanged(location)* que é responsável por receber as localizações provenientes do GPS. Sempre que o Android tiver uma nova localização, este método será invocado passando como parâmetro um objeto *android.location.Location*. Este método pertence a interface *LocationListener* e está representado no Código 9.

Código 9. Implementação do método onLocationChanged(location)()

```
1 public void onLocationChanged(Location location) {
2         mLocation = location;
3         if(mInterface!=null)
4             mInterface.seekingGPSData(location);
5     }
```

Uma localização pode ser constituída pela latitude, longitude, data e hora, e outras informações tais como altitude e velocidade. Para obter esses dados foi necessário utilizar à classe *Location* que é uma classe de dados que representa uma localização geográfica [14]. Esta classe possui alguns métodos como:

- `getAccuracy()` - Devolve a precisão estimada de localização, em metros.
- `getLatitude()` - Devolve a Latitude, em graus.
- `getLatitude()` - Devolve a Longitude, em graus.
- `getAltitude()` - Devolve a altitude, em metros.
- `getBearing()` - Devolve a direção (sentido) do dispositivo, em graus.
- `getSpeed()` - Velocidade em metros por segundo;
- `getTime()` - Tempo Universal Coordenado da localização, em milisegundos desde 01/01/1970.

No código 10, são apresentadas as implementações das funções que chamam os métodos da classe *Location*.

Código 10. Implementação dos métodos da classe Location

```
1     public double getLatitude() {
2         return mLocation==null?null:mLocation.getLatitude();
3     }
4
5     public double getLongitude() {
6         return mLocation==null?null:mLocation.getLongitude();
7     }
8
```

```
9       public double getAltitude() {
10          return mLocation==null?null:mLocation.getAltitude();
11      }
12
13      public String getOrientation() {
14          return convertDirOrient(mLocation);
15      }
16
17      public float getSpeed() {
18          return mLocation==null?null:mLocation.getSpeed()*(float)3.6;
19      }
20
21      public float getPrecision() {
22          return mLocation==null?null:mLocation.getAccuracy();
23      }
24
25      public float getDirection() {
26          return mLocation==null?null:mLocation.getBearing();
27      }
28
29      public String getTime() {
30          return mLocation==null?null:new SimpleDateFormat("hh:mm:ss").
                format(new Date(mLocation.getTime()));
31      }
```

Foi implementada também uma nova função que não pertence a classe *Location*, chamada de *getOrientation()*. Esta função realiza uma chamada para outra função *convertDirOrient(location)* que recebe como parâmetro um objeto *android.location.Location* e realiza uma conversão do sentido em graus para as siglas dos pontos cardeais (Norte, Sul, Leste e Oeste), colaterais (nordeste, sudeste, noroeste e sudoeste) e subcolaterais (norte-nordeste, norte-noroeste, leste-nordeste, leste-sudeste, sul-sudeste, sul-sudoeste, oeste-sudoeste e oeste-noroeste).

getOrientation()

Código 11. Interface SensorsInterface

```
1  public String convertDirOrient(Location location) {
2          mLocation = location;
3          String orientacao = "Nao obtida";
4          if (mLocation == null){
5              orientacao = "null";
6          }
7          else if (mLocation.hasBearing() != false) {
8              float sentido = mLocation.getBearing();
9              if (sentido >= 0 && sentido <= 10) {
10                     orientacao = "N";
11             } else if (sentido >= 11 && sentido <= 33) {
```

```
12                  orientacao = "N-NE";
13              } else if (sentido >= 34 && sentido <= 55) {
14                  orientacao = "NE";
15              } else if (sentido >= 56 && sentido <= 78) {
16                  orientacao = "E-NE";
17              } else if (sentido >= 79 && sentido <= 100) {
18                  orientacao = "E";
19              } else if (sentido >= 101 && sentido <= 123) {
20                  orientacao = "E-SE";
21              } else if (sentido >= 124 && sentido <= 145) {
22                  orientacao = "SE";
23              } else if (sentido >= 146 && sentido <= 168) {
24                  orientacao = "S-SE";
25              } else if (sentido >= 169 && sentido <= 190) {
26                  orientacao = "S";
27              } else if (sentido >= 191 && sentido <= 213) {
28                  orientacao = "S-SO";
29              } else if (sentido >= 214 && sentido <= 235) {
30                  orientacao = "SO";
31              } else if (sentido >= 236 && sentido <= 258) {
32                  orientacao = "O-SO";
33              } else if (sentido >= 259 && sentido <= 280) {
34                  orientacao = "O";
35              } else if (sentido >= 281 && sentido <= 303) {
36                  orientacao = "O-NO";
37              } else if (sentido >= 304 && sentido <= 325) {
38                  orientacao = "NO";
39              } else if (sentido >= 326 && sentido <= 348) {
40                  orientacao = "N-NO";
41              } else if (sentido >= 349 && sentido <= 359) {
42                  orientacao = "N";
43              }
44          }
45          return orientacao;
46      }
```

Para utilizar o GPS na aplicação é necessário declarar algumas permissões necessárias no ficheiro *AndroidManifest.xml* que podem ser:

- `ACCESS FINE LOCATION` - Permite a aplicação receber à localização através de GPS por hardware.
- `ACESS COARSE LOCATION` - Permite a aplicação receber à localização através do Wi-Fi ou triangulação de antenas.

A permissão a ser escolhida vai depender das configurações de precisão feitas no objeto *LocationRequest*. O código 10 apresenta o trecho de código utilizado no ficheiro *AndroidManifest.xml*.

Código 12. Declaração das permissões no ficheiro AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="gta.ComuniCAR">
<uses-permission android:name="android.permission.ACCESS_GPS" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"
    />
```

Depois de implementar a classe *Sensors*, foi desevolvida uma interface chamada de *SensorsInterface*. Esta interface implementa o método *seekingGPSData()*, que é chamado dentro do método *onLocationChanged(location)* da classe *Sensors*.

Código 13. Interface SensorsInterface

```
public interface SensorsInterface {
    void seekingGPSData(Location newLocation);
}
```

Repare que ao utilizar esta interface o utilizador irá implementar somente o método *seekingGPSData()* para obter os dados de localização do GPS, pois todos os outros métodos já estão encapsulados na classe Sensors. O código 10 apresenta o uso da biblioteca ComuniCAR e a implementação do método *seekingGPSData* da interface *SensorsInterface*.

Código 14. Obtendo os dados do GPS utilizando o ComuniCAR

```
import gta.ComuniCAR.Sensors;
import gta.ComuniCAR.SensorsInterface;

public class ApSensores extends Activity implements SensorsInterface {

Sensores mSensores; // Cria uma nova instancia de Sensores

public void seekingGPSData(Location newLocation) {
        txtLatitude.setText(String.valueOf(mSensores.getLatitude()));
        txtLongitude.setText(String.valueOf(mSensores.getLongitude()));
        txtAltitude.setText(String.valueOf(mSensores.getAltitude()));
        txtVelocidade.setText(String.valueOf(mSensores.getSpeed()));
        txtOrientacao.setText(mSensores.getOrientation());
        txtPrecisao.setText(String.valueOf(mSensores.getPrecision()));
        txtHora.setText(mSensores.getTime());
        txtDirecao.setText(String.valueOf(mSensores.getDirection()));
}
}
```

Mecanismo de comunicação

Para a construção do ComuniCAR foi utilizado a *API* do *Wi-Fi Direct* disponibilizado pelo *SDK Android*. Esta API permite ao aplicativo, instalado no sis-

tema operacional Android 4.0 ou versões posteriores com hardware apropriado se conectar diretamente com outros dispositivos via Wi-Fi, sem um ponto de acesso intermediário.

Código 15. Método Construtor da classe Neighbors

```
1 public Neighbors(Activity mActivity, VizinhosInterface mInterface){
2         this.mActivity = mActivity;
3         this.mInterface = mInterface;
4         setWifiDirect(); // chama o metodo setWifiDirect()
5 }
6 }
```

No método *setWifiDirect()* foi realizado uma instância do *WifiP2pManager* que irá ser utilizado para registrar a aplicação ao *Wi-Fi Direct*, através do método *initialize()*. Este método retorna um *WifiP2pManager.Channel*, que foi utilizado para ligar a aplicação com o *Wi-Fi Direct*. Foi criado também o *WiFiDirectBroadcastReceiver*, onde foi registrado os seus respectivos *Intents* como exemplo: *WIFI P2P STATE CHANGED ACTION* de forma a permitir que a aplicação receba notificações de eventos do seu interesse, como por exemplo: saber se o *Wi-Fi Direct* dos dispositivos encontram-se ligados/desligados ou se um dispositivo perdeu uma conexão Wi-Fi Direct.

Código 16. Obtendo os dados do GPS utilizando o ComuniCAR

```
1
2 public class Vizinhos implements
3         WifiP2pManager.PeerListListener,
4         WifiP2pManager.ConnectionInfoListener , SocketInterface {
5
6 Activity mActivity;
7 WifiP2pManager mManager;
8 WifiP2pManager.Channel mChannel;
9 BroadcastReceiver mReceiver;
10 IntentFilter mIntentFilter;
11 WifiP2pDevice mDevice;
12
13 public void setWifiDirect() {
14         mManager = (WifiP2pManager) mActivity.getSystemService(Context.
               WIFI_P2P_SERVICE);
15         mChannel = mManager.initialize(mActivity, mActivity.
               getMainLooper(), null);
16         mReceiver = new WiFiDirectBroadcastReceiver(mManager, mChannel,
                mActivity, this);
17         mIntentFilter = new IntentFilter();
18         mIntentFilter.addAction(WifiP2pManager.
               WIFI_P2P_STATE_CHANGED_ACTION);
```

```
19          mIntentFilter.addAction(WifiP2pManager.
                WIFI_P2P_PEERS_CHANGED_ACTION);
20          mIntentFilter.addAction(WifiP2pManager.
                WIFI_P2P_CONNECTION_CHANGED_ACTION);
21          mIntentFilter.addAction(WifiP2pManager.
                WIFI_P2P_THIS_DEVICE_CHANGED_ACTION);
22  }
23  }
```

Tendo os dois objetos criados do *WifiP2pManager* e *WifiP2pManager.Channel*, as aplicações já podem chamar os métodos da classe *WifiP2pManager* para utilizarem as funcionalidades fornecidas pelo *Wi-Fi Direct*. Foi utilizado o BroadcastReceiver é utilizado para a aplicação receber notificações de eventos importantes, como por exemplo: saber se o Wi-Fi está ligado ou não, se o dispositivo perdeu a conexão a uma rede ou conectou-se a uma rede, entre outras notificações [14].

Utilizando as API's *Wi-Fi Direct* do SDK Android, existe duas formas de se conectar um dispositivo P2P Group. Estes são realizadas através da chamada aos métodos *connect()* e *createGroup()* da classe *WifiP2pManager* [14].

No método *createGroup()*, será realizado a técnica de formação de grupo *Autonomous*, portanto o dispositivo irá criar um *P2P Group* onde ele é *P2P GO*. E através do método *connect()* um dispositivo irá conectar-se a um outro dispositivo previamente descoberto no processo de descoberta dos vizinhos. Para chamar o método *connect()* é necessário passar como parâmetro o objeto *WifiP2pConfig* que contém a configuração para estabelecer a conexão com o dispositivo desejado.

Código 17. Implementação do método onConnectionInfoAvailable()

```
1  public void onConnectionInfoAvailable(WifiP2pInfo info) {
2          mInfo = info;
3          if (info.isGroupOwner) {
4              t("Owner");
5              if(mServerThread!=null && mServerThread.isAlive()){
6                  mServerThread.interrupt();
7              }
8              if(mClientTimer!=null){
9                  mClientTimer.cancel();
10                 mClientTimer.purge();
11             }
12             mServerThread = new SocketServerThread(this);
13             mServerThread.start();
14         } else {
15             t("member");
16             if(mServerThread!=null && mServerThread.isAlive()){
17                 mServerThread.interrupt();
18             }
19             if(mClientTimer!=null){
```

```
20                  mClientTimer.cancel();
21                  mClientTimer.purge();
22              }
23              mClientTimer = new Timer();
24              mClientTimer.scheduleAtFixedRate(new TimerTask() {
25                  @Override
26                  public void run() {
27                      if ((mClientSocket == null || mClientSocket.
                            getStatus() != SocketClientThread.Status.
                            RUNNING) && !mInfo.isGroupOwner && mInfo.
                            groupFormed) {
28                          mClientSocket = new SocketClientThread(mInfo.
                                groupOwnerAddress.getHostAddress().toString
                                (), Vizinhos.this);
29                          mClientSocket.execute();
30                      }
31                  }
32              }, postTime,postTime);
33          }
34
35          Toast.makeText(mActivity,"Seu status de conexao mudou",Toast.
                LENGTH_LONG).show();
36      }
```

Foram implementados funções e métodos para trocar mensagens dos dados de localização entre os dispositivos.

Código 18. Implementação dos métodos para troca de mensagens

```
1  public String getServerMessage() {
2          return mGson.toJson(new MyLocation(mLocation));
3      }
4
5      public void onClientResponse(String reponse) {
6          t("CL "+reponse);
7      }
8
9      public String getClientMessage() {
10         return mGson.toJson(new MyLocation(mLocation));
11     }
12
13     public void onSeverResponse(String response) {
14         t("SV "+response);
15     }
```

Na conexão através da chamada ao método *connect()*, os dispositivos irão formar um *P2P Group*, negociando entre si qual deles é que irá realizar o papel de *P2P GO*, caso o dispositivo que irá realizar *connect()*, já se encontra num *P2P Group* e este

for *P2P GO*, a função *connect()* efetua um convite ao dispositivo que este deseja conectar, para se juntar ao grupo. Um dispositivo que não se encontra num *P2P GO*, juntando assim ao seu *P2P Group*, sendo irrefutável que o dispositivo terá que descobrir o dispositivo *P2P GO* antes de executar a função *connect()*.

Segundo a especificação *Wi-Fi Direct*, para realizar uma conexão com um outro dispositivo *Wi-Fi Direct*, é preciso que este seja antes descoberto. Nas API's *Wi-Fi Direct* do *sdk Android*, a descoberta dos pares pode ser realizada através da chamada do método *discoverPeers()* da classe *WifiP2pManager*.

Código 19. Método discoverPeers()

```
1  public void discoverPeers() {
2          mManager.discoverPeers(mChannel, new WifiP2pManager.
               ActionListener() {
3              @Override
4              public void onSuccess() {
5              }
6
7              @Override
8              public void onFailure(int reasonCode) {
9              }
10         });
11     }
```

Depois de implementar a classe *Neighbors*, foi desevolvida uma interface chamada de *NeighborsInterface*. Esta interface implementa o método *onNeighborsUpdate()*,

Código 20. Interface NeighborsInterface

```
1  public interface NeighborsInterface {
2      void onNeighborsUpdate(Collection<WifiP2pDevice> newList);
3  }
```

Repare que ao utilizar a interface *NeighborsInterface* o utilizador irá implementar somente o método *onNeighborsUpdate()* para obter a lista dos vizinhos encontrados, pois todos os outros métodos já estão encapsulados na classe Neighbors. O código 10 apresenta o uso da biblioteca ComuniCAR e a implementação do método *seekingGPSData* da interface *SensorsInterface*.

Código 21. Obtendo a lista de vizinhos em um dado momento

```
1
2  import gta.ComuniCAR.Sensors;
3  import gta.ComuniCAR.SensorsInterface;
4  import gta.ComuniCAR.Neighbors;
5  import gta.ComuniCAR.NeighborsAdapter;
6  import gta.ComuniCAR.NeighborsInterface;
7
```

```
8  public class Neighbors extends Activity implements NeighborsInterface,
       SensorsInterface {
9  public void onNeighborsUpdate(Collection<WifiP2pDevice> newList) {
10         mAdapter = new VizinhosAdapter(this, new ArrayList<
               WifiP2pDevice>(newList));
11         mListView.setAdapter(mAdapter);
12 }
```

Capítulo 5

Estudo de Caso

Neste capítulo é apresentado um exemplo de aplicativo denominado SmartCAR, o qual foi desenvolvido utilizando os recursos da biblioteca ComuniCAR. A implementação dessa aplicação possibilita que se avalie os ganhos trazidos pela biblioteca tanto no processo de desenvolvimento quanto na geração de cenários para a avaliação em redes móveis.

5.1 SmartCAR

Antes de apresentar a aplicação SmartCAR, será discutido como é realizada a importação da biblioteca ComuniCAR no Android Studio. Para utilizar esta biblioteca, o programador precisa importá-la para a sua aplicação. Existem duas formas de realizar esse procedimento no Android Studio. Na primeira forma é o programador adicionar o ComuniCAR diretamente no arquivo "build.gradle', apresentado no código 4, sendo que primeiramente ele deverá possuí-lo baixado em sua máquina [21].

Código 4. Trecho de código que mostra a biblioteca sendo utilizada na aplicação

```
1  apply plugin: 'com.android.application'
2
3  repositories {
4      flatDir {
5          dirs '../libraries'
6      }
7  }
8
9  dependencies {
10     compile 'gta.android:ComuniCAR:0.0.1'
11 }
```

Na segunda forma utiliza-se a interface provida pela IDE com o plugin do Gradle. Para isso o programador deverá clicar com o botão direito no módulo *app* e esco-

lher a opção *Open Module Settings*. Logo depois, será aberta uma janela contendo algumas abas de opções que o programador pode altera nos arquivos do *Gradle*. Para adicionar dependências, o programador deverá clicar na aba *Dependencies* e depois no botão "+". Posteriormente será apresentado três opções de dependências: *Library Dependency*, *File Dependency* e *Module Dependency*. Nesta dissertação, foi escolhido a opção *Module Dependency*, pois o ComuniCAR não é uma biblioteca nativa do Android. A figura **??** apresenta o ComuniCAR adicionado nas dependências do projeto.

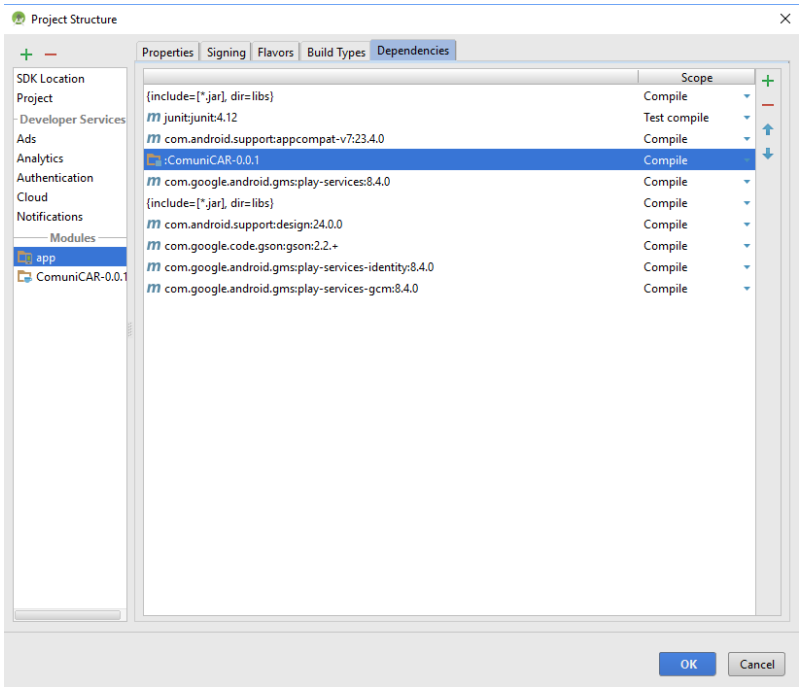

Figura 5.1: Janela de dependências da aplicação.

O SmartCAR é um aplicativo Android que foi desenvolvido utilizando os recursos do ComuniCAR demonstrando algumas aplicações que poderiam ser desenvolvidas na área de redes móveis. As aplicações são: (1) Obtenção de dados dos sensores para a troca de informações entre os dispositivos, (2) Detecção e conexão de vizinhos encontrados em um dado momento utilizando o *Wi-Fi Direct*. Ao estabelecer a conexão, os dados de localização entre os vizinhos são compartilhados entre eles. O SmartCAR foi implementado em Java e XML (*eXtensible Markup Language*) através da IDE Android Studio. A linguagem Java foi utilizada na elaboração das *Activities*. As *Activities* são as classes Java onde estão implementadas as telas das aplicações. São nessas *Activities* que são acionadas as rotinas da biblioteca desenvolvida do ComuniCAR de acordo com as funcionalidades acessadas pelo usuário. O XML por sua vez, foi empregado no posicionamento de elementos como menus, botões e imagens na tela do aplicativo. O SmartCAR funciona em dispositivos móveis com Android 4.0 ou superior. Quando o SmartCAR é executado pela primeira vez, é

apresentado um menu de opções conforme a Figura 5.2. As opções são: Sensores, Vizinhos e Sair. A seguir, cada uma dessas opções é explicada:

- Sensores: Oferece dados dos sensores como: Latitude, Longitude, Altitude, Velocidade em km/h, Orientação, Precisão e Direção.
- Vizinhos: Apresenta os vizinhos que estão no entorno em um dado momento.
- Sair: Sair da aplicação.

Figura 5.2: Tela do menu do SmartCAR.

5.2 Avaliação das Aplicações do SmartCAR

Nesta seção, foram avaliadas as aplicações que obtém os dados de localização provenientes do GPS em tempo real e a descoberta e conexão dos vizinhos em um dado momento utilizando o *Wi-Fi Direct*. Os resultados apresentados a seguir, são inerentes a economia de código utilizado para desenvolver tais aplicações.

5.2.1 Obtendo os dados de localização

Para obter os dados de localização provenientes do GPS em tempo real, foi criado uma classe chamada de *ApSensors* que herdou da classe *Activity* e implementou a interface *SensorsInterface* citados no 4 anterior. O código 22 abaixo, apresenta a aplicação desenvolvida:

Código 22. Aplicação que obtém os dados de localização pelo GPS

```
1  package gta.smartcar;
2  import android.app.Activity;
3  import android.location.Location;
4  import android.os.Bundle;
5  import android.view.View;
6  import android.widget.Button;
7  import android.widget.TextView;
8
9  import gta.ComuniCAR.Sensors;
10 import gta.ComuniCAR.SensorsInterface;
11
12 public class ApSensors extends Activity implements SensorsInterface {
13
14     public TextView txtLatitude, txtLongitude, txtAltitude,
           txtVelocidade, txtDirecao, txtHora, txtPrecisao, txtOrientacao;
15     public Button btnVoltar;
16     Sensores mSensors;
17
18     protected void onCreate(Bundle icicle) {
19         super.onCreate(icicle);
20         setContentView(R.layout.apsensors);
21         registerIds();
22         setListeners();
23         setGlobalVariable();
24     }
25
26     void registerIds(){
27         txtLatitude = (TextView) findViewById(R.id.txtLatitude);
28         txtLongitude = (TextView) findViewById(R.id.txtLongitude);
29         txtAltitude = (TextView) findViewById(R.id.txtAltitude);
30         txtVelocidade = (TextView) findViewById(R.id.txtVelocidade);
31         txtOrientacao = (TextView) findViewById(R.id.txtOrientacao);
32         txtHora = (TextView) findViewById(R.id.txtHora);
33         txtPrecisao = (TextView) findViewById(R.id.txtPrecisao);
34         txtDirecao = (TextView) findViewById(R.id.txtDirecao);
35         btnVoltar = (Button) findViewById(R.id.btnVoltar);
36     }
37
38     void setListeners(){
```

```
39          btnVoltar.setOnClickListener(new View.OnClickListener() {
40              @Override
41              public void onClick(View v) {
42                  onBackPressed();
43              }
44          });
45      }
46
47      void setGlobalVariable(){
48          mSensors = new Sensors(this,this);
49          mSensors.onCreate();
50      }
51
52
53      public void onResume() {
54          super.onResume();
55          mSensors.onResume();
56      }
57
58      public void onPause() {
59          mSensors.onPause();
60          super.onPause();
61      }
62
63      public void onStart() {
64          super.onStart();
65          mSensors.onStart();
66      }
67
68      public void onRestart() {
69          super.onRestart();
70          mSensors.onRestart();
71      }
72
73      public void onStop() {
74          mSensors.onStop();
75          super.onStop();
76      }
77
78  public void seekingGPSData(Location newLocation) {
79          txtLatitude.setText(String.valueOf(mSensors.getLatitude()));
80          txtLongitude.setText(String.valueOf(mSensors.getLongitude()));
81          txtAltitude.setText(String.valueOf(mSensors.getAltitude()));
82          txtVelocidade.setText(String.valueOf(mSensors.getSpeed()));
83          txtOrientacao.setText(mSensors.getOrientation());
84          txtPrecisao.setText(String.valueOf(mSensors.getPrecision()));
85          txtHora.setText(mSensors.getTime());
```

```
86            txtDirecao.setText(String.valueOf(mSensors.getDirection()));
87    }
88    }
```

Ao clicar no menu "Sensores" é exibido um conjunto de dados de localização provenientes do GPS nativo do dispositivo baseado em Android que podem ser utilizados como informações básicas para a construção de aplicações para redes móveis que necessitam da localização do dispositivo em tempo real.

A Figura 5.3 demonstra a aquisição desses dados pelo SmartCAR.

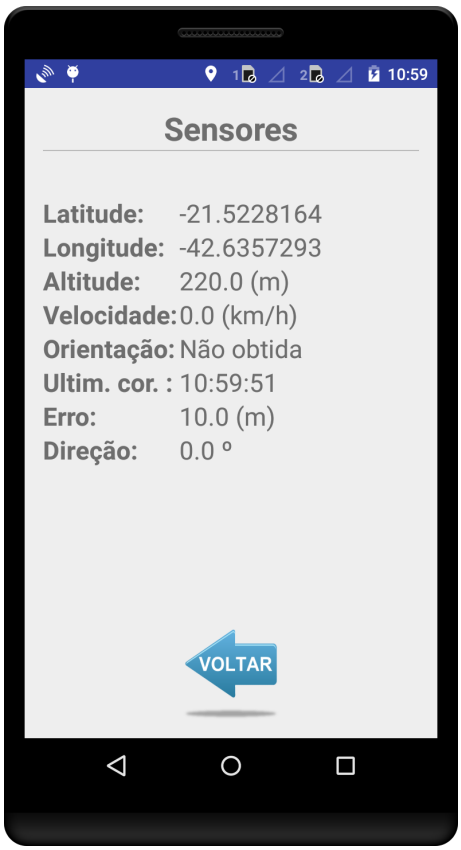

Figura 5.3: Tela para obter dados dos sensores.

A aplicação de obter dados dos sensores utilizando os recursos do ComuniCAR teve uma economia superior a 60% referente ao número de linhas de código em relação a aplicação que utilizou somente as interfaces de programação nativas do Android. Não foi considerado os códigos referentes aos arquivos XML das *activities* para estas métricas em nenhuma das aplicações. O ComuniCAR permite diminuir a quantidade de código concentrado nos métodos das APIs de localização do Android.

5.2.2 Descoberta de vizinhos

Para desenvolver esta aplicação foi criada a classe *ApNeighbors* que herdou da classe *Activity* e implementou as interfaces *NeighborsInterface* e *SensorsInterface*.

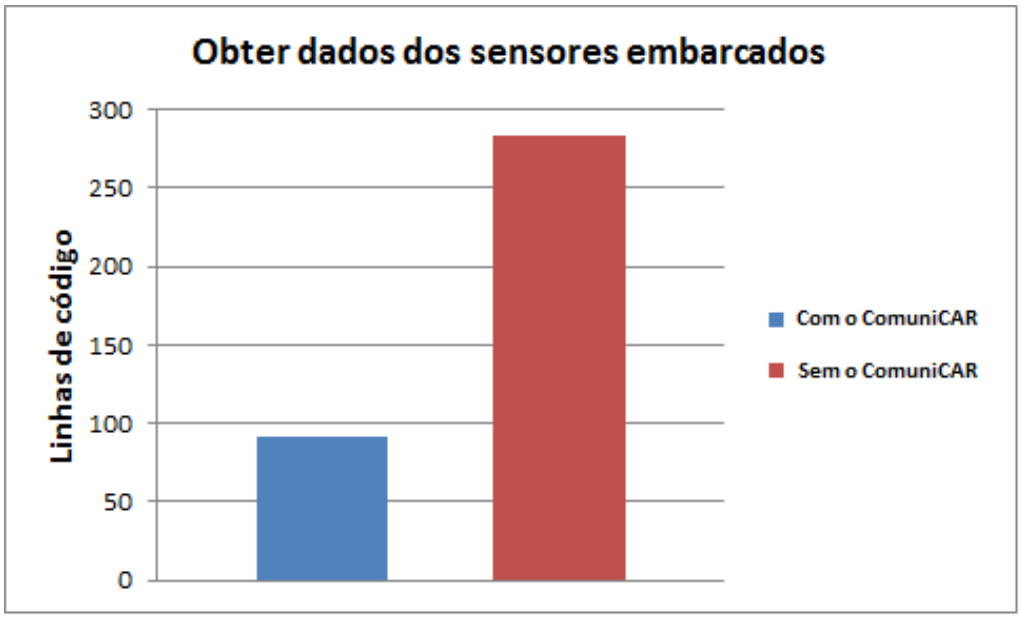

Figura 5.4: Comparação no uso do ComuniCAR.

Esta aplicação esta representada no Código 23.

Código 23. Aplicação que descobre os vizinhos e compartilha os dados do GPS

```
1  package gta.smartcar;
2  import android.app.Activity;
3  import android.location.Location;
4  import android.net.wifi.p2p.WifiP2pDevice;
5  import android.os.Bundle;
6  import android.view.View;
7  import android.widget.AdapterView;
8  import android.widget.Button;
9  import android.widget.ListView;
10
11 import java.util.ArrayList;
12 import java.util.Collection;
13
14 import gta.ComuniCAR.Sensors;
15 import gta.ComuniCAR.SensorsInterface;
16 import gta.ComuniCAR.Neighbors;
17 import gta.ComuniCAR.NeighborsAdapter;
18 import gta.ComuniCAR.NeighborsInterface;
19
20 public class ApVizinhos extends Activity implements VizinhosInterface,
       SensoresInterface {
21
22     ListView mListView;
23     Button btDesc;
24     NeighborsAdapter mAdapter;
25
```

```
26      Neighbors mVizinhos;
27      Sensors mSensores;
28      Location mLocation;
29
30      protected void onCreate(Bundle icicle) {
31          super.onCreate(icicle);
32          setContentView(R.layout.apneighbors);
33          mSensors = new Sensors(this,this);
34          mSensors.onCreate();
35          mNeighbors = new Neighbors(this,this);
36
37          registerIds();
38          setListeners();
39      }
40
41      @Override
42      protected void onResume() {
43          super.onResume();
44          mNeighbors.onResume();
45          mSensors.onResume();
46      }
47
48      @Override
49      protected void onPause() {
50          mNeighbors.onPause();
51          mSensors.onPause();
52          super.onPause();
53      }
54
55      public void onStart() {
56          super.onStart();
57          mSensors.onStart();
58      }
59
60      public void onRestart() {
61          super.onRestart();
62          mSensors.onRestart();
63      }
64
65      public void onStop() {
66          mSensors.onStop();
67          super.onStop();
68      }
69
70      void registerIds(){
71          btDesc = (Button) findViewById(R.id.btDesc);
72          mListView = (ListView) findViewById(R.id.lvDev);
```

```
73      }
74
75      void setListeners(){
76          btDesc.setOnClickListener(new View.OnClickListener() {
77              @Override
78              public void onClick(View v) {
79                  mNeighbors.discoverPeers();
80                  btDesc.setVisibility(View.GONE);
81              }
82          });
83          mListView.setOnItemClickListener(new AdapterView.
                OnItemClickListener() {
84              @Override
85              public void onItemClick(AdapterView<?> parent, View view,
                    int position, long id) {
86                  mNeighbors.connectToDevice(mAdapter.getItem(position));
87              }
88          });
89      }
90
91      @Override
92      public void onNeighborsUpdate(Collection<WifiP2pDevice> newList) {
93          mAdapter = new NeighborsAdapter(this, new ArrayList<
                WifiP2pDevice>(newList));
94          mListView.setAdapter(mAdapter);
95      }
96
97
98      @Override
99      public void seeking_GPS_Data(Location newLocation) {
100         this.mLocation = newLocation;
101         mNeighbors.setLocation(newLocation);
102     }
103 }
```

Ao clicar no menu “Vizinhos” é exibido uma tela contendo um botão chamado ”Descobrir Vizinhos”. Ao clicar nesse botão, é realizada uma busca por vizinhos localizados no entorno do dispositivo. Ao localizar os vizinhos, os mesmos são exibidos na tela contendo informações do nome do dispositivo, id e seu status referente à disponibilidade para conexão. Ao clicar no dispositivo é realizado uma conexão sem fio utilizando o *WiFi Direct* permitindo a troca de informações dos dados dos sensores entre os dispositivos, representado na Figura 5.5.

É importante ressaltar que para a aplicação funcionar corretamente, foi necessário habilitar o Wi-Fi do *smartphone* e não pode estar conectado a nenhum ponto de acesso.

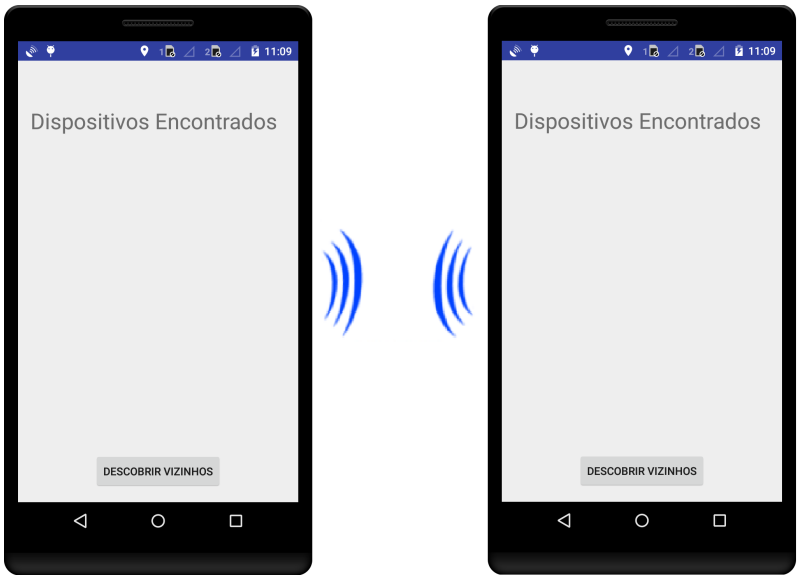

Figura 5.5: Tela de descoberta dos vizinhos em um dado momento.

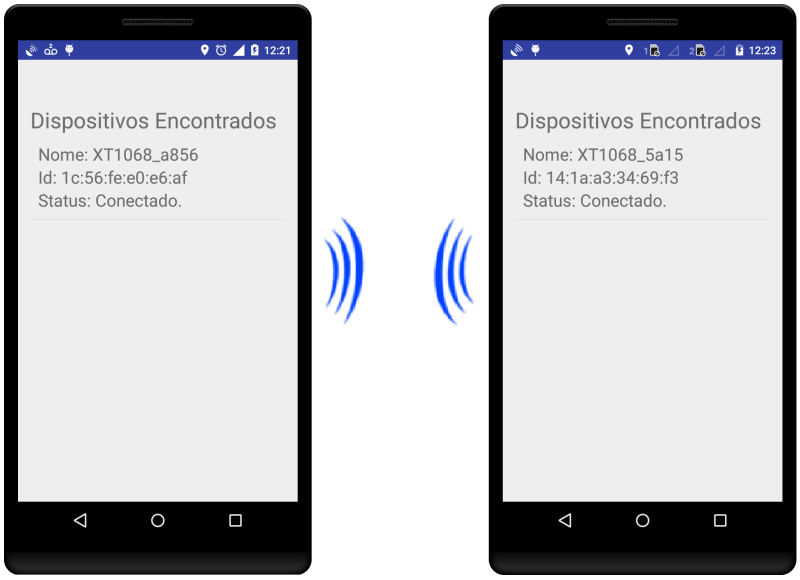

Figura 5.6: Dipositivos encontrados.

A aplicação de descobrir os vizinhos em um dado momento para compartilhar os dados de localização do GPS utilizando os recursos do ComuniCAR teve uma economia superior a 63% referente ao número de linhas de código em relação a aplicação que utilizou somente as interfaces de programação nativas do Android. Não foi considerado os códigos referentes aos arquivos XML das *activities* para estas métricas em nenhuma das aplicações. O ComuniCAR permite diminuir a quantidade de código concentrado nos métodos das API's de comunicação ponto a ponto do Android.

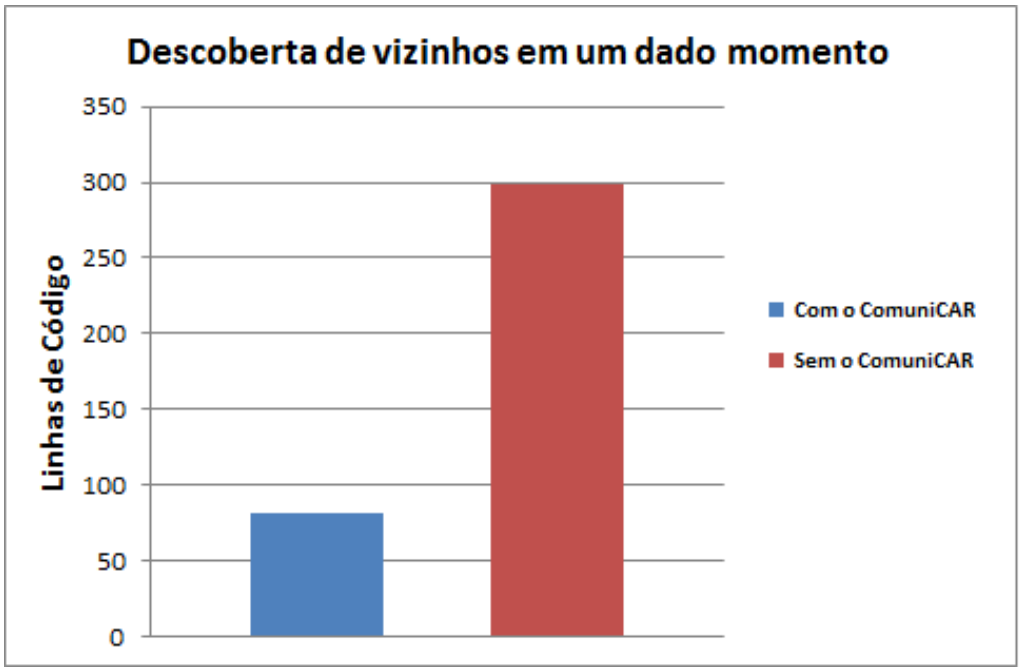

Figura 5.7: Comparação no uso do ComuniCAR.

Capítulo 6

Conclusões e Trabalhos Futuros

Nesta dissertação foi implementado o ComuniCAR, uma biblioteca para facilitar o desenvolvimento de aplicações de redes móveis. Esta biblioteca utilizou o emergente *Wi-Fi Direct* para realizar a conexão entre os dispositivos. Utilizando esta biblioteca no desenvolvimento dos casos de uso apresentados, houve uma redução superior de 60 % no código implementado em relação as API's oferecidas pelo Android.

Esta biblioteca abre várias possibilidades de estudos e pesquisas que podem trazer melhorias e benefícios na área de desenvolvimento de aplicações para redes móveis. Abaixo segue algumas sugestões de trabalhos futuros que podem ser realizados com base nesta dissertação:

- Implementar novas funcionalidades para que o ComuniCAR forneça suporte a redes infraestruturadas.
- Realizar um estudo aprofundado para determinar a escalabilidade do sistema: observar o número de nós que a rede e as sub-redes (grupos) possam realmente suportar utilizando o *WiFi- Direct*.
- Outra pesquisa importante é verificar o tempo de atraso no encaminhamento das mensagens da origem ao destino e até que ponto se pode expandir nesta rede e mantendo ao mesmo tempo uma qualidade aceitável.
- Implementação e validação de novos casos de uso utilizando o ComuniCAR.
- Realizar um estudo aprofundado relativamente a segurança da rede construída utilizando o *WiFi- Direct*.

Referências Bibliográficas

[1] INTERNATIONAL DATA CORPORATION. "Smartphone OS Market Share, 2015 Q2". Disponível em `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`, Acesso em 04 fev. 2016, 2016.

[2] LECHETA R, R. *Google Android: Aprenda a criar aplicações para dispositivos móveis com o Android SDK. 5ª edição.* São Paulo, Novatec, 2015.

[3] PARK, Y., HA, J., KUK, S., et al. "A Feasibility Study and Development Framework Design for Realizing Smartphone-Based Vehicular Networking Systems", *Mobile Computing, IEEE Transactions on*, v. 13, n. 11, pp. 2431–2444, Nov 2014.

[4] OLIVEIRA, F., SARGENTO, S., FERNANDES, J. M., et al. "Reivent: Acessing Vehicular Networks in Mobile Applications", *IEEE Symposium on Computers and Communication (ISCC)*, 2014.

[5] AMBROSIN, M., BUJARI, A., CONTI, M., et al. "Smartphone and Laptop Frameworks for Vehicular Networking Experimentation", *IEEE Conference Publications*, 2013.

[6] BARIS SIMSEK. "Libraries". Disponível em `http://www.enderunix.org/simsek/articles/libraries.pdf`, Acesso em 04 jan. 2016, 2004.

[7] N., G. *Dominando o Android: Do básico ao avançado. 5ª edição.* São Paulo, Novatec, 2015.

[8] CARNEIRO, M., ROMAN, C., FAGUNDEZ, I. "Vendas de smartphones e tablets crescem mais de 1002013". Disponível em Coluna sobre Mercado do Jornal Folha de São Paulo: http://www1.folha.uol.com.br/mercado/2014/01/1391973-vendas-de-smartphones-e-tablets-cresceram-mais-que-100-em-2013.shtm.

[9] CAPUTO, V. "Android e iPhone foram 93,8 2013". Disponível em `http://exame.abril.com.br/tecnologia/noticias/`

android-e-iphone-foram-93-8-dos-aparelhos-vendidos-em-2013, Acessoem08jun.2014 Year = 2013.

[10] APPBRAIN STATS. "Number of Android applications". http://www.appbrain.com/stats/number-of-android-apps.

[11] CHANDNANI, P., WADHVANI, R. "Evolution of Android and its Impact on Mobile Application Development", *International Journal of Scientific Engineering and Technology*, pp. 80–85, 2012.

[12] PEREIRA, L. C. O., DA SILVA, M. L. *Android para Desenvolvedores*. Rio de Janeiro, Brasport, 2009.

[13] DEITEL., P., DEITEL, H., DEITEL, A. *Android para Programadores - Uma abordagem baseada em aplicativos*. Bookman, 2015.

[14] ANDROID DEVELOPERS. "Android Studio Release Notes". Disponível em http://developer.android.com/intl/pt-br/develop/index.html, Acesso em 01 oct. 2015, 2016.

[15] CROCKFORD, D. "The application/json Media Type for JavaScript Object Notation (JSON)". Disponível em http://www.ietf.org/rfc/rfc4627.txt?number=4627, Acesso em 10 oct. 2015, 2006.

[16] JSON. "ECMA-404 The JSON Data Interchange Standard". Disponível em http://json.org, Acesso em 02 oct. 2015, 2016.

[17] WI-FI ALLIANCE. "Wi-Fi CERTIFIED Wi-Fi Direct: Personal, portable Wi-Fi technology". Disponível em https://www.wi-fi.org/download.php?file=/sites/default/files/private/wp_Wi-Fi_CERTIFIED_Wi-Fi_Direct_Industry_20140922_0.pdf, Acesso em 11 oct. 2015, 2014.

[18] CASETTI, C., CHIASSERINI, C., PELLE, L., et al. "Content-centric routing in Wi-Fi direct multi-group networks". In: *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2015 IEEE 16th International Symposium on a*, pp. 1–9, June 2015.

[19] FERNANDES, N. C., MOREIRA, M. D. D., VELLOSO, P. B., et al. "Ataques e Mecanismos de Segurança em Redes Ad Hoc". In: *Minicursos do Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSEG 2006)*, p. 49–102, Agosto 2006.

[20] SHARMA, P. "A Review: Security Issues in Mobile Ad Hoc Network", *International Journal of Computer Science and Mobile Computing*, v. 3, n. 5, pp. 365–370, May 2014.

[21] ORG, G. "Gradle Org". Disponível em `http://http://gradle.org/`, Acesso em 09 mai. 2016, 2016.