ComuniCAR: Uma Biblioteca de Auxílio ao Desenvolvimento de Aplicações para Redes Móveis

Eduardo G. de Oliveira*, José Geraldo R. Júnior†, Miguel Elias M. Campista*, Luís Henrique M. K. Costa*
*Grupo de Teleinformática e Automação – Universidade Federal do Rio de Janeiro (UFRJ)
Rio de Janeiro – RJ – Brasil
†Centro Federal de Educação Tecnológica de Minas Gerais - CEFET - MG
Leopoldina – MG – Brasil
{oliveira,jgrjunior,miguel,luish}@gta.ufrj.br

***Abstract*—Mobile networks are constructed from specific technologies, such as Bluetooth, Zigbee or the emerging WiFi Direct, and other standards specific to cellular networks. Moreover, there is a variety of mobile devices, increasingly rich in terms of sensors such as GPS, accelerometer, and orientation. The counterpart of this richness of technology is additional complexity to the development of specific applications, because of different programming interfaces. This work proposes ComuniCAR, a library designed to facilitate the development of mobile networking applications for smartphones. The proposed library was developed for Android, one of the most popular smartphone and tablet operating systems. ComuniCAR provides a programming interface that abstracts the details of access to the internal sensor modules and the Android software components. Contextual information extraction, such as speed, location and direction of movement, communication and sharing of such information between neighbors becomes simpler for the programmer. ComuniCAR requires fewer lines of code compared to using specific application programming interfaces. This paper describes the implementation of the ComuniCAR library and presents qualitative and quantitative analysis of the application development using it.**

I. Introdução

O enorme sucesso das comunicações móveis vem estimulando o desenvolvimento de mais e mais dispositivos eletrônicos para uso em rede com tamanhos, preços e finalidades diferentes. Dentre os dispositivos, pode-se destacar os atuais *smartphones*, *tablets* e *smartwatchs*, que vêm dominando o mercado principalmente com a queda de seus preços e o aumento acelerado das suas capacidades computacionais. Como consequência, há um crescimento acentuado no número de usuários, impulsionando a criação de diversas aplicações para o ambiente móvel. Algumas dessas aplicações requerem conectividade entre dispositivos e, portanto, demandam a incorporação de tecnologias de comunicação *Device-to-Device* (D2D), como o *Bluetooth*, e o emergente *WiFi Direct* [1].

O desenvolvimento de aplicações na área de comunicações móveis conta fundamentalmente com interfaces de programação de aplicações (*Application Programming Interface* – API) oferecidas de maneira isolada, tipicamente por sistemas operacionais móveis, como o *Android*. Tais APIs, apesar de já ocultarem detalhes do dispositivo, não permitem um nível elevado de abstração, fazendo com que os desenvolvedores se deparem ainda com procedimentos pouco amigáveis. Sendo assim, há uma carência por bibliotecas de mais alto nível que abstraiam detalhes da programação, notadamente as que se referem às APIs oferecidas pelo sistema operacional móvel. As bibliotecas de mais alto nível podem portanto ajudar na popularização das aplicações ao simplificar o seu desenvolvimento.

Este artigo propõe uma biblioteca denominada ComuniCAR para facilitar o desenvolvimento de aplicações no contexto de redes móveis. A proposta de uma biblioteca é motivada por duas razões distintas. A primeira é que, apesar da complexidade existente para o desenvolvimento de aplicações para as redes móveis, a maioria utiliza um grande número de procedimentos semelhantes que poderiam ser encapsulados e oferecidos para o programador. A segunda é mais simples e consiste na falta de propostas de código aberto [2], [3], [4]. Para cumprir o seu papel, o ComuniCAR assume uma arquitetura modular, considerando a existência de três camadas: aplicação, desenvolvimento e dispositivo. Na primeira, a camada de aplicação, estão representadas as aplicações desenvolvidas por terceiros que utilizam o ComuniCAR para acesso ao dispositivo móvel. Já a camada de desenvolvimento, onde se concentra o ComuniCAR, os procedimentos mais comuns em redes móveis são encapsulados, aumentando o grau de abstração das APIs do dispositivo. A API do sistema operacional do dispositivo móvel é oferecida no nível mais inferior da arquitetura, chamado de nível de dispositivo.

A biblioteca ComuniCAR proposta é avaliada a partir da construção de aplicações específicas para redes ad hoc e para o sistema operacional *Android*. Nesse sentido, dois casos de uso foram definidos, cada um utilizando uma aplicação específica. O primeiro apresenta uma aplicação para obter os dados de localização provenientes do GPS nativo de um *smartphone Android*. O segundo apresenta uma aplicação para descobrir os vizinhos no entorno do *smartphone* em um dado momento utilizando o *WiFi Direct*. Quando a comunicação entre os dispositivos é realizada, os dados de localização são compartilhados entre eles. Considerando os dois casos de uso, a redução da quantidade de esforço do programador pode chegar até 60% em número de linhas de código.

O restante deste trabalho está organizado da seguinte forma. A Seção II introduz os trabalhos relacionados ao ComuniCAR. Já a Seção III apresenta a implementação do ComuniCAR.

Na Seção IV, os dois casos de uso utilizando os recursos do ComuniCAR são apresentados. Por fim, a Seção V conclui este trabalho e apresenta os planos futuros.

II. Trabalhos Relacionados

Esta seção descreve e compara os trabalhos relacionados com o ComuniCAR, sabendo que os trabalhos relacionados também procuram facilitar o desenvolvimento de aplicações para redes móveis.

Park *et al.* [2] propuseram o VoCell, um arcabouço de software que fornece uma biblioteca para o desenvolvimento de aplicações para *smartphones 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 aplicações, mesmo sem grande conhecimento em programação como a linguagem Java, é capaz de criar aplicações no contexto de redes veiculares de forma simplificada, incluindo o VoCell como uma biblioteca nos seus projetos. Esse 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.

Outro arcabouço para a plataforma Android, denominado Reivent, foi proposto por Cardote *et al.* [4]. O Reivent oferece uma biblioteca e foi utilizada no desenvolvimento de duas aplicações na área de entretenimento para redes veiculares: o *VNChat* e o *iThere*. O *VNChat* é uma aplicação de troca de mensagens de texto entre usuários. Já o *iThere* é uma aplicação que tem como objetivo informar ao usuário sobre os amigos (vizinhos) que estão próximos. Ele funciona como uma rede social para redes veiculares, transmitindo, em tempo real, a localização do usuário e de seus vizinhos próximos, utilizando o serviço do Google Maps.

Diferentemente dos trabalhos citados anteriormente, o ComuniCAR tem como objetivo oferecer recursos para o desenvolvimento de aplicações mais genéricas no contexto de redes móveis. O mesmo poderia ser utilizado no desenvolvimento de aplicações para redes veiculares. Por exemplo, uma aplicação de alerta de colisão poderia ser implementada obtendo os dados de localização, sentido e velocidade que são oferecidos pelo GPS do *smartphone*. Ao descobrir e estabelecer uma conexão entre os dispositivos utilizando o *WiFi Direct*, esses dados poderiam ser compartilhados para todos os dispositivos desse grupo. Após obter essas informações, poderia ser calculada 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 [5]. Outro exemplo de aplicação na área de entretenimento poderia ser desenvolvido utilizando os recursos do ComuniCAR onde, com os mesmos recursos utilizados de localização, sentido e velocidade e de conectividade, os dispositivos poderiam compartilhar arquivos como áudio e vídeo.

Um fator importante observado nas pesquisas é que nenhum dos trabalhos citados anteriormente oferecem de forma gratuita as bibliotecas implementadas. 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 aplicações na área de redes móveis.

III. A Biblioteca ComuniCAR

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. Esta biblioteca foi proposta com vistas à extensão, permitindo que os desenvolvedores possam estendê-la a fim de atender as necessidades das aplicações que desejarem implementar. Assim, torna-se mais simples criar novas funcionalidades não pensadas em um primeiro momento ou melhorar as funcionalidades existentes[1].

A ideia é que o ComuniCAR seja disponibilizado de forma gratuita. Assim, quando o usuário for utilizá-lo, deverá incluir os códigos fonte em seu projeto e estender os trechos de código que achar necessário para o desenvolvimento de sua aplicação. Ao oferecer aos desenvolvedores um conjunto de funcionalidades já implementadas no segmento de redes móveis, espera-se facilitar a investigação e estimular o desenvolvimento de aplicações na área.

A. Arquitetura

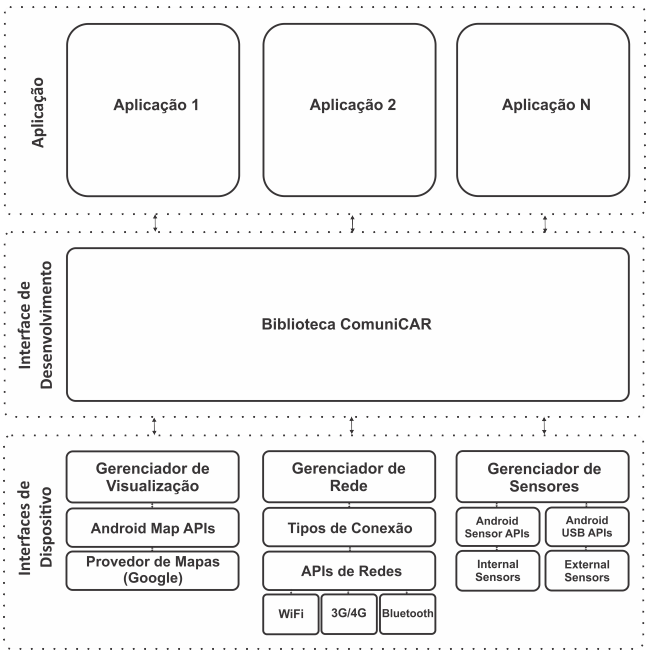


Figura 1. Arquitetura da biblioteca ComuniCAR composta por três camadas: Interfaces de Dispositivo, Interfaces de Desenvolvimento e Aplicação.

[1]A biblioteca pode ser acessada em: http://www.gta.ufrj.br/~oliveira.

A arquitetura do ComuniCAR possui três camadas de implementação: Interfaces de Dispositivo, Interfaces de Desenvolvimento e Aplicação, como apresentado na Figura 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. Note que a biblioteca ComuniCAR pode coexistir com outras bibliotecas também para desenvolvimento. Por fim, a camada mais elevada da biblioteca são as aplicações desenvolvidas por terceiros que podem utilizar o ComuniCAR. A seguir, cada uma das camadas do ComuniCAR é apresentada em maiores detalhes.

Camada de Interfaces de Dispositivo: Camada responsável por realizar o acesso às funcionalidades dos dispositivos, seja para extrair informações de interesse, interagir com outros dispositivos ou acessar funções externas. Essa camada pode ser subdividida em três componentes principais que são o Gerenciador de Sensores, o Gerenciador de Rede e o Gerenciador de Visualização, como definidos no *Android*. 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 executem as operações necessárias para obter dados, como velocidade, sentido e localização. Já o Gerenciador de Rede fornece um conjunto de funções para a criação de conexões e transferência de dados entre os dispositivos utilizando tecnologias de rede sem fio como WiFi, 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.

Camada de Interfaces de Desenvolvimento: Apesar dos dispositivos já oferecerem interfaces de programação a partir do sistema operacional móvel ou de *drivers*, o programador ao criar aplicações específicas no contexto de redes móveis pode encontrar dificuldades durante o desenvolvimento das aplicações. Essas dificuldades são resultado do grande número de funções oferecidas pelas APIs, por exemplo 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 frustrações.

Como o objetivo da biblioteca é facilitar o desenvolvimento das aplicações e muitas funcionalidades básicas são comuns a conjuntos de aplicações, este trabalho define uma camada superior à de Interfaces de Dispositivo, chamada Interfaces de Desenvolvimento. Nessa camada, as funcionalidades comuns são encapsuladas em funções de mais alto nível para o desenvolvimento de aplicações no contexto de redes móveis. O encapsulamento de procedimentos utilizados com frequência em 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. Essa última vantagem, em especial, é importante no contexto do *Android*, visto que as alterações na API do *Android* ainda são frequentes. Nesse caso, quando ocorrer alguma atualização, a camada de Interface de Desenvolvimento é afetada, mas não o código da camada de Aplicação. Sendo assim, todas as atualizações das APIs dos dispositivos podem ser consideradas transparentes à aplicação, o que torna o código das aplicações mais robusto às atualizações nas APIs de mais baixo nível.

Camada de Aplicação: Camada que abriga as aplicações de redes móveis como desenvolvidas por terceiros. As aplicações utilizam as funcionalidades disponibilizadas em alto nível pela camada de Interface de Desenvolvimento. Um dos grandes desafios em desenvolver aplicações para o *Android* é acompanhar a evolução desenfreada da plataforma. A cada novidade, surgem mais APIs, novos arcabouços, novos *widgets*, entre outros. Porém, as atualizações podem comprometer o funcionamento das aplicações surpreendendo os desenvolvedores. Por exemplo, a partir do *Android* 6.0 (API 23), as aplicações passaram a solicitar permissões em tempo de execução para utilização de determinadas APIs, como por exemplo: localização, câmera, escrita no *sdcard*, entre outros [6].

B. Implementação

A implementação do ComuniCAR assume o uso de componentes de Gerenciamento de Visualização, Sensores e Rede, como fornecidos pelo *Android*. 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 nos casos de uso deste trabalho.

Assumindo o uso do *Android* na camada de Interfaces de Dispositivos, é possível implementar as funções da Camada de Interfaces de Desenvolvimento usando um ambiente de programação específico. Tal ambiente facilita a programação, pois possui as funções dos dispositivos já embutidas. A seguir, a implementação da camada de Interfaces de Desenvolvimento, assim como as principais funções criadas para essa camada, é apresentada. A implementação da Camada de Interfaces de Desenvolvimento resultou na criação de novas classes e funções de mais alto nível, levando em conta as funções disponíveis através da API do *Android*. Abaixo, a implementação do serviço de localização e o mecanismo de comunicação são apresentados.

Serviço de localização: A maioria dos dispositivos *Android* permitem a determinação de sua localização geográfica atual. Isso pode ser realizado por meio da utilização de um módulo GPS, normalmente interno, rede GSM ou através de redes WiFi [7]. O Google Play Services facilita a conexão com os serviços do Google, além de oferecer várias APIs como

localização, Google Fit, Google+, GoogleDrive, Games, entre outros [7]. A API de localização que foi utilizada neste trabalho é conhecida como *Fused Location Provider*. Depois de definir a API de localização, foi implementada a classe `Sensors` que é responsável por facilitar a programação para a obtenção dos dados de localização do GPS em tempo real. A seguir, serão explicados os principais métodos e funções que foram implementados nessa classe.

Para utilizar qualquer uma das APIs gerenciadas pelo Google Play Services, é necessário se conectar aos serviços do Google. A classe responsável pela conexão é a *GoogleApiClient*, que encapsula toda a comunicação com os servidores do Google. O Código 1 apresenta o método `void callConnection()`, implementado para construir um objeto do tipo `GoogleApiClient`.

```
/**
* Codigo 1: Trecho de codigo utilizado para conectar
     com os servidores do Google
*/
protected synchronized void callConnection() {
      mGoogleApiClient = new GoogleApiClient.Builder
           (mActivity)
        .addOnConnectionFailedListener(this)
        .addConnectionCallbacks(this)
        .addApi(LocationServices.API)
        .build();
}
```

Uma vez que a aplicação está conectada aos serviços do Google, foi implementado o método `startLocationUpdate()`, responsável por configurar e iniciar o GPS. Primeiro, um objeto `LocationRequest` é instanciado contendo as configurações referentes à precisão e ao intervalo de tempo com que o desenvolvedor deseja receber as coordenadas. O método `setPriority(int)` define a precisão do GPS, já o `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 o desempenho da aplicação. Outro método importante é o *FusedLocationApi*, responsável por recuperar a última localização conhecida pelo sistema.

```
/**
* Codigo 2: Implementacao do metodo
    startLocationUpdate()
*/
protected void startLocationUpdate() {
  mLocationRequest = new LocationRequest();
  mLocationRequest.setInterval(0);
  mLocationRequest.setFastestInterval(0);
  mLocationRequest.setPriority(LocationRequest.
      PRIORITY_HIGH_ACCURACY);

  if (ActivityCompat.checkSelfPermission(mActivity,
      Manifest.permission.ACCESS_FINE_LOCATION) !=
      PackageManager.PERMISSION_GRANTED &&
      ActivityCompat.checkSelfPermission(mActivity,
      Manifest.permission.ACCESS_COARSE_LOCATION) !=
       PackageManager.PERMISSION_GRANTED) {
    return;
  }
LocationServices.FusedLocationApi.
    requestLocationUpdates(mGoogleApiClient,
    mLocationRequest, Sensores.this);
}
```

No Código 3, o método `stopLocationUpdate()` responsável por parar o GPS é apresentado.

```
/**
* Codigo 3: Implementacao do metodo
    stopLocationUpdate()
*/
protected void stopLocationUpdate(){
    LocationServices.FusedLocationApi.
    removeLocationUpdates(mGoogleApiClient, Sensores
    .this);
}
```

O método `onLocationChanged(location)` foi implementado para receber as localizações provenientes do GPS. Sempre que o *Android* tiver uma nova localização, esse método é invocado passando como parâmetro um objeto `android.location.Location`. Esse método pertence à interface `LocationListener` e está representado no Código 4.

```
/**
* Codigo 4: Implementacao do metodo
    onLocationChanged(location)()
*/
public void onLocationChanged(Location location) {
  mLocation = location;
  if(mInterface!=null)
    mInterface.seekingGPSData(location);
}
```

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 a classe `Location` que é uma classe de dados que representa uma localização geográfica [6]. Foram implementadas as funções que chamam os métodos da classe `Location`, representada no Código 5.

```
/**
* Codigo 5: Implementacao dos metodos da classe
    Location()
*/
public double getLatitude() {
  return mLocation==null?null:mLocation.getLatitude
      ();
}
public double getLongitude() {
  return mLocation==null?null:mLocation.getLongitude
      ();
}
public double getAltitude() {
  return mLocation==null?null:mLocation.getAltitude
      ();
}
public String getOrientation() {
  return convertDirOrient(mLocation);
}
public float getSpeed() {
  return mLocation==null?null:mLocation.getSpeed()*(
      float)3.6;
}
public float getPrecision() {
  return mLocation==null?null:mLocation.getAccuracy
      ();
}
```

```
public float getDirection() {
  return mLocation==null?null:mLocation.getBearing()
      ;
}
public String getTime() {
  return mLocation==null?null:new SimpleDateFormat("
      hh:mm:ss").format(new Date(mLocation.getTime()
      ));
}
```

Foi implementada ainda uma nova função que não pertence à classe `Location`, chamada de `getOrientation()`, que está representada no Código 5. Essa função realiza uma chamada para outra função `convertDirOrient(location)` que recebe como parâmetro um objeto `android.location.Location` e realiza 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).

Depois de implementar a classe `Sensors`, foi desenvolvida uma interface chamada de `SensorsInterface` representado no Código 6. Essa interface implementa o método `seekingGPSData()`, que é chamado dentro do método `onLocationChanged(location)`, da classe `Sensors`.

```
/**
* Codigo 6: Implementacao da interface
    SensorsInterface
*/
public interface SensorsInterface {
    void seekingGPSData(Location newLocation);
}
```

Mecanismo de comunicação: Para o mecanismo de comunicação, foi utilizada a API do *WiFi Direct* disponibilizada pelo *SDK Android*. Essa API permite que a aplicação, instalada em sistema operacional Android 4.0 ou versões posteriores com hardware apropriado, se conecte diretamente com outros dispositivos via WiFi, sem um ponto de acesso intermediário. Depois de definir a API, foi implementada a classe `Neighbors`, representada no Código 7, que é responsável por facilitar a programação de conectividade entre os *smartphones* utilizando o *WiFi Direct*. A seguir, segue a explicação dos principais métodos e funções que foram implementados nesta classe.

```
/**
* Codigo 7: Trecho de codigo da implementacao da
    classe Neighbors
*/
public class Neighbors implements WifiP2pManager.
    PeerListListener,        WifiP2pManager.
    ConnectionInfoListener, SocketInterface {

  public void setWifiDirect() {
    mManager = (WifiP2pManager)mActivity.
        getSystemService(Context.WIFI_P2P_SERVICE);
    mChannel = mManager.initialize(mActivity,
        mActivity.getMainLooper(), null);
    mReceiver = new WiFiDirectBroadcastReceiver(
        mManager, mChannel, mActivity, this);
    mIntentFilter = new IntentFilter();
      mIntentFilter.addAction(WifiP2pManager.
          WIFI_P2P_STATE_CHANGED_ACTION);
      mIntentFilter.addAction(WifiP2pManager.
          WIFI_P2P_PEERS_CHANGED_ACTION);
      mIntentFilter.addAction(WifiP2pManager.
          WIFI_P2P_CONNECTION_CHANGED_ACTION);
    mIntentFilter.addAction(WifiP2pManager.
        WIFI_P2P_THIS_DEVICE_CHANGED_ACTION);
  }
}
```

No método `setWifiDirect()` foi realizada uma instância do `WifiP2pManager` que é utilizada para registrar a aplicação ao *WiFi Direct* por meio do método `initialize()`. Esse método retorna um `WifiP2pManager.Channel`, que foi utilizado para ligar a aplicação com o *WiFi Direct*. Foi criado também o `WiFiDirectBroadcastReceiver`, onde foram registrados 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 saber se o *WiFi Direct* dos dispositivos encontra-se ligado/desligado ou se um dispositivo perdeu uma conexão *WiFi Direct*.

Tendo os dois objetos criados, o `WifiP2pManager` e o `WifiP2pManager.Channel`, as aplicações já podem chamar os métodos da classe `WifiP2pManager` para utilizarem as funcionalidades fornecidas pelo *WiFi Direct*. Foi utilizado o *BroadcastReceiver* para a aplicação receber as notificações de eventos importantes, como por exemplo: saber se o WiFi está ligado ou não, se o dispositivo perdeu a conexão a uma rede ou conectou-se a uma rede, entre outras notificações [6]. Utilizando as APIs *WiFi Direct* do *SDK Android*, existem duas formas de se conectar a um dispositivo *2P Group*: através da chamada aos métodos `connect()` e `createGroup()` da classe `WifiP2pManager` [6].

No método `createGroup()`, é realizada a técnica de formação de grupo *Autonomous*. Portanto o dispositivo cria um *P2P Group* onde ele é *P2P GO*. Através do método `connect()` um dispositivo conecta-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 estabelecimento da conexão com o dispositivo desejado.

Segundo a especificação do *WiFi Direct*, para realizar uma conexão com outro dispositivo *WiFi Direct*, é preciso que este seja antes descoberto. Nas APIs do *WiFi Direct* do *SDK Android*, a descoberta dos pares pode ser realizada através da chamada do método `discoverPeers()` da classe `WifiP2pManager`, representado no Código 8.

```
/**
* Codigo 8: Implementacao do metodo discoverPeers()
*/
public void discoverPeers() {
  mManager.discoverPeers(mChannel, new
      WifiP2pManager.ActionListener() {
    public void onSuccess() {
    }
    public void onFailure(int reasonCode) {
    }
  });
}
```

Também foram implementadas as funções e os métodos para possibilitar a troca de mensagens dos dados de localização (provenientes do GPS) entre os dispositivos utilizando o protocolo JSON, como apresentado no Código 9.

```
/**
* Codigo 9: Implementacao dos metodos
    getServerMessage(), onClientResponse(String
    response), getClientMessage() e onSeverResponse(
    String response)
*/
public String getServerMessage() {
  return mGson.toJson(new MyLocation(mLocation));
}
public void onClientResponse(String response) {
  t("CL "+ response);
}
public String getClientMessage() {
  return mGson.toJson(new MyLocation(mLocation));
}
public void onSeverResponse(String response) {
  t("SV "+ response);
}
```

Depois de implementar a classe `Neighbors`, foi desenvolvida uma interface chamada de `NeighborsInterface`, representada no Código 10. Essa interface implementa o método `onNeighborsUpdate()`.

```
/**
* Codigo 10: Implementacao da interface
    NeighborsInterface
*/
public interface NeighborsInterface {
    void onNeighborsUpdate(Collection<WifiP2pDevice>
         newList);
}
```

Para exemplificar a eficiência do ComuniCAR, dois trechos de códigos que implementam a mesma função `getOrientation()`, explicada anteriormente, são apresentados. O primeiro trecho, Código 11, apresenta a implementação da função utilizando os recursos do ComuniCAR. Caso a implementação não utilize o ComuniCAR, o número de linhas de código necessárias para a sua implementação é bem maior, como pode ser observado por comparação com o Código 12.

```
/**
* Codigo 11: Trecho de codigo da implementacao da
    funcao getOrientation() utilizando o ComuniCAR
*/
public String getOrientation() {
  return convertDirOrient(mLocation);
}
```

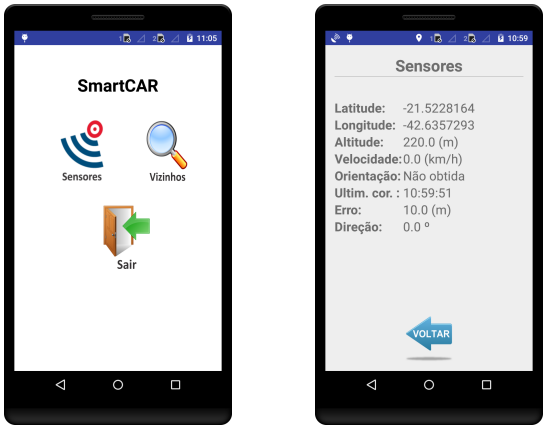
```
/**
* Codigo 12: Trecho de codigo da implementacao da
    funcao getOrientation() sem a utilizacao do
    ComuniCAR
*/
public String getOrientation() {
    mLocation = location;
    String orientation = "Nao obtida";
        if (mLocation == null){
            orientacao = "null";
        }
        else if (mLocation.hasBearing() != false) {
            float sense = mLocation.getBearing();
            if (sense >= 0 && sense <= 10) {
                orientation = "N";
            } else if (sense >= 11 && sense <= 33) {
                orientation = "N-NE";
            } else if (sense >= 34 && sense <= 55) {
                orientation = "NE";
            } else if (sense >= 56 && sense <= 78) {
                orientation = "E-NE";
            } else if (sense >= 79 && sense <= 100)
                {
                orientation = "E";
            } else if (sense >= 101 && sense <= 123)
                {
                orientation = "E-SE";
            } else if (sense >= 124 && sense <= 145)
                {
                orientation = "SE";
            } else if (sense >= 146 && sense <= 168)
                {
                orientation = "S-SE";
            } else if (sense >= 169 && sense <= 190)
                {
                orientation = "S";
            } else if (sense >= 191 && sense <= 213)
                {
                orientation = "S-SO";
            } else if (sense >= 214 && sense <= 235)
                {
                orientation = "SO";
            } else if (sense >= 236 && sense <= 258)
                {
                orientation = "O-SO";
            } else if (sense >= 259 && sense <= 280)
                {
                orientation = "O";
            } else if (sense >= 281 && sense <= 303)
                {
                orientation = "O-NO";
            } else if (sense >= 304 && sense <= 325)
                {
                orientation = "NO";
            } else if (sense >= 326 && sense <= 348)
                {
                orientation = "N-NO";
            } else if (sense >= 349 && sense <= 359)
                {
                orientation = "N";
            }
        }
        return orientation;
}
```

IV. Estudo de Caso

Nesta seção é apresentado um exemplo de aplicação denominada SmartCAR, o qual foi desenvolvido utilizando os recursos da biblioteca ComuniCAR. A implementação dessa aplicação permite a avaliação dos ganhos trazidos pela biblioteca tanto no processo de desenvolvimento quanto na geração de cenários para a avaliação em redes móveis. O SmartCAR foi implementado na camada de Aplicação, conforme a arquitetura da Figura 1.

O SmartCAR foi implementado em Java e XML (*eXtensible Markup Language*) através da IDE Android Studio [6]. A linguagem Java foi utilizada na elaboração das *Activities*, que 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 da aplicação. 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 2(a). As opções são: *Sensores* (obtém os dados de localização em tempo real do GPS. Os dados são: Latitude, Longitude, Altitude, Velocidade em km/h, Orientação, Precisão e Direção), *Vizinhos* (realiza a descoberta e apresenta os vizinhos que estão no entorno em um dado momento utilizando o *WiFi Direct*. Ao estabelecer a conexão, os dados de localização entre os vizinhos podem ser compartilhados utilizando o JSON [8] como protocolo de comunicação) e *Sair* (fecha a aplicação).



(a) Tela do menu principal.

(b) Tela para obtenção de dados.

Figura 2. Telas do SmartCAR.

A. *Avaliação das aplicações do SmartCAR*

Nesta seção, são avaliadas as aplicações para obtenção de dados de localização do GPS em tempo real e a descoberta e conexão dos vizinhos em um dado momento utilizando o *WiFi Direct*. Os resultados se referem à economia de código no desenvolvimento das aplicações com e sem o ComuniCAR.

Obtenção dos dados de localização: 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 necessitem da localização do dispositivo em tempo real (Figura 2(b)).

Para obter esses dados, foi criada uma classe chamada de `ApSensors` que herdou da classe `Activity` e implementou a interface `SensorsInterface`. O código 13 apresenta um trecho de código da aplicação desenvolvida:

```
/**
* Codigo 13: Trecho de codigo da aplicacao que obtem
     os dados de localizacao em tempo real do GPS
    utilizando o ComuniCAR.
*/
package gta.smartcar;
import gta.ComuniCAR.Sensors;
import gta.ComuniCAR.SensorsInterface;

public class ApSensors extends Activity implements
    SensorsInterface {
    Sensores mSensors;
    mSensors = new Sensors(this,this);
    mSensors.onCreate();

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

Ao utilizar a interface `SensorsInterface`, o programador implementa somente o método `seekingGPSData()` para obter os dados de localização em tempo real do GPS, pois todos os outros métodos já estão encapsulados na classe `Sensors` (Seção IV). Essa aplicação, ao utilizar os recursos do ComuniCAR, teve uma economia superior a 60% referente ao número de linhas de código quando comparada a aplicação que utilizou somente as interfaces de programação nativas do *Android*, como visto na Figura 3. Não foram considerados os códigos referentes aos arquivos XML das *Activities* para essas métricas. O ComuniCAR diminui a quantidade de código concentrado nos métodos de localização do *Android*.

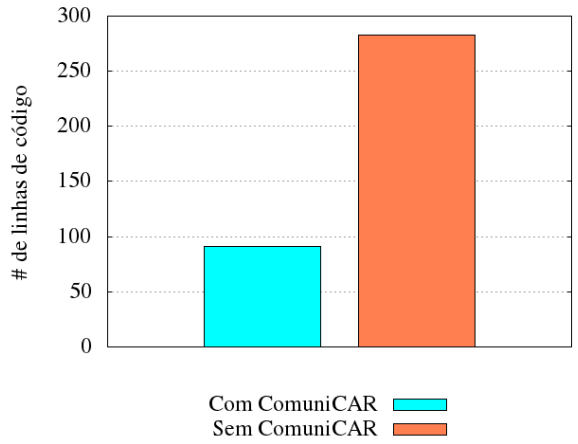


Figura 3. Economia no número de linhas do ComuniCAR para obtenção dos dados de localização.

Descoberta dos vizinhos e compartilhamento dos dados do GPS em tempo real: Ao clicar no menu "Vizinhos" (Figura 2(a)) é exibida 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 realizada uma conexão sem fio utilizando o *WiFi Direct* permitindo a troca de informações dos dados dos sensores entre os dispositivos, como ilustrado pela Figura 4.

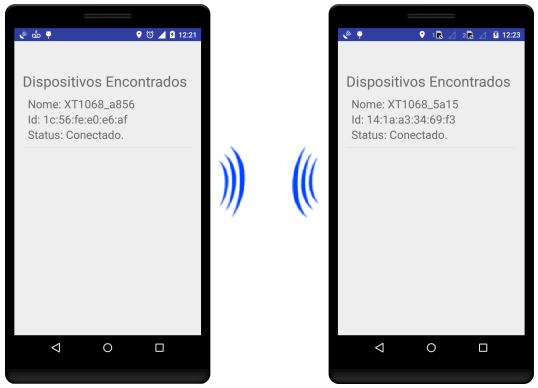


Figura 4. Tela de descoberta dos vizinhos em um dado momento.

Para desenvolver esta aplicação, foi criada a classe `ApNeighbors` que herdou da classe `Activity` e implementou as interfaces `NeighborsInterface` e `SensorsInterface`, como apresentada no Código 14. Ao utilizar a interface `NeighborsInterface`, o desenvolvedor implementa somente o método `onNeighborsUpdate()` para obter a lista dos vizinhos, pois todos os outros métodos já estão encapsulados na classe `Neighbors`, apresentada na Seção IV.

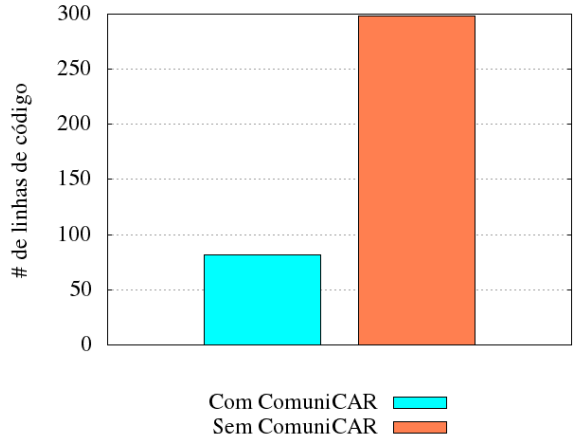


Figura 5. Economia no número de linhas do ComuniCAR para descoberta dos vizinhos e compartilhamento dos dados do GPS em tempo real.

```
/**
* Codigo 14: Trecho de codigo da aplicacao que
    descobre os vizinhos e compartilha os dados do
    GPS em tempo real utilizando o ComuniCAR
*/
import gta.ComuniCAR.Sensors;
import gta.ComuniCAR.SensorsInterface;
import gta.ComuniCAR.Neighbors;
import gta.ComuniCAR.NeighborsAdapter;
import gta.ComuniCAR.NeighborsInterface;

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

A aplicação, que utilizou os recursos do ComuniCAR, teve uma economia superior a 63% referente ao número de linhas de código quando comparado à aplicação que utilizou somente as interfaces de programação nativas do *Android*, como ilustrado na Figura 5. Mais uma vez, não foram considerados os códigos referentes aos arquivos XML das *activities*.

V. Conclusões e Trabalhos Futuros

Neste artigo foi apresentado o ComuniCAR, uma biblioteca que visa facilitar o desenvolvimento de aplicações de redes móveis. A biblioteca proposta utilizou o emergente *WiFi Direct* para realizar a conexão entre os dispositivos. Utilizando essa biblioteca no desenvolvimento dos casos de uso apresentados, houve uma redução superior de 60% no número de linhas de código em relação às APIs oferecidas pelo *Android*. O ComuniCAR abre várias possibilidades de estudos e pesquisas que podem trazer melhorias e benefícios na área de desenvolvimento de aplicações para redes móveis. Entre as sugestões de trabalhos futuros é possível citar: o suporte a redes infraestruturadas, a verificação da escalabilidade do sistema, a verificação do tempo de atraso no encaminhamento das mensagens da origem ao destino, a implementação e validação de novos casos de uso e o estudo sobre a segurança da rede construída utilizando o *WiFi Direct*.

Agradecimentos

Os autores gostariam de agradecer o apoio da CAPES, do CNPq e da FAPERJ.

Referências

[1] 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, Wi-Fi Alliance, 2014.

[2] Y. Park, J. Ha, S. Kuk, H. Kim, C.-J. M. Liang, and J. Ko, "A feasibility study and development framework design for realizing smartphone-based vehicular networking systems," *Mobile Computing, IEEE Transactions on*, vol. 13, no. 11, pp. 2431–2444, Nov 2014.

[3] M. Ambrosin, A. Bujari, M. Conti, F. Gaspari, and C. E. Palazzi, "Smartphone and laptop frameworks for vehicular networking experimentation," *IEEE Conference Publications*, 2013.

[4] F. Oliveira, S. Sargento, J. M. Fernandes, and A. Cardote, "Reivent: Acessing vehicular networks in mobile applications," *IEEE Symposium on Computers and Communication (ISCC)*, 2014.

[5] Y. L. Chen, K. Y. Shen, and S. C. Wang, "Forward collision warning system considering both time-to-collision and safety braking distance," 2013, pp. 972–977.

[6] Android Developers, "Android studio release notes," Disponível em http://developer.android.com/intl/pt-br/develop/index.html, Acesso em 01 oct. 2015, Google, 2016.

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

[8] JSON, "Ecma-404 the json data interchange standard," Disponível em http://json.org, Acesso em 02 oct. 2015, JSON, 2016.