

**MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO**

ANDRÉ LUIZ MATOS ROCHA CARNEIRO

1º Ten JULIANDRO LOPES TRUGILHO

1º Ten RODRIGO GOMES DEMETERKO

**DESENVOLVENDO UMA INTERFACE GRÁFICA DE USUÁRIO
UTILIZANDO A API NEARBY DO GOOGLE**

**Rio de Janeiro
2017**

INSTITUTO MILITAR DE ENGENHARIA

**ANDRÉ LUIZ MATOS ROCHA CARNEIRO
1º Ten JULIANDRO LOPES TRUGILHO
1º Ten RODRIGO GOMES DEMETERKO**

**DESENVOLVENDO UMA INTERFACE GRÁFICA DE
USUÁRIO UTILIZANDO A API NEARBY DO GOOGLE**

Projeto de Fim de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Engenheiro de Computação.

Orientador: Maj Anderson Fernandes Pereira dos Santos - D.Sc.

Rio de Janeiro
2017

c2017

INSTITUTO MILITAR DE ENGENHARIA
Praça General Tibúrcio, 80 - Praia Vermelha
Rio de Janeiro - RJ CEP 22290-270

Este exemplar é de propriedade do Instituto Militar de Engenharia, que poderá incluí-lo em base de dados, armazenar em computador, microfilmar ou adotar qualquer forma de arquivamento.

É permitida a menção, reprodução parcial ou integral e a transmissão entre bibliotecas deste trabalho, sem modificação de seu texto, em qualquer meio que esteja ou venha a ser fixado, para pesquisa acadêmica, comentários e citações, desde que sem finalidade comercial e que seja feita a referência bibliográfica completa.

Os conceitos expressos neste trabalho são de responsabilidade do(s) autor(es) e do(s) orientador(es).

006.3 Carneiro, André Luiz Matos Rocha
C289d Desenvolvendo uma Interface Gráfica de Usuário utilizando a API Nearby do Google / André Luiz Matos Rocha Carneiro, Juliandro Lopes Trugilho, Rodrigo Gomes Demeterko, orientado por Anderson Fernandes Pereira dos Santos - Rio de Janeiro: Instituto Militar de Engenharia, 2017.

83p.: il.

Projeto de Fim de Curso (graduação) - Instituto Militar de Engenharia, Rio de Janeiro, 2017.

1. Curso de Graduação em Engenharia de Computação - projeto de fim de curso. 1. Android. 2. API. 3. Nearby. 4. Beacon. 5. Bluetooth. I. dos Santos, Anderson Fernandes Pereira . II. Título. III. Instituto Militar de Engenharia.

INSTITUTO MILITAR DE ENGENHARIA

ANDRÉ LUIZ MATOS ROCHA CARNEIRO
1º Ten JULIANDRO LOPES TRUGILHO
1º Ten RODRIGO GOMES DEMETERKO

DESENVOLVENDO UMA INTERFACE GRÁFICA DE
USUÁRIO UTILIZANDO A API NEARBY DO GOOGLE

Projeto de Fim de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Engenheiro de Computação.

Orientador: Maj Anderson Fernandes Pereira dos Santos - D.Sc.

Aprovado em 29 de Setembro de 2017 pela seguinte Banca Examinadora:

Maj Anderson Fernandes Pereira dos Santos - D.Sc. do IME - Presidente

Prof. Ricardo Choren Noya - D.Sc. do IME

Maj Humberto Henriques de Arruda - M.Sc. do IME

Rio de Janeiro
2017

Às nossas famílias, que nos apoiaram em todo o caminho percorrido até aqui. Aos amigos, que sempre estiveram presentes, nos melhores e piores momentos.

AGRADECIMENTOS

Agradeçemos a todos os mestres, que no decorrer de nossa formação se empenharam para formar engenheiros competentes e compromissados com o país.

Agradecemos em especial ao nosso Professor Orientador Dr. Anderson Fernandes Pereira dos Santos e aos outros membros da banca pela disponibilidade e atenção.

“Os que se encantam com a prática sem a ciência são como os timoneiros que entram no navio sem timão nem bússola, nunca tendo certeza do seu destino.”

LEONARDO DA VINCI

SUMÁRIO

LISTA DE ILUSTRAÇÕES	9
LISTA DE SIGLAS	10
1 INTRODUÇÃO	13
1.1 Motivação	13
1.2 Objetivo	14
1.3 Justificativa	14
1.4 Metodologia	14
1.5 Organização da Monografia	14
2 FUNDAMENTAÇÃO TEÓRICA	16
2.1 Plataforma Android	16
2.1.1 Arquitetura	16
2.1.2 Componentes de uma Aplicação	18
2.1.2.1 Ativação de Componentes	19
2.1.3 Manifesto da Aplicação	19
2.1.4 Permissões	19
2.1.4.1 Especificidades do Android 6.0	20
2.2 Arquitetura <i>Publish/Subscribe</i>	21
2.3 API Nearby Messages	21
2.3.1 <i>Beacon</i>	22
2.3.1.1 Formato <i>Eddystone</i>	22
2.4 API <i>Nearby Connections</i>	23
2.5 REST	23
2.5.1 REST API	24
2.6 Django <i>Framework</i>	24
3 MODELAGEM	25
3.1 Aplicação <i>Mobile</i>	25
3.1.1 <i>Activities</i>	25
3.1.2 djangoAPI	26
3.1.3 NearbyWrapper	26
3.1.4 Classes Auxiliares	27

3.2	Servidor	27
3.2.1	Banco de Dados.....	27
3.2.1.1	Django <i>Models</i>	27
3.2.1.2	API <i>Models</i>	28
3.2.1.3	<i>Boards Models</i>	28
3.2.2	REST API	28
3.2.2.1	<i>Login</i>	29
3.2.2.2	Troca de arquivos	29
3.2.2.3	<i>Beacon</i>	29
3.2.3	Quadro de Avisos	29
4	DESENVOLVIMENTO	30
4.1	Servidor	30
4.1.1	<i>api</i>	30
4.1.1.1	<i>views.py</i>	30
4.1.1.2	<i>models.py</i>	32
4.1.1.3	<i>serializers.py</i>	32
4.1.2	<i>boards</i>	32
4.1.2.1	<i>views.py</i>	32
4.1.2.2	<i>models.py</i>	33
4.2	Aplicação <i>Mobile</i>	33
4.2.1	<i>Login</i> do usuário	33
4.2.2	Troca de Arquivos	34
4.2.2.1	<i>Upload</i>	35
4.2.2.2	Listagem de Arquivos	38
4.2.2.3	Envio do Código	39
4.2.2.4	Recebimento do código	40
4.2.2.5	<i>Download</i>	41
4.2.3	<i>Beacons</i> e Quadros de Avisos	43
4.3	Dificuldades Encontradas	44
5	CONCLUSÃO	45
5.1	Oportunidades para Trabalhos Futuros	45
6	REFERÊNCIAS BIBLIOGRÁFICAS	46

7	APÊNDICES	48
7.1	APÊNDICE 1: Códigos das Classes Principais	49
7.1.1	HomeActivity.java	49
7.1.2	ListActivity.java	55
7.1.3	Receive.java	60
7.1.4	beaconActivity.java	64
7.1.5	NearbyAPIWrapper.java	69
7.1.6	beaconAPI.java	72
7.1.7	downloadAPI.java	74
7.1.8	listAPI.java	77
7.1.9	uploadAPI.java	78

LISTA DE ILUSTRAÇÕES

FIG.2.1	Arquitetura em camadas da plataforma Android.	18
FIG.2.2	Exemplo de arquivo <i>AndroidManifest.xml</i>	20
FIG.2.3	Funcionamento da API <i>Nearby Messages</i> (LEGENDRE, 2015)	22
FIG.4.1	Tela de <i>login</i>	34
FIG.4.2	Política de <i>threads</i> e exemplo de requisição de permissão	35
FIG.4.3	Estrutura básica da troca de arquivos	36
FIG.4.4	Escolha do arquivo com <i>Samsung File Manager</i>	36
FIG.4.5	Listagem de arquivos	38
FIG.4.6	Listagem dos usuários que querem enviar um arquivo	41
FIG.4.7	Requisição da senha do arquivo desejado	42
FIG.4.8	Lista dos quadros de avisos disponíveis	43

LISTA DE SIGLAS

API	<i>Application Programming Interface</i>
APK	<i>Android Package</i>
BLE	<i>Bluetooth Low Energy</i>
HAL	<i>Hardware Abstraction Layer</i>
HTTP	<i>Hypertext Transfer Protocol</i>
IDE	<i>Integrated Development Environment</i>
JSON	<i>JavaScript Object Notation</i>
LIFO	<i>Last In, First Out</i>
REST	<i>Representational State Transfer</i>
SDK	<i>Software Development Kit</i>
SO	Sistema Operacional
SSID	<i>Service Set Identifier</i>
URI	<i>Uniform Resource Identifier</i>
URL	<i>Uniform Resource Locator</i>
VM	<i>Virtual Machine</i>

RESUMO

Este projeto tem como finalidade propor, modelar e desenvolver uma aplicação *Android* utilizando a API *Nearby* do *Google*, e que, como caso de teste, realize troca de arquivos entre dois dispositivos.

Inicialmente, a finalidade da aplicação não era clara. Coube à equipe do projeto, a partir dos estudos feitos sobre o funcionamento da API e de seus possíveis usos, definir as funcionalidades da aplicação final, que são: troca de arquivos entre dispositivos próximos e acesso a quadros de avisos baseado na localização do usuário, utilizando *beacons*.

Tanto para a implementação da troca de arquivos quanto para a criação de conteúdo a ser acessado a partir dos *beacons*, foi necessário o desenvolvimento de um servidor *Web*, que foi implementado utilizando o *Framework* Django. Optou-se por esse *framework* pois ele oferece grande portabilidade, escalabilidade e segurança, além de várias bibliotecas *open source*.

Após o término do desenvolvimento, o aplicativo foi submetido a testes de verificação das funcionalidades previstas. Os testes foram realizados dentro do Instituto Militar de Engenharia, e mostraram o bom comportamento do aplicativo durante seu uso.

ABSTRACT

This project aims to propose, model and develop an Android application using the Google Nearby API and, as a test case, exchange files between two devices.

Initially, the purpose of the application was not clear. From the studies made on the functioning of the API and its possible uses, it was left to the project team to define the functionalities of the final application, which are: exchange of files between devices and access to bulletin boards based on the user's location, using beacons.

Both the implementation of file exchange and the creation of content to be accessed from the beacons required the development of a web server, which was implemented using the Framework Django. This framework was chosen because it offers great portability, scalability and security, in addition to several open source libraries.

After the development has finished, the application has undergone verification tests for the expected functionalities. The tests were carried out inside the Military Engineering Institute, and showed the good behavior of the application during its use.

1 INTRODUÇÃO

A vida do ser humano é baseada no modo como ele convive e se comunica com seus semelhantes. Toda a obtenção e propagação de conhecimento, motor que impulsiona o desenvolvimento de uma sociedade, se dá através de meios de comunicação.

Analizando-se o processo evolutivo das sociedades, percebe-se que a evolução dos meios de comunicação, e tecnologias a eles associadas, aumentou as possibilidades de busca de conhecimento, o que por si só influencia o ritmo de desenvolvimento de uma sociedade. Essa afirmação faz mais sentido ainda quando se observa a atual situação do mundo, em plena era do conhecimento, onde informações são o ativo mais importante. Portanto, o modo como informações são tratadas e compartilhadas possui um papel primordial dentro desse escopo.

Com a grande difusão do uso de dispositivos móveis e suas respectivas plataformas, ocorrida nos últimos dez anos, foram abertos precedentes para grandes investimentos na área. Um exemplo disso é o crescimento acelerado do desenvolvimento e uso de sistemas multiplataforma. Desse modo, ainda existem inúmeras possibilidades não exploradas nesse escopo.

Este projeto procura explorar algumas dessas possibilidades. A API Nearby do Google, ponto principal do projeto, oferece a possibilidade de realizar interações entre dispositivos próximos, com várias aplicações possíveis.

1.1 MOTIVAÇÃO

A transferência de arquivos entre dispositivos baseada em proximidade possui inúmeras aplicações, seja no meio acadêmico (crescimento elevado de sistemas EaD e o uso de tecnologias em sala de aula) como no meio empresarial (distribuição de informações pela empresa com maior dinamicidade). Existem ainda ferramentas ligadas à API Nearby que podem criar ainda mais possibilidades, tais como os *Beacons*. Logo, o entendimento mais aprofundado sobre as capacidades da API Nearby pode trazer *insights* para projetos de alto valor agregado no futuro.

1.2 OBJETIVO

Este projeto tem como objetivo propor, modelar e desenvolver uma interface de usuário que possibilite a troca de arquivos entre dispositivos Android próximos, fazendo uso da API Nearby do Google.

1.3 JUSTIFICATIVA

O desenvolvimento dessa aplicação levará a um aprofundamento nos conhecimentos acerca do uso da plataforma Android, bem como no seu uso associado à funcionalidades de uma API. Esses conhecimentos são utilizados amplamente no mercado de trabalho na área de Computação, o que serve como motivação tanto para alunos quanto professores. A troca de arquivos por proximidade gera uma praticidade dificilmente encontrada atualmente, aumentando o valor agregado de um projeto nessa área. Para o Exército Brasileiro, possuir profissionais capacitados nessa área contribui para o aumento das possibilidades de desenvolvimento de produtos de defesa, dado que a área cibernética é o foco dos atuais projetos estratégicos da Força.

1.4 METODOLOGIA

Inicialmente havia a necessidade de um estudo prévio sobre a plataforma Android, sobre o funcionamento da API Nearby do Google e suas aplicações. Após entender as possibilidades e limitações dessas ferramentas, foi feita a modelagem da aplicação final desejada, englobando algumas das possibilidades de uso da API. Após a modelagem completa, seguiram-se as etapas de implementação e testes de cada módulo da aplicação. Finalmente, foram feitos os testes finais de funcionamento do aplicativo completo.

1.5 ORGANIZAÇÃO DA MONOGRAFIA

No capítulo 2 deste trabalho é realizada uma introdução aos conceitos, características e modo de funcionamento da plataforma Android e da API Nearby. Também são abordados conceitos da REST API e do Django Framework, que serão utilizados na implementação do aplicativo.

No capítulo 3, é exposta a modelagem da aplicação. Nela, a equipe do projeto se baseará para as etapas de implementação de cada módulo.

No capítulo 4, são expostos detalhes acerca da implementação das funcionalidades, como, por exemplo, descrições das classes utilizadas e de como elas interagem entre si

para o funcionamento correto das soluções propostas.

Finalmente, no capítulo 5, são apresentadas as conclusões obtidas com os resultados alcançados pelo projeto, além de possíveis temas para pesquisas e projetos futuros. Na sequência, são listadas as referências utilizadas neste trabalho, e nos apêndices são incluídos os códigos das classes principais da aplicação.

2 FUNDAMENTAÇÃO TEÓRICA

Serão abordados, nessa seção, os conceitos relativos à plataforma Android, à API Nearby do Google, à REST API e também ao Django *Framework*, temas principais deste trabalho, proporcionando ao leitor uma melhor compreensão sobre esses assuntos e do trabalho como um todo.

2.1 PLATAFORMA ANDROID

No mercado de dispositivos móveis, a plataforma Android tem sido a mais difundida dos últimos anos. Alguns dos motivos para tal fato são seus atrativos para empresas de tecnologia, tais como ser uma solução pronta, altamente personalizável (boa parte de seu código é *open-source*) e de baixo custo, se comparada a outras soluções no mercado.

Aplicações Android utilizam, em sua maioria, Java como linguagem de programação. A compilação do código é feita por ferramentas do Android SDK, que geram um pacote APK, contendo todas as informações relativas à aplicação Android, e que é usado na instalação da aplicação.

No funcionamento de um aplicativo, seu acesso a recursos é bastante restrito, utilizando apenas os recursos que necessita. Isso ocorre porque cada aplicação é iniciada em sua própria Sandbox (ferramenta que isola a execução de um programa e seus processos, com o fim de testá-lo em um ambiente seguro). Mais especificamente, o SO Android se baseia num sistema Linux multiusuário (cada aplicativo é um usuário), e cada processo é executado em sua própria instância de máquina virtual (VM), o que isola as aplicações entre si. Assim, a aplicação não consegue acessar partes do sistema para as quais não tem permissão. Para casos específicos, onde aplicações diferentes precisam compartilhar recursos, essa necessidade precisa ser explicitada e configurada, para não comprometer o bom funcionamento do sistema (DEVELOPERS, 2017b).

2.1.1 ARQUITETURA

Considerando que a arquitetura Android é uma pilha de *software* com base em Linux, e fazendo uso de uma abordagem *top-down*, tem-se:

Camada de Aplicativos do Sistema: camada na qual se situam os aplicativos a serem executados. O Android possui nativamente aplicações instaladas para, por exemplo,

navegador de internet, calendário, envio de e-mail, SMS, dentre outros possíveis usos. Entretanto, as aplicações nativas não possuem privilégios sobre aplicações instaladas pelo usuário (é possível utilizar outro navegador de internet, por exemplo). Essas aplicações também podem ser invocadas de dentro de aplicações terceirizadas, facilitando o trabalho do desenvolvedor (por exemplo, caso a aplicação precise enviar um e-mail, não é necessário programar a funcionalidade, pois ela já existe nativamente).

Camada Java API Framework: camada ligada à reutilização de componentes por parte dos desenvolvedores. As APIs programadas em Java provêem os recursos (blocos de programação) necessários para uma programação mais simplificada, do ponto de vista da modularidade.

Camada de Bibliotecas C/C++ nativas: camada relacionada a bibliotecas C/C++ que são utilizadas por serviços do sistema Android implementados por código nativo. As Java *Framework* APIs fornecem as funcionalidades dessas bibliotecas às aplicações (desenho e manipulação de gráficos 2D e 3D, por exemplo).

Camada Android Runtime: para versões Android anteriores à 5.0, cada aplicação utilizava uma instância própria da Dalvik VM. A partir da versão Android 5.0, as aplicações utilizam instâncias próprias do Android Runtime (ART), que é projetado para funcionar com consumo mínimo de memória. Dentre as melhorias com a adoção do ART, destacam-se a compilação *ahead-of-time* (AOT) e *just-in-time* (JIT) e a coleta de lixo otimizada.

Camada de abstração de hardware: a HAL funciona como uma interface entre o *hardware* e a Java API *Framework*. Na prática, quando uma API necessita utilizar algum componente de *hardware*, é feita uma chamada para esse componente. A HAL possui módulos de biblioteca relativos às funcionalidades dos componentes *hardware*. Após essa chamada, o módulo correspondente é carregado e utilizado pela API.

Camada Kernel Linux: base da plataforma Android. Além de prover acesso a inúmeros mecanismos de segurança, também é responsável pelo gerenciamento de memória em baixo nível, gerenciamento de *drivers*, dentre outras funções. O fato de se utilizar um kernel Linux torna o desenvolvimento de *drivers* mais consistente por parte dos fabricantes, pois estarão fazendo-o para um kernel conhecido (DEVELOPERS, 2017a).



FIG. 2.1: Arquitetura em camadas da plataforma Android.

Na Figura 2.1 pode-se ter uma visualização da separação das camadas da arquitetura Android.

2.1.2 COMPONENTES DE UMA APLICAÇÃO

Existem quatro tipos de blocos de construção de uma aplicação. Com finalidades, ciclo de vida e modo de funcionamento diferentes, juntos eles definem o comportamento da aplicação de um modo geral.

Atividades: fornecem uma tela para interação com o usuário. Durante o uso de uma aplicação, é possível invocar outra atividade a partir da atual. Ao se iniciar uma nova atividade (uma nova tela), a atividade anterior é armazenada numa pilha de retorno. Ao se encerrar uma atividade, ela é destruída na pilha e a anterior é retomada (modo de funcionamento LIFO).

Serviços: componentes executados em *background*, geralmente para processos de execução longa (*download* de arquivos e reprodução de músicas em segundo plano, por exemplo) e processos remotos, sem haver de fato uma interface de usuário.

Provedores de conteúdo: responsáveis pelo armazenamento e disponibilização de dados para as aplicações. Fazem a gerência desse conjunto compartilhado de dados, armazenando-os em um SGBD SQLite ou em locais de armazenamento persistente acessíveis à aplicação.

Caso seja permitido pelo provedor, outras aplicações podem acessar e/ou modificar esses dados.

Receptores de transmissão: componentes responsáveis pelo recebimento e tratamento de eventos provenientes do sistema e de outras aplicações. Também chamados de Receptores de *Broadcast*, eles respondem a anúncios de transmissão por todo o sistema. Alguns exemplos de transmissões vindas do sistema são avisos de bateria baixa, capturas de tela efetuadas, dentre outras (DEVELOPERS, 2017b), (YAGHMOUR, 2013). Aplicações podem, por exemplo, anunciar o *download* de dados necessários para seu funcionamento.

2.1.2.1 ATIVAÇÃO DE COMPONENTES

Quando se deseja ativar um componente, devem ser utilizados *intents*, mensagens assíncronas que requisitam ações a componentes, não necessariamente da mesma aplicação. *Intents* podem requisitar três dos quatro componentes de aplicação. Para atividades e serviços, o *intent* possui o nome da ação requisitada, enquanto possui o nome da transmissão a ser feita para receptores de transmissão.

Além disso, caso um componente queira apenas *intents* específicos, a aplicação pode criar filtros para receber apenas os *intents* desejados. Para isso, é necessário definir essa lista de filtros. Isso é feito no arquivo *AndroidManifest.xml* (DEVELOPERS, 2017c).

2.1.3 MANIFESTO DA APLICAÇÃO

O arquivo de manifesto (*AndroidManifest.xml*) é primordial para o funcionamento da aplicação, pois provê informações essenciais sobre a aplicação ao sistema Android. Dentro as funções do manifesto, destacam-se a descrição dos componentes da aplicação (ver figura 2.2), declaração de permissões de acesso a áreas protegidas da API e permissões de interação com outras aplicações, listar bibliotecas vinculadas à aplicação, informar o nível mínimo da API para o funcionamento da aplicação, dentre outras. O arquivo deve estar armazenado na pasta raiz do projeto Android (DEVELOPERS, 2017d).

2.1.4 PERMISSÕES

Um dos principais aspectos que contribuem para o bom funcionamento do sistema Android é a camada de segurança da plataforma. Ela isola as aplicações entre si, e restringe sua utilização de recursos, afim de não ferir a privacidade de outras aplicações.

Por definição, as aplicações não possuem permissão para utilizar recursos do sistema que possam causar problemas ao SO ou à experiência do usuário. Desse modo, todo

```

1<?xml version="1.0" encoding="utf-8"?>
2<manifest xmlns:android="http://schemas.android.com/apk/res/android"
3    package="com.androidapp.basicElements"
4    android:versionCode="1"
5    android:versionName="1.0">
6    <application android:icon="@drawable/icon" android:label="@string/app_name">
7        <activity android:name=".BasicElements"
8            android:label="@string/app_name">
9            <intent-filter>
10                <action android:name="android.intent.action.MAIN" />
11                <category android:name="android.intent.category.LAUNCHER" />
12            </intent-filter>
13        </activity>
14
15    </application>
16    <uses-sdk android:minSdkVersion="2" />
17
18</manifest>

```

FIG. 2.2: Exemplo de arquivo *AndroidManifest.xml*.

recurso do dispositivo que a aplicação tenha que usar, como acesso a recursos de *hardware*, devem ser estaticamente declarados no arquivo *AndroidManifest.xml*, e estas permissões são apresentadas ao usuário durante a instalação (DEVELOPERS, 2017e).

2.1.4.1 ESPECIFICIDADES DO ANDROID 6.0

Foi escolhida a versão Marshmallow (Android 6.0) para o desenvolvimento deste projeto. Se comparada à sua versão anterior (Android Lollipop), várias melhorias foram implementadas, dentre elas: um melhor gerenciamento de energia; suporte nativo para impressões digitais; melhorias no compartilhamento (RAKOWSKI, 2015). Porém, a mudança que mais provoca questionamentos quanto ao desenvolvimento de aplicações é a nova política de permissões de aplicações.

Até a versão Lollipop, as permissões deveriam ser aceitas pelo usuário no momento da instalação do aplicativo. A partir da versão Marshmallow, isso não ocorrerá mais. Algumas permissões serão requisitadas ao usuário no momento da execução do aplicativo. O usuário poderá, se preferir, desabilitar algumas dessas permissões. Caso determinada permissão seja indispensável ao funcionamento do aplicativo, o usuário será informado dessa necessidade.

As permissões foram divididas em 2 grupos. O grupo de permissões normais não necessita de aprovação do usuário, pois supostamente não oferecem ameaças à privacidade. O grupo de permissões perigosas é assim chamado porque trabalha com dados privados do usuário. Portanto, essas permissões deverão ser explicitamente concedidas. Para este

projeto, algumas das permissões perigosas que possivelmente serão necessárias são: gravar áudio (ver seção 2.3); leitura de dados externos e escrita de dados externos, dado que o projeto propõe uma transferência de arquivo (DEVELOPERS, 2017e).

2.2 ARQUITETURA *PUBLISH/SUBSCRIBE*

São arquiteturas onde os nós se comunicam, indiretamente, por troca de mensagens, onde quem envia a mensagem por um método *publish* é chamado de publicador e quem recebe a mensagem por meio do método *subscribe* é chamado de receptor.

O publicador publica a mensagem para um servidor ou *middleware* que armazena as mensagens e as associa a identificadores. A partir desse ponto, a mensagem pode ser recuperada por qualquer receptor que se associe ao conjunto de mensagens determinado pelo identificador. Quem posteriormente se associar ao conjunto de mensagens poderá receber as mensagens anteriores válidas desse conjunto.

Esse tipo de arquitetura oferece vantagens como:

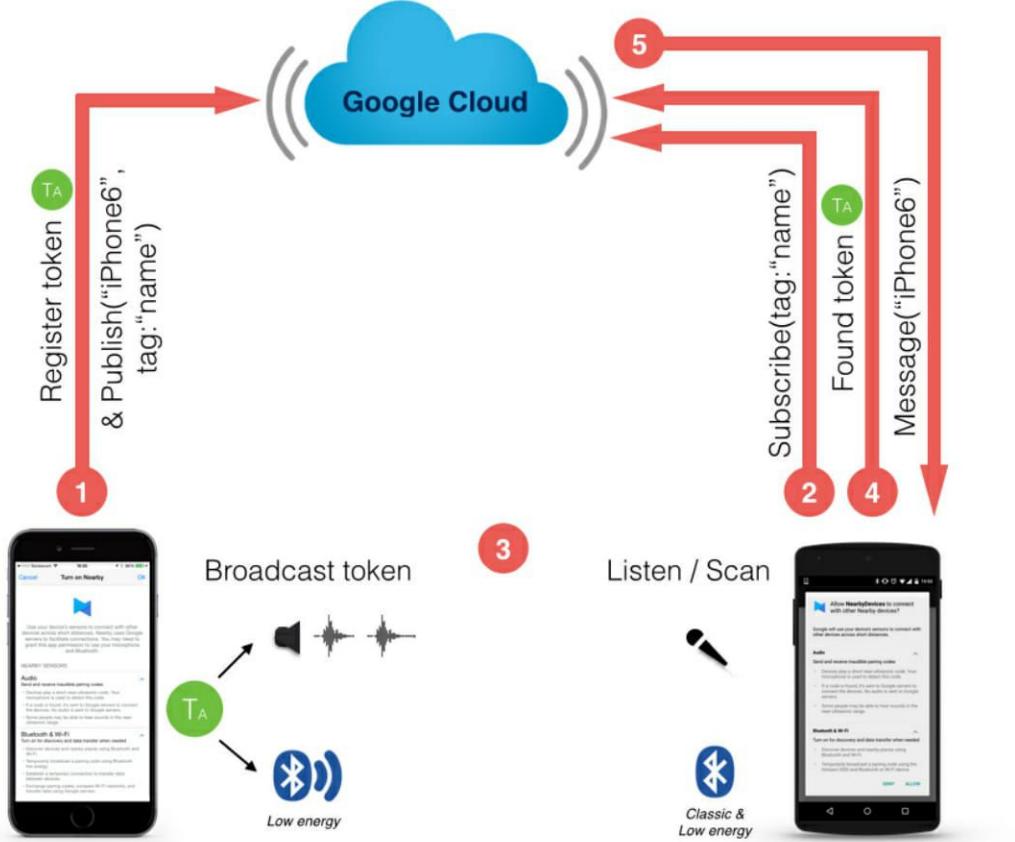
Escalabilidade: possibilita o crescimento do sistema mantendo sua performance estável, pois possibilita maior paralelismo das operações e outras formas de roteamento de mensagens; e

Desacoplamento: O publicador não tem informação nenhuma do receptor da mensagem, nem se existe receptor para a mensagem. O funcionamento do sistema está ligado à sintaxe das mensagens e independe do sistema operacional e do *hardware* utilizados pelos nós.

2.3 API NEARBY MESSAGES

A Google anunciou oficialmente a API Nearby Messages em 13 de agosto de 2015 (HYTTSEN, 2015). O *Nearby Messages* é uma API multiplataforma cuja finalidade é encontrar e se comunicar com dispositivos e *beacons*, baseado na proximidade. Informações sensíveis, como localização exata, não são divulgadas, e a única informação disponibilizada pela API é a lista de usuários próximos, por motivos de segurança e privacidade dos próprios usuários. A estrutura simples de *publish* e *subscribe* para troca de mensagens, definidas como pequenos *payloads* binários (DEVELOPERS, 2016), favorece aplicações *real-time* e funciona com uma combinação de Bluetooth, *Wi-Fi*, e áudio ultrasom para se conectar a outros dispositivos (ver figura 2.3).

Wi-fi: As publicações são feitas ao servidor do *Nearby Messages*, que recebe o *token* e a mensagem pelo método *publish* e que identifica a mensagem pelo *token*, para que outros



How Google Nearby.Messages works?

FIG. 2.3: Funcionamento da API *Nearby Messages* (LEGENDRE, 2015)

dispositivos possam acessar as mensagens.

Bluetooth e Áudio Ultrassom: O envio de *tokens* é feito por esses dois canais. Quando um dispositivo identifica um *token* de algum dispositivo próximo, ele acusa o recebimento do token ao servidor do *Nearby Messages*, para validar o *token* e receber quaisquer mensagens publicadas associadas ao *token* enviado.

2.3.1 BEACON

Os *beacons* BLE interagem com o *Nearby Messages* e com a *Google Beacon Platform* para se comunicar com dispositivos próximos. Essas tecnologias estão abrindo inúmeras possibilidades no marketing de proximidade. *Beacons* utilizam a API *Nearby* com o formato *Eddystone* para comunicação entre várias plataformas.

2.3.1.1 FORMATO EDDYSTONE

É um protocolo bluetooth 4.0 desenvolvido pela Google que define um formato para mensagens enviadas por *beacons* BLE. Nele são definidos alguns *frames* que podem ser

usados individualmente ou em conjunto para diversos tipos de aplicações:

Eddystone-UID: Um identificador único com separação por *namespace* para identificar um único beacon ou uma rede de beacons.

Eddystone-URL: Uma URL comprimida que pode ser usada para expor algum recurso ou sistema.

Eddystone-TLM: Envia status do beacon, como energia, que pode ser usada para manutenção de uma rede de beacons.

Eddystone-EID: Um frame criptografado e variável com o tempo, usado por motivos de segurança.

Junto com o *Eddystone* a Google desenvolveu o *Google Beacon Platform* e a *Proximity Beacon API*, que é usada para gerenciar e manter *beacons* BLE cadastrados na plataforma. A *Proximity Beacon API* pode ser usada para gerenciar os *attachments*, textos de até 1024 *bytes*, que são enviados para outros dispositivos através da API *Nearby Messages*.

2.4 API NEARBY CONNECTIONS

O *Nearby Connections* foi concebido para realizar conexão *peer-to-peer* criptografada de alta banda e baixa latência entre dispositivos mobile. Do mesmo modo que o *Nearby Messages*, o *Connections* se utiliza de BLE e de *Wifi Hotspots* para estabelecer uma conexão com outro dispositivo, porém não utiliza áudio ultrassom.

Dentre as principais diferenças em relação ao *Nearby Messages*, destacam-se o fato da possibilidade de transferência direta de arquivos entre os dispositivos sem a necessidade de uma conexão à internet, além da conexão em modo um-para-muitos (no *Messages*, todos os dispositivos fazem *broadcast* e atuam como receptores).

Ele também facilita o descobrimento, conexão e passagem de dados, além de fornecer uma forma segura e completamente *offline* de comunicação entre dispositivos. Junto a isso, também desempenha muito bem em aplicações *real-time*, o que a leva a ser muito utilizada em jogos *multiplayer* e aplicativos com atividades colaborativas (DEVELOPERS, 2017f).

2.5 REST

Roy Fielding, como parte de seu doutorado, definiu os princípios da estrutura *Web* e introduziu o *Representational State Transfer*, ou REST, como um estilo de arquitetura que permite, através de um conjunto limitado de operações com semântica uniforme, construir

qualquer tipo de aplicação. REST, como definido por Fielding, descreve a internet como uma aplicação distribuída onde os recursos se comunicam por troca de representações de estado, o que justifica o nome REST (WEBBER et al., 2010).

2.5.1 REST API

Um serviço ou aplicação que segue as definições de uma estrutura REST é denominado *RESTful* e, consequentemente, uma REST API é uma API *RESTful*. Uma REST API permite chamadas através do protocolo HTTP para acessar funcionalidades external à aplicação.

2.6 DJANGO FRAMEWORK

Django é uma *Web framework* (*opensource* e grátis) para Python que foca em auxiliar o desenvolvimento rápido e bem organizado de aplicações *Web*. Na framework um projeto é o produto final e ele pode conter vários aplicativos que são usados pra separar a implementação de partes diferentes do projeto.

Dentro de um projeto Django os aplicativos são separados em diretórios com suas *Views*, *Models* e outras configurações do aplicativo e num diretório do projeto estão as configurações gerais e comuns entre os aplicativos.

O Django também oferece um script *manage.py* para debugar, desenvolver, interagir com o banco de dados (usando a linguagem Python) e enviar comandos ao projeto. Devido à grande comunidade do Django existem muitos módulos para a framework, facilitando o desenvolvimento de aplicações.

3 MODELAGEM

3.1 APLICAÇÃO MOBILE

Após compreender as possibilidades oferecidas pela API *Nearby*, decidiu-se quais delas seriam exploradas, e nisso baseou-se a modelagem da aplicação. Assim, as funcionalidades a serem oferecidas ao usuário final são:

- Transferência de arquivos entre dois dispositivos próximo - uso do *Nearby Messages*; e
- Disponibilização de quadros de avisos de acordo com a localização do usuário - uso da combinação de *Beacons* com a API *Nearby*.

Dessa forma, o projeto foi estruturado em quatro módulos distintos:

- *Activities*;
- djangoAPI;
- *NearbyWrapper*; e
- Classes auxiliares.

3.1.1 ACTIVITIES

Como em qualquer aplicação *Android*, o desenvolvimento das *activities* é extremamente importante, pois são elas que formam a experiência visual do usuário com o aplicativo. De acordo com a modelagem desejada da aplicação, foram desenvolvidas as seguintes *activities*:

- *MainActivity* - tela inicial do aplicativo, onde é feita a identificação do usuário;
- *HomeActivity* - tela inicial de um usuário autenticado, contendo botões referentes às funcionalidades desejadas na modelagem;
- *TransferActivity* - tela para transferência de arquivos. Oferece as opções de envio ou recebimento;

- *ListActivity* - escolhida a opção de envio, é a tela onde os arquivos disponíveis para envio são disponibilizados;
- *Receive* - escolhida a opção de recebimento, é a tela onde o usuário verifica os usuários que desejam lhe enviar um arquivo; e
- *beaconActivity* - tela que mostra ao usuário os quadros de avisos disponíveis naquele local.

3.1.2 DJANGOAPI

Nesse módulo, foram desenvolvidas as classes responsáveis pela interação do aplicativo com o servidor Django. Cada classe realiza uma tarefa específica de vital importância para o funcionamento da aplicação:

- *loginAPI* - enviar os dados de *login* e senha ao servidor, de modo a autenticar a sessão;
- *uploadAPI* - enviar um arquivo a ser armazenado no servidor, além de uma senha de acesso ao arquivo;
- *downloadAPI* - enviar informações que identifiquem o arquivo a ser recebido, e retorná-lo à aplicação;
- *listAPI* - requisitar uma lista de todos os arquivos pertencentes ao usuário em questão; e
- *beaconAPI* - enviar a URL recebida de um beacon, de modo a receber uma lista com os quadros de aviso existentes naquele local.

3.1.3 NEARBYWRAPPER

Após os estudos teóricos feitos sobre a API *Nearby*, verificou-se que seria necessária a criação de classes e estruturas de dados que tornassem a implementação das funcionalidades da API mais simples para o desenvolvedor. Assim, de acordo com as necessidades previstas na modelagem da aplicação, foram desenvolvidos quatro arquivos *.java*, que serão abordados no próximo capítulo deste relatório:

- enum *MessageType*;
- classe *NearbyAPIActivity*;

- classe *NearbyAPIWrapper*; e
- classe *NearbyMessage*.

3.1.4 CLASSES AUXILIARES

Percebeu-se que, durante o desenvolvimento da aplicação, novas necessidades poderiam surgir e que não necessariamente elas seriam atendidas apenas pelas classes previstas na modelagem inicial do problema. Assim, três classes foram desenvolvidas de acordo com necessidades específicas, listadas abaixo:

- *AsyncResponse.java* - interface utilizada para disponibilizar o resultado de uma *AsyncTask* para uma *activity*;
- *FileUtility.java* - classe criada para auxiliar a implementação de um método de obtenção do caminho de acesso a um arquivo; e
- *MyApplication.java* - classe implementada para permitir a criação de variáveis globais à aplicação.

3.2 SERVIDOR

Foi necessário o desenvolvimento de um servidor exclusivo para a aplicação, dado que algumas funcionalidades essenciais não podem ser desenvolvidas no *Android*. O projeto Django foi dividido em 2 aplicativos Django e um banco de dados, sendo usadas várias bibliotecas e softwares *open source* para Django e *Python*.

3.2.1 BANCO DE DADOS

Todas as tabelas que não foram geradas automaticamente por bibliotecas nativas do Django possuem campos relativos ao *timestamp* do momento de criação e última alteração de cada entrada.

3.2.1.1 DJANGO MODELS

Foram utilizados pacotes *open source* (*django.contrib.auth* e *rest_framework.authtoken*) para gerar automaticamente três tabelas no banco de dados:

- *User* - Tabela de usuários;

- *Group* - Grupos dos usuários para segregação e controle de acesso; e
- *Auth Token* - Tokens para autenticação do usuário numa chamada HTTP pelo aplicativo.

3.2.1.2 API MODELS

- *FileModel* - Arquivos salvos no servidor para permitir a transferência pelo aplicativo;
- *BeaconModel* - Modelo de *beacons* no banco de dados; e
- *LocationModel* - Tabela de localização de usuário por *beacon* e horário.

3.2.1.3 BOARDS MODELS

- *Board* - Modelo do quadro de avisos;
- *Notification* - Modelo do aviso; e
- *NotificationFile* - Arquivos dos avisos.

3.2.2 REST API

A API foi criada como um aplicativo Django para permitir a comunicação entre o aplicativo e o servidor.

A API é composta de 4 chamadas:

- *Upload* - Recebimento de um arquivo e uma senha (é salva como hash de sha512 junto com um salt aleatório), retornando um código associado ao arquivo;
- *Download* - Recebimento de um código e uma senha. Se um arquivo ligado ao código existir e a senha estiver correta, são devolvidos os dados necessários para realizar o download do arquivo;
- *Listagem* - O usuário é identificado pelo token recebido e uma lista dos arquivos do usuário é devolvida; e
- *Beacon* - Retorna os quadros de avisos que um beacon pode compartilhar, além de gravar um *log* do acesso do usuário ao beacon na tabela *Location*.

3.2.2.1 LOGIN

Views e tabelas geradas automaticamente pelo *rest_framework* para passagem do *token* de autenticação à usuários autorizados (por *login* e senha).

3.2.2.2 TROCA DE ARQUIVOS

Os arquivos são definidos nos *models* com o *DocumentField*, que salva uma referência no banco de dados para o arquivo que é salvo no sistema e servido estaticamente. Os arquivos são salvos com o nome alterado para um código UUID, para evitar o reconhecimento do teor do arquivo. Também é gerado um código UUID de referência (para identificar o arquivo na *view* de *download*) e outro para o *salt* da senha.

Para o *upload* do arquivo, é necessário que sejam enviados o arquivo e a senha, com os outros campos sendo gerados automaticamente.

Para o *download*, é necessário o envio do código e da senha do arquivo. Uma vez que a senha foi verificada, é devolvida a URL de acesso ao arquivo.

3.2.2.3 BEACON

As URLs próprias de cada *beacon* contém o código do *beacon*, que é usado como referência para a montagem da resposta à requisição, que contém as URLs dos quadros de avisos aos quais o *beacon* está associado, junto com os respectivos nomes dos quadros.

Os *beacons* não dão acesso a todos os quadros de avisos pois, por questões de segurança, alguns quadros podem ser acessados somente em locais físicos específicos.

3.2.3 QUADRO DE AVISOS

Os quadros de avisos foram feitos em um aplicativo Django separado (*boards*) por questões de organização, e para demonstrar que os *beacons* podem ser usados para redirecionar o usuário para outra aplicação WEB.

Composto de duas *views*:

- *Board* - Quadro de avisos; e
- *Notification* - Avisos.

As duas *views* verificam se o usuário está autenticado (se não é acesso anônimo) e se possui autorização para acessar os dados, baseado nos grupos do usuário, pois cada *board* possui um conjunto de grupos que estão autorizados a acessá-lo.

4 DESENVOLVIMENTO

4.1 SERVIDOR

Um aplicativo Django é feito em módulos, onde cada módulo tem seu objetivo. A seguir, estão os módulos usados neste projeto e suas funcionalidades.

- *models.py* - Modelagem das tabelas da aplicação;
- *views.py* - Definição das *views* da aplicação;
- *urls.py* - Definição das URLs e associação com as *views*;
- *admin.py* - Definição das ações relativas ao aplicativo na página de administrador Django; e
- *serializers.py*- Classes para auxiliar a conversão dos dados nas chamadas HTTP.

O servidor foi dividido em dois aplicativos: *api* e *boards*, onde cada um possui seu diretório. Há também um diretório do projeto, onde ficam as configurações do projeto.

4.1.1 API

4.1.1.1 VIEWS.PY

- *FileList*
 - Recebe chamadas GET.
 - Autenticação por token.
 - Retorna em formato *JSON* a lista de arquivos do usuário.
 - *FileModel*(leitura).
- *FileUpload*
 - Recebe chamadas POST.
 - Autenticação por token.
 - *Serializer* : FileUploadSerializer

- *Input* : Um arquivo e uma senha enviados em 'multi-part/form-data'.

A senha é salva em *sha512* com um *salt* aleatório em UUID modificado. A implementação dessa função de *hash* foi feita com a biblioteca nativa de Python *hashlib*, e está no módulo *pfcserver.utils.hash*.

- *Output* : *JSON string* com o código do arquivo criado ou os respectivos erros de autenticação e validação.
- *FileModel*(leitura e criação).

- *File Download*

- Recebe chamadas POST;
- Autenticação por token;
- *Serializer* : FileDownLoadSerializer;
- *Input* : Um código e uma senha enviados em 'application/json';
- *Output* : *JSON string* com a url do arquivo e o nome original do arquivo, caso o teste de senha tenha retornado positivo, e caso contrário, retorna os respectivos erros; e
- *FileModel*(leitura).

- *BeaconBoards*

- Recebe chamadas GET.
- Autenticação por token.
- *Input* : beacon_id retirado da URL de chamada.
- Após autenticação do usuário, uma entrada em *LocationModel* com o usuário, *beacon* acessado e o timestamp é criada.

Como o *scan bluetooth* fica ativado em plano de fundo e os *beacons* podem ter um período curto de envio (de 0.1 a 10 segundos por pacote), pode-se acompanhar por onde o usuário esteve dentro da rede de *beacons*.

Com essas informações é possível buscar padrões e identificar falhas de segurança pelo comportamento do usuários.

- *Output* : *JSON string* com a lista de Boards que o *beacon* pode compartilhar.
- *BeaconModel*(leitura),*LocationModel*(criação).

4.1.1.2 MODELS.PY

- *FileModel*
- *BeaconModel*
- *LocationModel*

4.1.1.3 SERIALIZERS.PY

- *FileUploadSerializer*
- *FileDownLoadSerializer*

4.1.2 BOARDS

4.1.2.1 VIEWS.PY

- *BoardView*
 - Recebe chamadas GET.
 - Autenticação por sessão.
 - *Input* : board_id retirado da url de chamada.
É feito um teste pra ver se o usuário faz parte de algum grupo que tem acesso ao quadro. Este teste é feito com a função *has_group* definida no módulo <textitpfcserver.utils.auth>.
 - *Output* : Página do quadro de aviso.
 - *Board*(leitura).
- *NotificationView*
 - Recebe chamadas GET.
 - Autenticação por sessão.
 - *Input* : board_id retirado da URL de chamada.
É feita uma checagem para verificar se o usuário tem acesso ao *board* ao qual o aviso pertence(com a função *_group*).
 - *Output* : Página do aviso com links para o *download* dos arquivos.
 - *Board*(leitura), *Notification*(leitura), *NotificationFile*(leitura).

4.1.2.2 MODELS.PY

- *Board*
- *Notification*
- *NotificationFile*

4.2 APLICAÇÃO MOBILE

4.2.1 LOGIN DO USUÁRIO

O usuário deve fazer o *login* com suas informações de acesso (usuário e senha) e ser direcionado para a tela inicial da aplicação. Essa interação é representada na *MainActivity*. Entretanto, antes da implementação dessa funcionalidade, foram necessários alguns ajustes iniciais para o bom funcionamento do aplicativo.

Durante o desenvolvimento, foi necessária a utilização da função *runOnUiThread()*, pois ela permite a atualização da interface de usuário a partir de um contexto assíncrono. Nativamente, o *Android* não permite a utilização dessa função. Para resolver esse problema, foi definida uma política do uso de *threads*, explicitada na figura 4.2.

Além disso, de acordo com as novas políticas sobre permissões das versões mais recentes do *Android*, as permissões perigosas devem ser requisitadas na inicialização da aplicação, caso elas ainda não tenham sido dadas. Isso acontecerá apenas na primeira vez que o aplicativo seja usado, ou após uma atualização. Essa requisição ao usuário também está explicitada na figura 4.2.

Após as verificações iniciais, o aplicativo é iniciado, apresentando a tela de *login*. A interação com o servidor, nesse caso, prevê que sejam enviadas pela aplicação as informações fornecidas pelo usuário. Ao receber o nome de usuário e a respectiva senha, o servidor retorna o *token* associado ao usuário, que deverá ser utilizado pela aplicação nas próximas interações com o servidor, como modo de autenticação.

Para a validação das credenciais e posterior obtenção do token associado, foi utilizada a classe *loginAPI.java*. Nela, é implementado o envio das credenciais por meio de uma requisição HTTP POST, cuja resposta é um objeto JSON que contém o valor do *token* associado. Após isso, o resultado é disponibilizado à *activity* como uma *string*, que será convertido em um objeto JSON, para que o valor do *token* possa ser acessado e armazenado. Nesse momento é instanciado um objeto da classe *MyApplication.java*. Por meio dela, esse valor será armazenado em uma variável global do aplicativo, de modo que possa

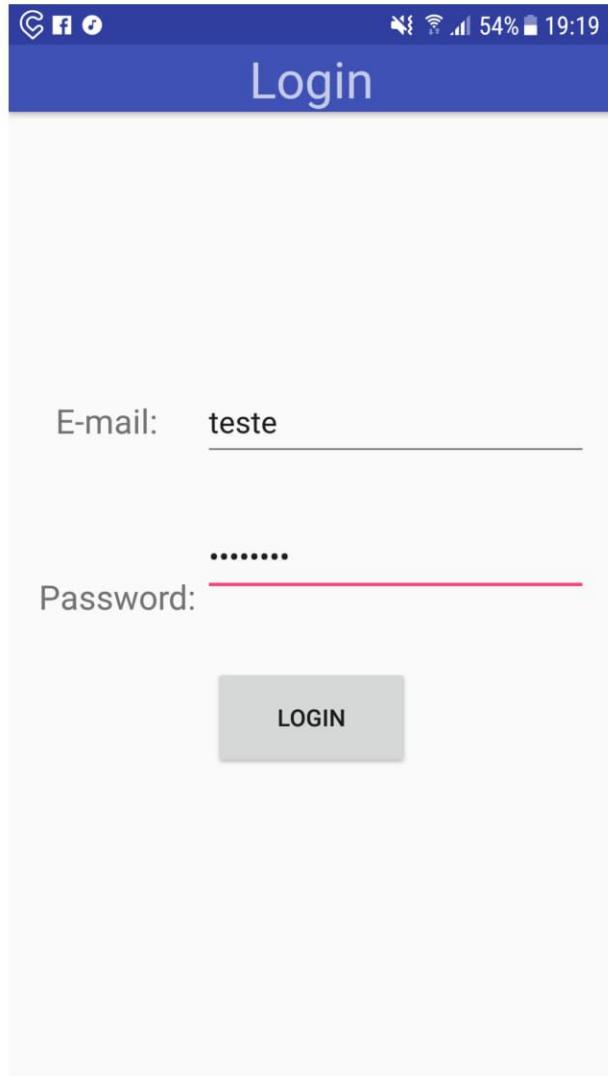


FIG. 4.1: Tela de *login*

ser utilizada nas *activities* subsequentes. Isso é possível porque *MyApplication.java* é subclasse de *Application*, permitindo seu uso em todas as *activities* daquela sessão depois que um objeto da classe é instanciado.

Caso tenham sido dadas credenciais corretas, a aplicação apresentará sua tela inicial, representada na *HomeActivity*. Caso as credenciais estejam incorretas, é disparado um aviso na tela (*Invalid Credentials*).

4.2.2 TROCA DE ARQUIVOS

Para esta etapa, foi necessária a implementação de cinco funcionalidades básicas, de acordo com as seguintes ações:

- Remetente deve fazer o *upload* do arquivo desejado para o servidor da aplicação;

```

// permite o uso da função runOnUiThread()

if (android.os.Build.VERSION.SDK_INT > 17) {
    StrictMode.ThreadPolicy policy = new StrictMode.ThreadPolicy.Builder().permitAll().build();
    StrictMode.setThreadPolicy(policy);
}

if (ContextCompat.checkSelfPermission(this,
        Manifest.permission.READ_EXTERNAL_STORAGE)
    != PackageManager.PERMISSION_GRANTED) {

    // Should we show an explanation?
    if (ActivityCompat.shouldShowRequestPermissionRationale(this,
        Manifest.permission.READ_EXTERNAL_STORAGE)) {

        // Show an explanation to the user *asynchronously* -- don't block
        // this thread waiting for the user's response! After the user
        // sees the explanation, try again to request the permission.

    } else {

        // No explanation needed, we can request the permission.

        ActivityCompat.requestPermissions(this,
            new String[]{Manifest.permission.READ_EXTERNAL_STORAGE},
            3);
    }
}

```

FIG. 4.2: Política de *threads* e exemplo de requisição de permissão

- Remetente deve acessar lista dos seus arquivos no servidor e escolher o arquivo desejado;
- Remetente deve enviar aos destinatários informações que identifiquem o arquivo;
- Destinatários devem receber e armazenar esses dados; e
- Destinatários, em posse dessas informações, contatam o servidor e fazem o download do arquivo enviado.

A figura 4.3 mostra a estrutura básica do funcionamento da troca de arquivos entre dois dispositivos. Em algumas funcionalidades, será necessária a interação com o servidor da aplicação, dentro de uma rede interna, por fins de segurança. Em outras, onde é necessário o uso do *Nearby Messages*, será necessária a interação com os servidores da *Google*.

4.2.2.1 UPLOAD

Após iniciar a *HomeActivity* e pressionar o botão *Upload*, um *intent* é ativado, apresentando ao usuário um *filechooser* para que ele selecione o arquivo a ser enviado ao servidor. Após a escolha do arquivo, é apresentada a opção de proteger o arquivo com senha, que será armazenada junto ao arquivo.

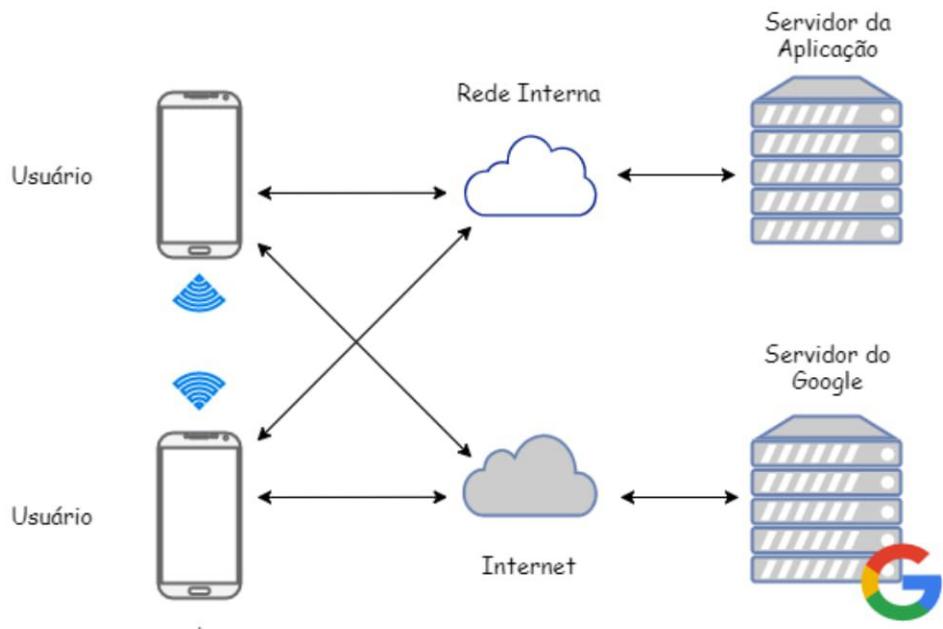


FIG. 4.3: Estrutura básica da troca de arquivos

É necessária uma observação quanto à escolha de arquivos. O gerenciador de arquivos padrão do Android não permite acesso ao caminho dos arquivos que estejam fora da pasta "Galeria". Desse modo, é necessário que o celular possua um gerenciador de arquivos próprio instalado, de modo a prover acesso a todos os arquivos. Sem acesso ao caminho do arquivo, não será possível fazer o *upload* (exemplo na figura 4.4).

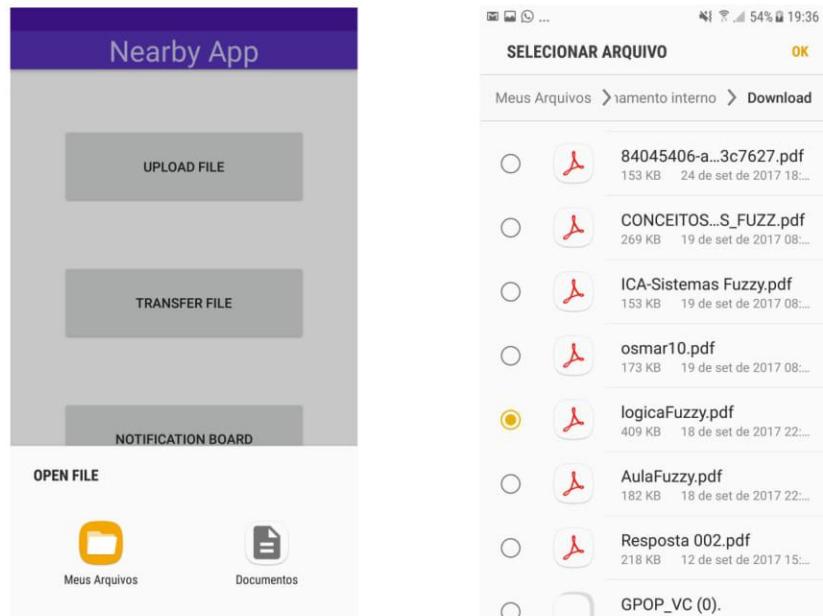


FIG. 4.4: Escolha do arquivo com *Samsung File Manager*

Para a escolha de arquivos, era necessário que o caminho para o arquivo dentro do dispositivo fosse passado à *activity*, de modo a localizar o arquivo a ser enviado. Entretanto, o *filechooser* não retorna diretamente o caminho para o arquivo, mas uma URI. Desse modo, foi implementada a classe *FileUtility.java*, que possui o método *getPath()*. Ele recebe como parâmetros um contexto e URI, e retorna uma *string* contendo o caminho para o arquivo.

Após a definição da senha, ocorre o *upload* do arquivo. Para isso, foi desenvolvida a classe *uploadAPI*, que provê toda a interação com a Django API relacionada ao recebimento de arquivos, localizada no servidor. Para que essa carga de processamento não interrompesse a *activity*, a classe foi definida como subclasse de *android.os.AsyncTask*, de modo a realizar o processamento em *background* (sobrecrevendo o método *doInBackground()*). No seu uso, instancia-se um objeto da classe, contendo como atributos o caminho para o arquivo, a senha escolhida e o *token* de usuário, para que seja verificada a identidade do usuário que está fazendo o *upload*. Dessa forma, o *upload* é feito a partir de uma requisição HTTP POST, com o envio do *token* no cabeçalho da requisição, a senha e o arquivo no corpo da mensagem.

Após o processamento do arquivo recebido no servidor, o arquivo é salvo, e é retornado um código associado a ele. Cada arquivo armazenado possui um código, e cada código está associado a apenas um arquivo. Dentro do escopo dessa funcionalidade, a resposta não será utilizada. Porém, pensando sobre atualizações futuras, foi sobrecrevendo o método *processFinish()* da interface *AsyncResponse.java*, que permite o uso da resposta à requisição na *activity*. Essa ferramenta será melhor analisada na próxima seção.

Um dos problemas dessa abordagem é o recebimento da resposta à requisição junto à classe *uploadAPI.java*, e não diretamente na *activity*. Foi adotada como solução o uso de uma interface como canal de comunicação entre *uploadAPI.java* e a *activity*, chamada de *AsyncResponse*. Desse modo, a *activity* é definida também como uma implementação de *AsyncResponse*, com a sobreulação do método *processFinish()* da interface. A passagem do resultado para a *activity* ocorre quando o método *processFinish()* é chamado na classe *uploadAPI*, utilizando como parâmetro a resposta à requisição. Assim, essa informação pode ser utilizada no contexto da *activity*, podendo ser manipulada dentro do método sobrecrevendo *processFinish()*.

4.2.2.2 LISTAGEM DE ARQUIVOS

Dado que cada arquivo está associado a um código único, os destinatários precisam dessa informação para identificar o arquivo a ser recebido do servidor. Assim, foi necessária a implementação da comunicação entre dispositivos via *Nearby Messages*.

Inicialmente, o usuário que deseja enviar um arquivo deve escolhê-lo dentre os arquivos armazenados no servidor. Para mostrar a lista dos arquivos presentes no servidor ao usuário, foi desenvolvida a classe *listAPI*. Nela é definida uma requisição HTTP GET, enviando no cabeçalho o *token* de usuário e sem parâmetros adicionais. Com o *token*, o servidor verifica os arquivos associados ao respectivo usuário e retorna um objeto JSON contendo o nome e código de cada arquivo.



FIG. 4.5: Listagem de arquivos

Nesse caso, observou-se a necessidade de manipular o resultado da requisição no

contexto da *activity*, de modo a listar os arquivos presentes ao usuário. Desse modo, foi novamente utilizada a interface *AsyncResponse*.

As classes que interagem com o servidor (*listAPI* inclusa) foram implementadas como subclasses de *android.os.AsyncTask*, para serem utilizadas em *background*, como já explicado na seção anterior. Desse modo, a *activity* é definida também como uma implementação de *AsyncResponse*, com a sobrecarga do método *processFinish()* da interface. A passagem do resultado para a *activity* ocorre quando o método *processFinish()* é chamado no método *onPostExecute()* da classe *listAPI.java*, utilizando como parâmetro a resposta à requisição. Assim, essa informação pode ser utilizada no contexto da *activity*, podendo ser manipulada dentro do método sobreescrito *processFinish()*.

Desse modo, é recebida a resposta da requisição na *ListActivity* em uma *string* com formato JSON. Para otimizar o acesso aos dados de cada arquivo, a *string* é convertida em um objeto JSON, e então os dados necessários para montar a lista de arquivos são obtidos. No caso, deve-se mostrar ao usuário o nome de cada arquivo, mantendo o respectivo código oculto para ser utilizado apenas quando necessário. Para tal fim, foi utilizada uma *ListView*, e cada entrada recebeu um objeto da classe *serverFile*, modelado para reunir as informações de cada arquivo. Porém, as entradas de uma *ListView* devem ser do tipo *string*. Para que apenas o nome do arquivo fosse visualizado, foi sobreescrito o método *toString()* de modo a retornar apenas o nome do arquivo.

Após a geração da lista, o usuário deve escolher um arquivo para ser enviado, dentre as opções disponíveis, pressionando o respectivo item da lista. Após pressionar o item, é apresentada uma caixa de diálogo, perguntando se o usuário deseja mesmo enviar o arquivo. Caso o usuário confirme a decisão, o envio das informações do arquivo via *Nearby Messages* é iniciado.

4.2.2.3 ENVIO DO CÓDIGO

Apesar da lista de arquivos mostrar ao usuário apenas o nome de cada arquivo, é possível acessar o código identificador do arquivo, já que foram armazenados objetos da classe *serverFile*. Assim, deve-se primeiramente acessar esse valor e armazená-lo em uma variável.

De modo a simplificar o uso da API e facilitar a organização do código-fonte do programa, foram implementados módulos que encapsulassem as principais estruturas de dados e funções necessários para o funcionamento do *Nearby Messages*. O primeiro deles é *MessageType.java*, um *enum* que tem por objetivo definir os tipos de mensagens que podem trafegar, permitindo tratamentos diferentes para tipos de mensagem

diferentes. Tem-se também a classe abstrata *NearbyAPIActivity.java*, que tem por objetivo implementar as interfaces *GoogleApiClient.ConnectionCallbacks* e *GoogleApiClient.OnConnectionFailedListener* para interagir com o *GoogleApiClient*, necessário para a interação com os servidores da *Google*. Representando-a como subclasse de *AppCompatActivity*, serve como base para qualquer *activity* que precise utilizar o *Nearby Messages*. Na classe *NearbyAPIWrapper.java*, as funcionalidades necessárias do *Nearby Messages* são agrupadas, sendo utilizadas como métodos da classe. Finalmente, na classe *NearbyMessage.java*, é representado o encapsulamento dos dados a serem enviados entre os dispositivos, de modo que sejam utilizados no formato correto pela classe *NearbyAPIWrapper.java*.

Antes da utilização das funcionalidades presentes em *NearbyAPIWrapper.java*, é necessário iniciar a conexão com os servidores da *Google*. Para iniciar a conexão, deve ser criada uma instância da classe *NearbyAPIWrapper*, utilizando os parâmetros de acordo com objetivo da transmissão. O primeiro deles é a própria *activity*, que precisa ser subclasse de *NearbyAPIActivity*. O segundo é estratégia de transmissão a ser utilizada. Como já abordado em seções anteriores, o *Nearby Messages* utiliza a estratégia *Publish/Subscribe*. O último parâmetro é um objeto da classe *MessageListener*, nativa da API. Nela, são definidas as ações a se tomar quando uma mensagem é encontrada ou perdida.

Foi definido o envio dos dados do arquivo no método *ListView.setOnItemClickListener()*, ou seja, ao selecionar um arquivo da lista. Para recuperar os dados do arquivo, o item selecionado é armazenado em uma instância da classe *serverFile*, o que permite o acesso direto ao nome e código identificador do arquivo. Esses dados, junto com o nome do usuário, devem ser enviados ao outro dispositivo. Assim, eles são armazenados em um *HashMap*. Portanto, é possível utilizar o método *newNearbyMessage()* da classe *NearbyMessage*, utilizando como parâmetros o mapa de parâmetros, o tipo da mensagem e um identificador utilizado para diferenciar mensagens que possuam o mesmo *payload* (método *getUUID()*). O retorno dessa função é armazenado em uma variável do tipo *Message*. Após isso, basta utilizar a instância da classe *NearbyAPIWrapper* para publicar a mensagem (método *publish()*).

4.2.2.4 RECEBIMENTO DO CÓDIGO

O usuário que deseja receber um arquivo deve chegar à *ReceiveActivity*. Nela, é disponibilizada a lista de usuários que deseja enviar algum arquivo e que estão nas proximidades. Dessa forma, a estrutura inicial dessa funcionalidade se assemelha à estrutura de envio das mensagens. Também é necessária a criação de uma *ListView* que armazene os dados

recebidos. Uma diferença se baseia no modo de preencher a tabela. Anteriormente, os dados vieram de uma resposta à requisição feita pela *listAPI.java*, enquanto no recebimento do código a atualização da *ListView* é feita na instanciação do objeto da classe *MessageListener*, o que permite a atualização da lista sempre que um novo dispositivo é encontrado. No caso, a lista recebe objetos que contém três parâmetros (nome e código identificador do arquivo, além do nome de usuário do remetente), mas disponibiliza na tela apenas o nome do remetente, utilizando a mesma ideia da listagem de arquivos.

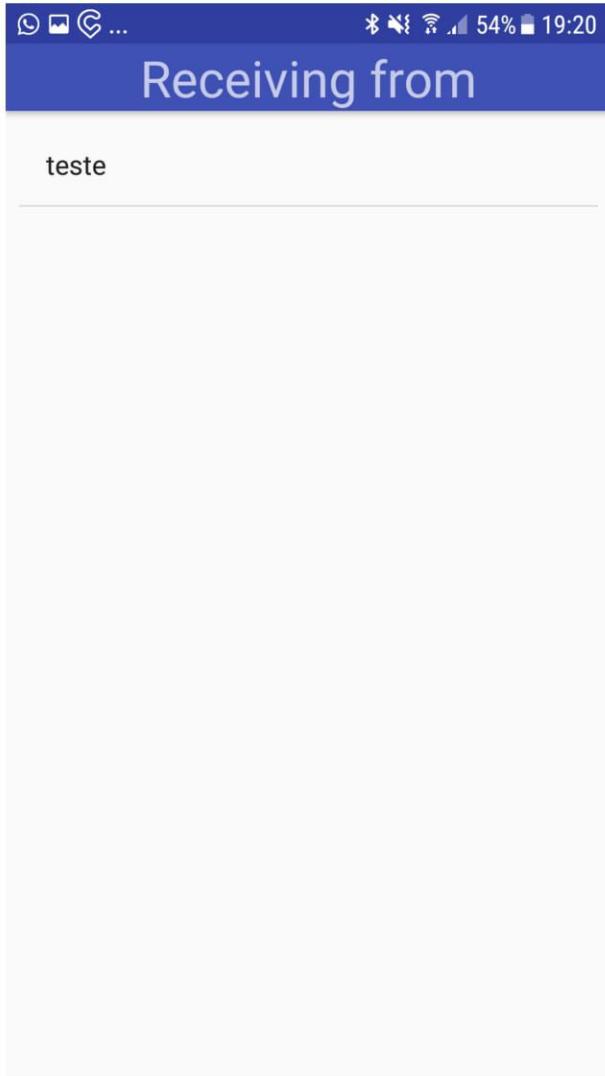


FIG. 4.6: Listagem dos usuários que querem enviar um arquivo

4.2.2.5 DOWNLOAD

Após escolher um dos remetentes possíveis da lista, é apresentada uma caixa de diálogo que pede ao usuário que ele digite a senha do arquivo. Após inserir a senha e confirmar a ação, o *download* deve ser iniciado. Para o recebimento do arquivo, foi utilizada a

classe *downloadAPI*. Nela, é definida uma requisição HTTP POST que envia o *token* de usuário no cabeçalho da requisição e dois parâmetros no corpo da requisição, relativos ao código identificador do arquivo e à senha inserida pelo usuário. A passagem da resposta à requisição à *activity* é feita de modo idêntico às situações anteriores, utilizando a interface *AsyncResponse*.

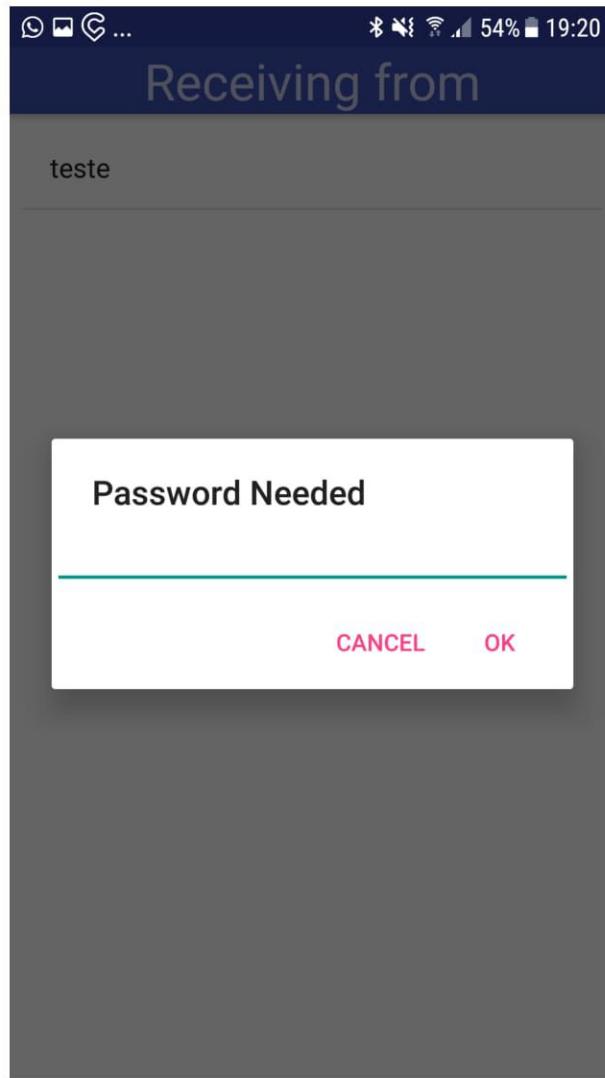


FIG. 4.7: Requisição da senha do arquivo desejado

Caso o servidor verifique que o arquivo referenciado pelo código existe e que sua senha é a mesma inserida pelo receptor, é retornada como resposta um objeto JSON que contém o nome do arquivo e a URL de acesso ao arquivo no servidor, que será aberta no navegador do dispositivo. Assim, a transferência de arquivos é terminada.

4.2.3 BEACONS E QUADROS DE AVISOS

Na *beaconActivity*, é inicializada uma *ListView* que deverá conter os quadros de avisos disponíveis naquele local, cujo modo de preenchimento difere um pouco dos descritos anteriormente. Um dos métodos chamados na *activity* é o *didRangeBeaconsInRegion()*, que captura as informações dos *beacons* varridos. Dentro da chamada desse método, é instanciado um objeto da classe *beaconAPI*, que assim como as outras classes de interação com o servidor, é implantada como subclasse de *AsyncTask*. A partir da URL lançada pelo *beacon*, é retornado como resposta à requisição HTTP GET um objeto JSON contendo as informações dos quadros de avisos associados ao *beacon* em questão, e que estejam disponíveis para o usuário em questão. Caso exista mais de um *beacon* no local, serão obtidos os quadros de todos.

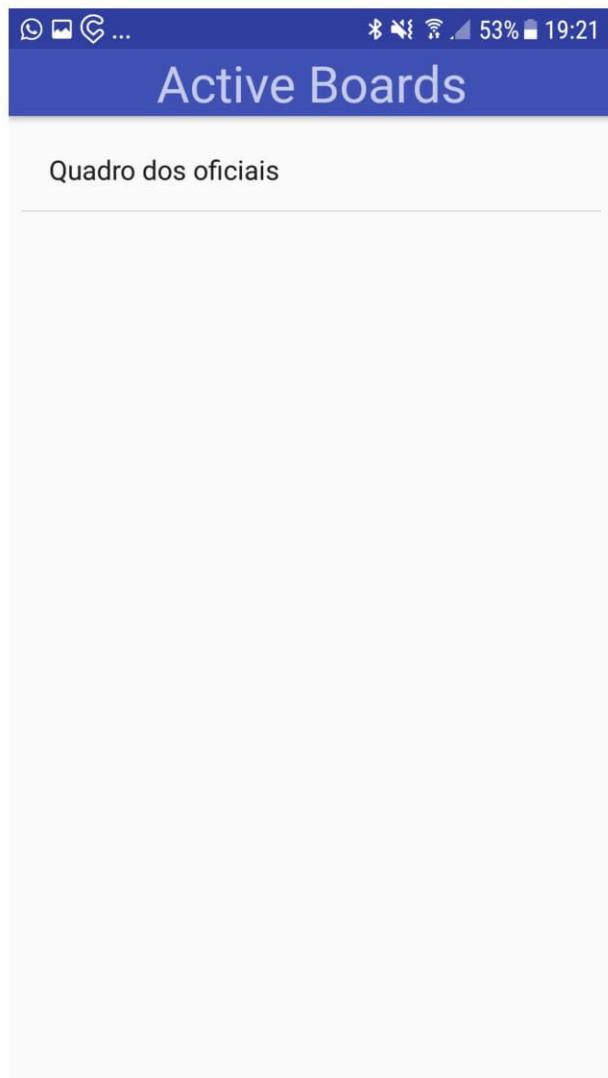


FIG. 4.8: Lista dos quadros de avisos disponíveis

Com essa informação, procede-se de forma semelhante a outras listas mencionadas anteriormente, definindo os itens da lista como objetos da classe *board*, mostrando ao usuário apenas o nome de cada quadro. Ao clicar sobre um item da lista, a URL associada é acessada no *browser* do dispositivo, mostrando os avisos daquele quadro.

4.3 DIFICULDADES ENCONTRADAS

Inicialmente, após a término do desenvolvimento e da etapa de testes, a aplicação funcionou perfeitamente. As etapas de teste abordaram desde o funcionamento sem falhas do aplicativo *mobile*, até a análise das respostas do servidor e a verificação das alterações no banco de dados.

Durante o desenvolvimento da aplicação, porém, alguns fatores tornaram a tarefa um pouco mais árdua. Durante a fase inicial do projeto, a primeira dificuldade encontrada está associada à falta de bibliotecas nativas de Java para requisições HTTP, o que não ocorre em uma linguagem como Python. Desse modo, a implementação do *upload*, listagem e *download* de arquivos levou mais tempo do que o planejado para ser finalizada.

Quanto à implementação das funcionalidades da API *Nearby*, o fato de ser uma API muito recente também gerou problemas no desenvolvimento. Isso leva a 2 fatos que, somados, geram bastante impacto no ritmo de desenvolvimento da aplicação. Primeiramente, a falta de documentação: as poucas documentações que o Google oferece abrangem apenas parte das possibilidades da API, tornando difícil a implantação de usos da API em estruturas mais complexas. Além disso, a falta de projetos similares na *internet* também é um fator prejudicial ao ritmo de desenvolvimento, dado que outros projetos servem como complementação à documentação.

Para a implementação da funcionalidade relacionada aos *beacons*, inicialmente foram utilizadas as bibliotecas *BluetoothLeScanner* e *BluetoothLeAdvertiser*. Porém, as funções dessas bibliotecas apresentaram um comportamento não consistente, funcionando em alguns momentos e não em outros. Assim, foi decidido utilizar funções da biblioteca *AltBeacon*, que foram desenvolvidas especificamente para interação com *beacons*, e não só com interações BLE genéricas. Após isso, o aplicativo se comportou do modo esperado.

5 CONCLUSÃO

Verifica-se que o aplicativo desenvolvido cumpre os requisitos estabelecidos previamente para este trabalho, que abordavam o uso da API *Nearby* e a transferência de arquivos como caso teste.

Outro ponto interessante é a combinação de diferentes conceitos para chegar à solução proposta. Exemplificando, para a funcionalidade de troca de arquivos, foram utilizados os conceitos da API *Nearby* em conjunto com uma REST API baseada em chamadas HTTP. Para a funcionalidade dos quadros de avisos, foram utilizados conceitos do uso de BLE para a varredura de ambiente, interação com uma REST API por meio de chamadas HTTP e a implantação de um *WebServer* para a disponibilização dos quadros de avisos.

Após entender o número de oportunidades de uso da API *Nearby*, e a partir das ideias desenvolvidas neste trabalho, algumas delas foram levantadas como possíveis pontos de partida para trabalhos futuros.

5.1 OPORTUNIDADES PARA TRABALHOS FUTUROS

- Uso de *beacons* para marcação de presença: a atual configuração do servidor permite aos administradores verificar os momentos em que um usuário estava no alcance de um determinado *beacon*. Desse modo, com um tratamento adequado desses *logs*, é possível determinar a presença de funcionários em determinada instalação, ou a presença de alunos em sala de aula.
- Smart-Cards para monitoramento: do ponto de vista de organizações, é muito importante saber a localização de um visitante ou até mesmo de um funcionário, caso existam áreas de acesso restrito (comum em instalações militares). Novamente, com o tratamento adequado dessas informações, é possível monitorar o caminho de um visitante dentro de uma organização. Para realizar a interação com os *beacons*, seria interessante o desenvolvimento de um dispositivo pequeno o suficiente para ser utilizado por uma pessoa como identificação (como um crachá é utilizado nos dias de hoje). Esse dispositivo deve conter um processador e interfaces *Bluetooth* e *Wifi*, de modo a possibilitar a interação com *beacons* e servidores.

6 REFERÊNCIAS BIBLIOGRÁFICAS

ANDROID DEVELOPERS. Arquitetura da Plataforma. Disponível em: <<https://developer.android.com/guide/platform/index.html?hl=pt-br>>. Acesso em: 20 de abril de 2017.

ANDROID DEVELOPERS. Fundamentos de Aplicativos. Disponível em: <<https://developer.android.com/guide/components/fundamentals.html?hl=pt-br>>. Acesso em: 22 de abril de 2017.

ANDROID DEVELOPERS. Intents e Filtros de Intents. Disponível em: <<https://developer.android.com/guide/components/intents-filters.html?hl=pt-br>>. Acesso em: 30 de abril de 2017.

ANDROID DEVELOPERS. Manifesto do aplicativo. Disponível em: <<https://developer.android.com/guide/topics/manifest/manifest-intro.html?hl=pt-br>>. Acesso em: 02 de maio de 2017.

ANDROID DEVELOPERS. Permissões do Sistema. Disponível em: <<https://developer.android.com/guide/topics/security/permissions.html?hl=pt-br>>. Acesso em: 27 de abril de 2017.

GOOGLE DEVELOPERS. Overview | Nearby Messages API | Google Developers. Disponível em: <<https://developers.google.com/nearby/messages/overview>>. Acesso em: 3 de maio de 2017.

GOOGLE DEVELOPERS. Overview | Nearby Connections API | Google Developers. Disponível em: <<https://developers.google.com/nearby/connections/overview>>. Acesso em: 15 de julho de 2017.

MAGNUS HYTTSTEN. Google Play services 7.8 - Let's see what's Nearby!. Disponível em: <<https://android-developers.googleblog.com/2015/08/google-play-services-78-lets-see-whats.html>>. Acesso em: 2 de maio de 2017.

FRANCK LEGENDRE. How Google Nearby (really) works – and what else it does?. Disponível em: <<http://blog.p2pkit.io/how-google-nearby-really-works-and-what-else-it-does>>. Acesso em: 3 de maio de 2017.

OFFICIAL ANDROID BLOG - BRIAN RAKOWSKI. Get ready for the sweet taste of Android 6.0 Marshmallow. Disponível em: <<https://android.googleblog.com/2015/10/get-ready-for-sweet-taste-of-android-60.html>>. Acesso em: 02 de maio de 2017.

WEBBER, J.; PARASTATIDIS, S. ; ROBINSON, I. The web as a platform for building distributed systems. In: LAURENT, S. S. (Org.). **REST in Practice: Hypermedia and Systems Architecture**. Califórnia: O'Reilly, 2010. p. 12.

YAGHMOUR, K. Internals primer. In: YAGHMOUR, K. (Org.). **Embedded Android**. Califórnia: O'Reilly, 2013. p. 26–29.

7 APÊNDICES

APÊNDICE 1: CÓDIGOS DAS CLASSES PRINCIPAIS

7.1.1 HOMEACTIVITY.JAVA

```
1 public class HomeActivity extends NearbyAPIActivity implements
2     AsyncCallback {
3
4     private String path;
5
6     private static final int FILE_SELECT_CODE = 3;
7
8     private Button uploadButton;
9     private Button transferButton;
10    private Button boardButton;
11
12    private NearbyAPIWrapper nearby;
13
14    private static final int TTL_IN_SECONDS = 3 * 60; // Three minutes.
15
16    private static final Strategy PUB_SUB_STRATEGY = new Strategy.
17        Builder()
18            .setTtlSeconds(TTL_IN_SECONDS).build();
19
20    // Key used in writing to and reading from SharedPreferences.
21    private static final String KEY_UUID = "key_uuid";
22
23    private String password = "";
24
25    private static String getUUID(SharedPreferences sharedpreferences) {
26        String uuid = sharedpreferences.getString(KEY_UUID, "");
27        if (TextUtils.isEmpty(uuid)) {
28            uuid = UUID.randomUUID().toString();
29            sharedpreferences.edit().putString(KEY_UUID, uuid).apply();
30        }
31        return uuid;
32    }
33
34}
```



```

69             chooserIntent = Intent.createChooser(intent, "Open
70                 file");
71
72         }
73
74         try {
75             startActivityForResult(chooserIntent,
76                 FILE_SELECT_CODE);
77         } catch (android.content.ActivityNotFoundException ex) {
78             Toast.makeText(getApplicationContext(), "No suitable
79                 File Manager was found.", Toast.LENGTH_LONG).
80                 show();
81         }
82     }
83
84     });
85
86
87
88     boardButton.setOnClickListener(new View.OnClickListener() {
89
90         @Override
91
92         public void onClick(View v) {
93
94             startActivityForResult(new Intent(HomeActivity.this,
95                 beaconActivity.class));
96
97         }
98     });
99
100
101
102     mMessageListener = new MessageListener() {
103
104         @Override

```

```

105     public void onLost(final Message message) {
106         // Called when a message is no longer detectable nearby.
107     }
108 }
109
110     nearby = new NearbyAPIWrapper(this, PUB_SUB_STRATEGY,
111                                     mMessageListener);
112
113 }
114
115 @Override
116 public void onConnected(@Nullable Bundle bundle) {
117     nearby.unsubscribe();
118     //nearby.subscribe();
119     Log.i(HomeActivity.class.getSimpleName(), "[+] Google Client
120             Connected");
121 }
122
123 @Override
124 public void onConnectionSuspended(int i) {
125     Toast.makeText(getApplicationContext(),"Connection Failed",
126                     Toast.LENGTH_LONG).show();
127     nearby.unsubscribe();
128 }
129
130 @Override
131 public void onConnectionFailed(@NonNull ConnectionResult
132                               connectionResult) {
133     Toast.makeText(getApplicationContext(),"Connection Failed",
134                     Toast.LENGTH_LONG).show();
135 }
136
137 @Override
138 protected void onActivityResult(int requestCode, int resultCode,
139                                 Intent data) {
140
141     switch (requestCode) {
142         case FILE_SELECT_CODE:
143             if (resultCode == RESULT_OK) {
144                 // Get the Uri of the selected file
145                 Uri uri = data.getData();

```

```

141
142     Log.d("", "File Uri: " + uri.toString());
143     // Get the path
144     try {
145         path = FileUtility.getPath(this, uri);
146         Log.d("", "File Path: " + path);
147
148         AlertDialog.Builder builder = new AlertDialog.
149             Builder(this);
150             builder.setTitle("Password Needed");
151
152             // Set up the input
153             final EditText input = new EditText(this);
154             // Specify the type of input expected; this, for
155             // example, sets the input as a password, and
156             // will mask the text
157             input.setInputType(InputType.TYPE_CLASS_TEXT |
158                 InputType.TYPE_TEXT_VARIATION_PASSWORD);
159             builder.setView(input);
160
161             // Set up the buttons
162             builder.setPositiveButton("OK", new
163                 DialogInterface.OnClickListener() {
164                     @Override
165                     public void onClick(DialogInterface dialog,
166                         int which) {
167                         password = input.getText().toString();
168                         uploadAPI upload = new uploadAPI();
169
170                         String token = ((MyApplication)
171                             HomeActivity.this.getApplication()).
172                             getToken();
173                         upload.set(HomeActivity.this,
174                             HomeActivity.this, getApplication(),
175                             path, password, token);
176                         upload.execute();
177
178                     }
179                 });
180             builder.setNegativeButton("Cancel", new
181                 DialogInterface.OnClickListener() {
182                     @Override

```

```

172             public void onClick(DialogInterface dialog ,
173                     int which) {
174                 dialog.cancel();
175             }
176         });
177         builder.show();
178     }
179     catch(URISyntaxException e){
180         Toast.makeText(getApplicationContext(),"Failed",
181             Toast.LENGTH_LONG).show();
182     }
183 }
184 break;
185 }
186 }
187 super.onActivityResult(requestCode , resultCode , data);
188 }
189
190 @Override
191 public void processFinish(String result){
192
193     try{
194
195         JSONObject jsonObj = new JSONObject(result);
196         String code = jsonObj.getString("code");
197
198         Toast.makeText(HomeActivity.this , code , Toast.LENGTH_LONG).
199             show();
200     }
201
202     catch(JSONException e){
203         Toast.makeText(getApplicationContext(),"Failed",Toast.
204             LENGTH_LONG).show();
205     }
206     //Here you will receive the result fired from async class
207     //of onPostExecute(result) method.
208 }

```

7.1.2 LISTACTIVITY.JAVA

```
1
2  public class ListActivity extends NearbyAPIActivity implements
3      AsyncResponse {
4
5
6      private ListView file_list;
7
8
9      private class serverFile {
10
11          private String filename;
12          private String filecode;
13
14          public serverFile(String filename, String code){
15              this.filename = filename;
16              this.filecode = code;
17          }
18
19
20          @Override
21          public String toString(){
22              return filename;
23          }
24
25      }
26
27      private NearbyAPIWrapper nearby;
28
29      private static final int TTL_IN_SECONDS = 3 * 60; // Three minutes.
30
31      private static final Strategy PUB_SUB_STRATEGY = new Strategy.
32          Builder()
33              .setTtlSeconds(TTL_IN_SECONDS).build();
34
35          // Key used in writing to and reading from SharedPreferences.
36          private static final String KEY_UUID = "key_uuid";
37
38          private static String getUUID(SharedPreferences sharedpreferences) {
39              String uuid = sharedpreferences.getString(KEY_UUID, "");
```

```

39     if (TextUtils.isEmpty(uuid)) {
40         uuid = UUID.randomUUID().toString();
41         sharedPreferences.edit().putString(KEY_UUID, uuid).apply();
42     }
43     return uuid;
44 }
45
46 @Override
47 protected void onCreate(Bundle savedInstanceState) {
48     super.onCreate(savedInstanceState);
49     setContentView(R.layout.activity_list);
50
51     file_list = (ListView) findViewById(R.id.file_list);
52
53     MessageListener mMessageListener = new MessageListener() {
54         @Override
55         public void onFound(final Message message) {
56             // Called when a new message is found.
57
58         }
59
60         @Override
61         public void onLost(final Message message) {
62             // Called when a message is no longer detectable nearby.
63
64         }
65     };
66
67     nearby = new NearbyAPIWrapper(this, PUB_SUB_STRATEGY,
68         mMessageListener);
69
70     String token = ((MyApplication) ListActivity.this.getApplication()
71         ().getToken();
72     listAPI fileList = new listAPI(ListActivity.this);
73     fileList.set(getApplicationContext(), token);
74     fileList.execute();
75
76     file_list.setOnItemClickListener(new AdapterView.
77         OnItemClickListener() {
78         @Override

```

```

78     public void onItemClick(AdapterView<?> adapterView, View
79             view, int position, long l) {
80
81         final ListActivity.serverFile item = (ListActivity.
82             serverFile) file_list.getItemAtPosition(position);
83
84         String name = item.toString();
85
86         AlertDialog.Builder builder = new AlertDialog.Builder(
87             ListActivity.this);
88
89         builder.setTitle("Send file?");
90         builder.setMessage("Send file" + name + " ?");
91
92         builder.setPositiveButton("YES", new DialogInterface.
93             OnClickListener() {
94
95                 public void onClick(DialogInterface dialog, int
96                     which) {
97
98                     String code = item.getFilecode();
99
100                    BluetoothAdapter myDevice = BluetoothAdapter.
101                        getDefaultAdapter();
102                    String deviceName = ((MyApplication)
103                        ListActivity.this.getApplication()).
104                        getUsername();
105
106                    HashMap params = new HashMap();
107                    params.put("device", deviceName);
108                    params.put("code", code);
109
110                    Message msg = NearbyMessage.newNearbyMessage (
111                        params,
112                        MessageType.CodeMessage,
113                        getUUID(getApplicationContext(),
114                            getSharedPreferences(
115                                getPackageName(),
116                                Context.MODE_PRIVATE))
117                );
118
119                nearby.publish(msg);

```

```

110
111             dialog.dismiss();
112         }
113     });
114
115     builder.setNegativeButton("NO", new DialogInterface.
116         OnClickListener() {
117
118             @Override
119             public void onClick(DialogInterface dialog, int
120                 which) {
121
122                 // Do nothing
123                 dialog.dismiss();
124             }
125         });
126
127         AlertDialog alert = builder.create();
128         alert.show();
129         // Toast.makeText(getApplicationContext(),item.
130         getFilecode(),Toast.LENGTH_LONG).show();
131
132     }
133
134     @Override
135     public void onConnected(@Nullable Bundle bundle) {
136
137         nearby.unsubscribe();
138         //nearby.subscribe();
139         Log.i(ListActivity.class.getSimpleName(), "[+] Google Client
140             Connected");
141     }
142
143     @Override
144     public void onConnectionSuspended(int i) {
145
146         Toast.makeText(getApplicationContext(),"Connection Failed",
147             Toast.LENGTH_LONG).show();
148         nearby.unsubscribe();
149     }
150
151     @Override
152     public void onConnectionFailed(@NonNull ConnectionResult

```

```

        connectionResult) {
    Toast.makeText(getApplicationContext(), "Connection Failed",
        Toast.LENGTH_LONG).show();
}

@Override
public void processFinish(String result){
}

try{
    final java.util.List<ListActivity.serverFile> output = new
    ArrayList<>();

    final ArrayAdapter<ListActivity.serverFile> adapter = new
    ArrayAdapter<>(this,
        android.R.layout.simple_list_item_1, android.R.id.
        text1, output);

    if(file_list != null){
        file_list.setAdapter(adapter);
    }

    JSONObject jsonObj = new JSONObject(result);
    JSONArray jArray = jsonObj.getJSONArray("rows");
    JSONObject data = null;

    for(int i = 0; i < jArray.length(); i++){
        data = jArray.getJSONObject(i);

        String file_code = data.getString("code");
        String file_name = data.getString("name");

        serverFile item = new serverFile(file_name, file_code);
        adapter.add(item);
    }
}

catch(JSONException e){
    Toast.makeText(getApplicationContext(), "Failed", Toast.
        LENGTH_LONG).show();
}

```

```

183         //Here you will receive the result fired from async class
184         //of onPostExecute(result) method.
185     }
186 }
```

7.1.3 RECEIVE.JAVA

```

1
2 public class Receive extends NearbyAPIActivity implements AsyncResponse
{
3
4     private ListView sender_list;
5
6     private static final int TTL_IN_SECONDS = 3 * 60; // Three minutes.
7
8     private static final Strategy PUB_SUB_STRATEGY = new Strategy.
9         Builder()
10            .setTtlSeconds(TTL_IN_SECONDS).build();
11
12
13
14     class CodeMessage {
15
16         private String device;
17
18         private String code;
19
20         CodeMessage(String device, String code){
21             this.device = device;
22             this.code = code;
23         }
24
25         public String getCode(){
26             return code;
27         }
28
29         @Override
30         public String toString(){
31             return device;
32         }
33     }
}
```



```

66         // Set up the buttons
67         // Set up the buttons
68         builder.setPositiveButton("OK", new DialogInterface.
69             OnClickListener() {
70                 @Override
71                 public void onClick(DialogInterface dialog, int
72                     which) {
73                     String pass = input.getText().toString();
74
75                     String token = ((MyApplication) Receive.this.
76                         getApplication()).getToken();
77
78                     downloadAPI download = new downloadAPI();
79                     download.set(Receive.this, Receive.this,
80                         getApplication(), item.getCode(), pass, token
81                         );
82                     download.execute();
83                 }
84             });
85         builder.setNegativeButton("Cancel", new DialogInterface.
86             OnClickListener() {
87                 @Override
88                 public void onClick(DialogInterface dialog, int
89                     which) {
90                     dialog.cancel();
91
92                     Receive.this.finish();
93                 }
94             });
95
96         builder.show();
97     }
98
99     MessageListener mMessageListener = new MessageListener() {
100        @Override
101        public void onFound(final Message message) {
102            // Called when a new message is found.
103
104            NearbyMessage msg = NearbyMessage.fromNearbyMessage(
105                message);
106
107        }
108    };

```

```

100         if (msg.getMessageType() == MessageType.CodeMessage){
101
102             Map body = msg.getMessageBody();
103             CodeMessage m = new CodeMessage( (String)body.get("device"),
104                                         (String)body.get("code"));
105             adapter.add(m);
106
107         }
108     }
109
110     @Override
111     public void onLost(final Message message) {
112         // Called when a message is no longer detectable nearby.
113
114         NearbyMessage msg = NearbyMessage.fromNearbyMessage(
115             message);
116
117         Map body = msg.getMessageBody();
118         CodeMessage m = new CodeMessage( (String)body.get("device"),
119                                         (String)body.get("code"));
120         adapter.remove(m);
121
122     };
123
124 }
125
126     @Override
127     public void onConnected(@Nullable Bundle bundle) {
128         nearby.unsubscribe();
129         nearby.subscribe();
130         Log.i(Receive.class.getSimpleName(), "[+] Google Client
131             Connected");
132
133     @Override
134     public void onConnectionSuspended(int i) {
135         Toast.makeText(getApplicationContext(), "Connection Failed",
136             Toast.LENGTH_LONG).show();

```

```

136         nearby.unsubscribe();
137     }
138
139     @Override
140     public void onConnectionFailed(@NonNull ConnectionResult
141             connectionResult) {
142         Toast.makeText(getApplicationContext(), "Connection Failed",
143             Toast.LENGTH_LONG).show();
144     }
145
146     @Override
147     public void processFinish(String result){
148
149         JSONObject jsonObj = new JSONObject(result);
150         String file_url = jsonObj.getString("file_url");
151         String name_url = jsonObj.getString("name");
152
153         Intent i = new Intent(Intent.ACTION_VIEW,
154             Uri.parse(getString(R.string.server_url) +file_url))
155             ;
156         startActivity(i);
157     }
158
159     catch(JSONException e){
160         Toast.makeText(getApplicationContext(),"Failed",Toast.
161             LENGTH_LONG).show();
162     }
163
164     //Here you will receive the result fired from async class
165     //of onPostExecute(result) method.
166 }
167 }
```

7.1.4 BEACONACTIVITY.JAVA

```

1
2 public class beaconActivity extends AppCompatActivity implements
3     AsyncResponse, BeaconConsumer, RangeNotifier {
4
5     private BluetoothManager mBTManager;
```

```

5     private BluetoothAdapter mAdapter;
6
7     private BeaconManager beaconManager;
8     private ListView board_list;
9
10    final java.util.List<board> output = new ArrayList<>();
11    ArrayAdapter<board> adapter;
12
13
14    private class board {
15
16        private String name;
17        private URL url;
18
19        public board(String name, URL url){
20            this.name = name;
21            this.url = url;
22        }
23
24        public String getName(){
25            return name;
26        }
27
28        @Override
29        public String toString(){
30            return name;
31        }
32
33        public String getURL(){
34
35            return this.url.toString();
36        }
37    }
38
39    @Override
40    protected void onCreate(Bundle savedInstanceState) {
41
42        super.onCreate(savedInstanceState);
43        setContentView(R.layout.activity_beacon);
44        board_list = (ListView) findViewById(R.id.board_list);
45
46

```

```

47     mBTManager = (BluetoothManager) getSystemService(Context.
48             BLUETOOTH_SERVICE);
49
50
51     beaconManager = BeaconManager.getInstanceForApplication(this.
52             getApplicationContext());
53     beaconManager.getBeaconParsers().add(new BeaconParser().
54             setBeaconLayout(BeaconParser.EDDYSTONE_URL_LAYOUT));
55     beaconManager.bind(this);
56
56     adapter = new ArrayAdapter<board>(beaconActivity.this,
57             android.R.layout.simple_list_item_1, android.R.id.text1,
58             output);
59
59     if(board_list != null){
60         board_list.setAdapter(adapter);
61     }
62
63     board_list.setOnItemClickListener(new AdapterView.
64             OnItemClickListener() {
65                 @Override
66                 public void onItemClick(AdapterView<?> adapterView, View
67                     view, int position, long l) {
68
69                     final board item = (board) board_list.getItemAtPosition(
70                         position);
71
72                     Intent i = new Intent(Intent.ACTION_VIEW,
73                         Uri.parse(item.getURL()));
74                     startActivity(i);
75
76                 }
77             });
78
79     @Override
80     public void onResume(){
81         super.onResume();
82         /*if (!mAdapter.isEnabled())
83         {
84             Intent enableBtIntent = new Intent(BluetoothAdapter.

```

```

        ACTION_REQUEST_ENABLE);
83     startActivityForResult(enableBtIntent, 1);
84 }
85 if (!mAdapter.isEnabled()){
86     mAdapter.enable();
87 }
88
89 beaconManager = BeaconManager.getInstanceForApplication(this.
90     getApplicationContext());
91 beaconManager.getBeaconParsers().add(new BeaconParser().
92     setBeaconLayout(BeaconParser.EDDYSTONE_URL_LAYOUT));
93 beaconManager.bind(this);
94 }
95
96 public void onBeaconServiceConnect() {
97     Region region = new Region("all-beacons-region", null, null,
98         null);
99     try {
100         beaconManager.startRangingBeaconsInRegion(region);
101     } catch (RemoteException e) {
102         e.printStackTrace();
103     }
104     beaconManager.setRangeNotifier(this);
105 }
106
107 @Override
108 public void didRangeBeaconsInRegion(Collection<Beacon> beacons,
109     Region region) {
110     for (Beacon beacon: beacons) {
111         if (beacon.getServiceUuid() == 0xfeaa && beacon.
112             getBeaconTypeCode() == 0x10) {
113             try {
114                 // This is a Eddystone-URL frame
115                 String url = UrlBeaconUrlCompressor.uncompress(
116                     beacon.getId1().toByteArray());
117                 String token = ((MyApplication) beaconActivity.this.
118                    getApplication()).getToken();
119                 beaconAPI getContent = new beaconAPI(beaconActivity.
120                     this);
121                 getContent.set(getApplicationContext(), new URL(url), token
122                     );
123                 getContent.execute();

```

```

116             /*Log.d(TAG, "I see a beacon transmitting a url: " +
117              url +
118              " approximately " + beacon.getDistance() + "
119              meters away.");*/
120         } catch (MalformedURLException e) {
121             e.printStackTrace();
122         }
123     }
124
125     @Override
126     protected void onPause() {
127         super.onPause();
128         if(mAdapter.isEnabled()){
129             mAdapter.disable();
130         }
131         beaconManager.unbind(this);
132     }
133
134     @Override
135     public void processFinish(String result){
136
137         try{
138             //TODO: Reject other beacons results
139             JSONObject jsonObj = new JSONObject(result);
140             JSONArray jArray = jsonObj.getJSONArray("rows");
141             JSONObject data = null;
142
143             for(int i = 0; i < jArray.length(); i++){
144
145                 data = jArray.getJSONObject(i);
146
147                 String beaconUrl = data.getString("url");
148                 String name = data.getString("name");
149
150                 URL bUrl = new URL( getApplicationContext().getString(R.
151                     string.server_url) + beaconUrl);
152
153                 board item = new board(name, bUrl);
154
155                 boolean exist = false;

```

```

155             for(int j = 0; j < adapter.getCount(); j++){
156                 if(adapter.getItem(j).getURL().equals(item.getURL())){
157                     exist = true;
158                 }
159             }
160             if(!exist) {
161                 adapter.add(item);
162             }
163         }
164     } catch (JSONException e) {
165         e.printStackTrace();
166     } catch (MalformedURLException e) {
167         e.printStackTrace();
168     }
169 }
170 }
171 }
```

7.1.5 NEARBYAPIWRAPPER.JAVA

```

1
2 public class NearbyAPIWrapper {
3
4     private String TAG = MainActivity.class.getSimpleName();
5     private GoogleApiClient mGoogleApiClient;
6     private final Strategy PUB_SUB_STRATEGY;
7     private final MessageListener mMessageListener;
8
9     public NearbyAPIWrapper(NearbyAPIActivity activity, Strategy
10                         strategy, MessageListener mMessageListener){
11         this.PUB_SUB_STRATEGY = strategy;
12         this.mMessageListener = mMessageListener;
13         buildGoogleApiClient(activity);
14     }
15
16     private void buildGoogleApiClient(NearbyAPIActivity activity) {
17         if (mGoogleApiClient != null) {
18             return;
19         }
20         this.mGoogleApiClient = new GoogleApiClient.Builder(activity)
21             .addApi(Nearby.MESSAGES_API)
```

```

21         .addConnectionCallbacks(activity)
22         .enableAutoManage(activity, activity)
23         .build();
24     }
25
26     public void publish(Message msg) {
27         Log.i(TAG, "Publishing");
28         PublishOptions options = new PublishOptions.Builder()
29             .setStrategy(PUB_SUB_STRATEGY)
30             .setCallback(new PublishCallback() {
31                 @Override
32                 public void onExpired() {
33                     super.onExpired();
34                     Log.i(TAG, "No longer publishing");
35                     // TODO : publish expired
36                 }
37             }).build();
38
39         Nearby.Messages.publish(mGoogleApiClient, msg, options)
40             .setResultCallback(new ResultCallback<Status>() {
41                 @Override
42                 public void onResult(@NonNull Status status) {
43                     if (status.isSuccess()) {
44                         // TODO : publish successfull
45                         Log.i(TAG, "Published successfully.");
46                     } else {
47                         // TODO : publish failure
48                         //mPublishSwitch.setChecked(false);
49                     }
50                 }
51             });
52     }
53
54     public void unpublish(Message msg) {
55         Nearby.Messages.unpublish(mGoogleApiClient, msg);
56     }
57
58     public void subscribe() {
59         Log.i(TAG, "Subscribing");
60         SubscribeOptions options = new SubscribeOptions.Builder()
61             .setStrategy(PUB_SUB_STRATEGY)
62             .setCallback(new SubscribeCallback() {

```

```

63             @Override
64             public void onExpired() {
65                 super.onExpired();
66                 // TODO : subscription expiration
67                 Log.i(TAG, "No longer subscribing");
68             }
69         }).build();
70
71     Nearby.Messages.subscribe(mGoogleApiClient, mMessageListener,
72         options)
73         .setResultCallback(new ResultCallback<Status>() {
74             @Override
75             public void onResult(@NonNull Status status) {
76                 if (status.isSuccess()) {
77                     // TODO : subscription successfull
78                     Log.i(TAG, "Subscribed successfully.");
79                 } else {
80                     // TODO : subscription failure
81                     //mSubscribeSwitch.setChecked(false);
82                 }
83             }
84         });
85
86     public void unsubscribe() {
87         Nearby.Messages.unsubscribe(mGoogleApiClient, mMessageListener);
88     }
89
90     public boolean isConnected(){
91         return (mGoogleApiClient != null && mGoogleApiClient.isConnected
92             ());
93     }
94
95     public void disconnect(){
96         if(mGoogleApiClient != null ){
97             unsubscribe();
98             mGoogleApiClient.disconnect();
99         }
100
101    public boolean connect(){
102        if(mGoogleApiClient == null){

```

```

103         return false;
104     } else{
105         mGoogleApiClient.connect();
106         subscribe();
107         return true;
108     }
109 }
110
111 }
```

7.1.6 BEACONAPI.JAVA

```

1
2 public class beaconAPI extends AsyncTask <String, String, String> {
3
4     private AsyncResponse get;
5
6     public beaconAPI(AsyncResponse get){
7         this.get = get;
8     }
9
10    private URL url;
11    private String token;
12
13    public void set(Context ctx, URL url, String token) {
14        this.url = url;
15        this.token = token;
16    }
17
18    private String beacon(){
19
20        StringBuilder result = new StringBuilder();
21
22        try {
23
24            HttpURLConnection urlConnection = (HttpURLConnection) url.
25                openConnection();
26            urlConnection.setDoInput(true); //Allow Inputs
27            //urlConnection.setDoOutput(true); //Allow Outputs
28            //urlConnection.setRequestProperty("Authorization", "Token
29                "+token);
30            urlConnection.setRequestProperty("Accept", "*/*");
31
32        } catch (IOException e) {
33            e.printStackTrace();
34        }
35    }
36
37    protected String doInBackground(String... params) {
38
39        String result = beacon();
40
41        return result;
42    }
43
44    protected void onPostExecute(String result) {
45
46        get.onResponse(result);
47    }
48}
```

```

29         urlConnection.setInstanceFollowRedirects(false);
30
31         int respCode = urlConnection.getResponseCode();
32
33         if(respCode == 301){
34
35             String location = urlConnection.getHeaderField("Location
36             ");
37             URL fUrl = new URL(location);
38             urlConnection = (HttpURLConnection) fUrl.openConnection
39             ();
40             urlConnection.setDoInput(true); //Allow Inputs
41             urlConnection.setRequestProperty("Authorization", "Token
42             "+token);
43             urlConnection.setRequestProperty("Accept", "*/*");
44             urlConnection.setInstanceFollowRedirects(false);
45
46         }
47
48         InputStream in = new BufferedInputStream(urlConnection.
49             getInputStream());
50
51         BufferedReader reader = new BufferedReader(new
52             InputStreamReader(in));
53
54         String line;
55         while ((line = reader.readLine()) != null) {
56             result.append(line);
57         }
58
59     } catch (Exception e) {
60
61         System.out.println(e.getMessage());
62
63         return e.getMessage();
64     }
65
66     return result.toString();
67 }
68
69
70     @Override

```

```

66     protected void onPreExecute() {
67         super.onPreExecute();
68     }
69
70     @Override
71     protected String doInBackground(String... params) {
72
73         return beacon();
74     }
75 }
76
77     @Override
78     protected void onPostExecute(String result) {
79
80         get.processFinish(result);
81     }
82 }
```

7.1.7 DOWNLOADAPI.JAVA

```

1
2 public class downloadAPI extends AsyncTask<String, String, String> {
3
4     private AsyncResponse get;
5
6     private String url;
7     private Activity act;
8     private Context ctx;
9     private String token;
10    private String code;
11    private String pass;
12    private int serverResponseCode;
13    private StringBuilder result;
14
15    public downloadAPI() {
16        this.url = "";
17        this.act = null;
18    }
19
20    public void set(AsyncResponse get, Activity act, Context ctx, String
21        code, String pass, String token) {
22        this.get = get;
```

```

22     this.act = act;
23     this.ctx = ctx;
24     this.url = ctx.getString(R.string.server_url) + ctx.getString(R.
25         string.url_from_code_endpoint);
26     // server URLs are stored in the project's configuration, in app
27     // /res/values/strings
28     this.code = code;
29     this.pass = pass;
30     this.token = token;
31
32     }
33
34     @Override
35     protected void onPreExecute() {
36         super.onPreExecute();
37     }
38
39     @Override
40     protected String doInBackground(String... params) {
41
42         this.result = new StringBuilder();
43
44         try {
45
46             URL url = new URL(this.url);
47
48             HttpURLConnection connection = (HttpURLConnection) url.
49                 openConnection();
50             // Token generated from Django Server, used to authorize
51             // connections
52             connection.setDoInput(true); //Allow Inputs
53             connection.setDoOutput(true); //Allow Outputs
54             connection.setUseCaches(false); //Don't use a cached Copy
55             connection.setRequestMethod("POST");
56             connection.setRequestProperty("Connection", "Keep-Alive");
57             connection.setRequestProperty("Content-Type", "application/
58                 json; charset=UTF-8");
59             connection.setRequestProperty("Authorization", "Token "+this
60                 .token);
61
62             JSONObject data = new JSONObject();

```

```

58
59         data.put("code", this.code);
60         data.put("password", this.pass);
61
62         OutputStreamWriter wr = new OutputStreamWriter(connection.
63             getOutputStream());
64         wr.write(data.toString());
65         wr.flush();
66
67         serverResponseCode = connection.getResponseCode();
68         InputStream in = new BufferedInputStream(connection.
69             getInputStream());
70
71         BufferedReader reader = new BufferedReader(new
72             InputStreamReader(in));
73         //BufferedReader reader = new BufferedReader(new
74             //InputStreamReader(connection.getInputStream()));
75
76         String line;
77         while ((line = reader.readLine()) != null) {
78             result.append(line);
79         }
80
81         Log.i(TAG, "***** Server Response is: " + result +
82             ": " + serverResponseCode);
83
84     } catch (Exception e) {
85
86         System.out.println(e.getMessage());
87         return e.getMessage();
88     }
89
90     return result.toString();
91
92
93     @Override
94     protected void onPostExecute(String result) {
95
96         get.processFinish(result);
97         // passes results to activities
98     }

```

```
95 }
```

7.1.8 LISTAPI.JAVA

```
1
2 public class listAPI extends AsyncTask <String, String, String>{
3
4     private AsyncResponse get;
5
6     public listAPI(AsyncResponse get){
7         this.get = get;
8     }
9
10    private static String url;
11    private String token;
12
13    public void set(Context ctx, String token) {
14        this.url = ctx.getString(R.string.server_url) + ctx.getString(R.
15            string.list_files_endpoint);
16        this.token = token;
17    }
18
19    private String listFile(){
20
21        StringBuilder result = new StringBuilder();
22
23        try {
24
25            URL url = new URL(this.url);
26
27            HttpURLConnection urlConnection = (HttpURLConnection) url.
28                openConnection();
29            urlConnection.setRequestProperty("Authorization", "Token "+
30                token);
31
32            InputStream in = new BufferedInputStream(urlConnection.
33                getInputStream());
34
35            BufferedReader reader = new BufferedReader(new
36                InputStreamReader(in));
37
38            String line;
```

```

34         while ((line = reader.readLine()) != null) {
35             result.append(line);
36         }
37     } catch (Exception e) {
38
39         System.out.println(e.getMessage());
40
41         return e.getMessage();
42     }
43
44     return result.toString();
45 }
46
47
48
49     @Override
50     protected void onPreExecute() {
51         super.onPreExecute();
52     }
53
54     @Override
55     protected String doInBackground(String... params){
56
57         return listFile();
58     }
59
60
61     @Override
62     protected void onPostExecute(String result) {
63
64         get.processFinish(result);
65     }
66 }
```

7.1.9 UPLOADAPI.JAVA

```

1
2 public class uploadAPI extends AsyncTask<String, String, String>{
3
4     private AsyncResponse get;
5     private Activity act;
6     private Context ctx;
```

```

7     private String path;
8     private String pass;
9     private String token;
10
11    public uploadAPI() {
12        this.path = "";
13        this.act = null;
14    }
15
16    public void set(AsyncResponse get, Activity act, Context ctx, String
17                    path, String pass, String token) {
18        this.get = get;
19        this.act = act;
20        this.ctx = ctx;
21        this.path = path;
22        this.pass = pass;
23        this.token = token;
24    }
25
26    @Override
27    protected void onPreExecute() {
28        super.onPreExecute();
29    }
30
31    @Override
32    protected String doInBackground(String... params) {
33
34        int serverResponseCode = 0;
35        String serverResponseMessage = "Failed";
36        HttpURLConnection connection;
37        DataOutputStream dataOutputStream;
38        StringBuilder result = new StringBuilder();
39        String lineEnd = "\r\n";
40        String twoHyphens = "--";
41        String boundary = Long.toHexString( System.currentTimeMillis() )
42                      ;
43
44        int bytesRead, bytesAvailable, bufferSize;
45        byte[] buffer;
46        int maxBufferSize = 1 * 1024 * 1024;
        File selectedFile = new File(path);

```

```

47
48
49     String[] parts = path.split("/");
50     final String fileName = parts[parts.length - 1];
51
52     if (!selectedFile.isFile()) {
53
54         act.runOnUiThread(new Runnable() {
55             @Override
56             public void run() {
57                 Toast.makeText(act, "File doesn't exist", Toast.
58                     LENGTH_SHORT).show();
59             }
60         });
61         return serverResponseMessage;
62     } else {
63         try {
64
65             FileInputStream fileInputStream = new FileInputStream(
66                 selectedFile);
67             URL url = new URL(ctx.getString(R.string.server_url) +
68                 ctx.getString(R.string.upload_file_endpoint));
69             connection = (HttpURLConnection) url.openConnection();
70             connection.setDoInput(true); //Allow Inputs
71             connection.setDoOutput(true); //Allow Outputs
72             connection.setUseCaches(false); //Don't use a cached Copy
73             connection.setRequestMethod("POST");
74             connection.setRequestProperty("Connection", "Keep-Alive"
75                 );
76             //connection.setRequestProperty("ENCTYPE", "multipart/
77             //form-data");
78             connection.setRequestProperty("Content-Type", "multipart
79                 /form-data;boundary=" + boundary);
80             connection.setRequestProperty("Authorization", "Token "
81                 + token);
82             //connection.setRequestProperty("file", path);

83
84             //creating new dataoutputstream
85             dataOutputStream = new DataOutputStream(connection.
86                 getOutputStream());
87
88             //writing bytes to data outputstream

```

```

81         dataOutputStream.writeBytes(twoHyphens + boundary +
82                                     lineEnd);
83
84         dataOutputStream.writeBytes("Content-Disposition: form-
85                                     data; name=\"password\" " + lineEnd);
86
87         dataOutputStream.writeBytes("Content-Type: text/plain;
88                                     charset=UTF-8"+lineEnd);
89
90         dataOutputStream.writeBytes(lineEnd);
91
92         dataOutputStream.writeBytes(pass);
93
94         dataOutputStream.writeBytes(twoHyphens + boundary +
95                                     twoHyphens + lineEnd);
96
97         dataOutputStream.writeBytes("Content-Disposition: form-
98                                     data; name=\"file\";filename=\""
99                                     + path + "\" " + lineEnd);
100
101
102         dataOutputStream.writeBytes(lineEnd);
103
104
105         //returns no. of bytes present in fileInputStream
106         bytesAvailable = fileInputStream.available();
107
108         //selecting the buffer size as minimum of available
109         //bytes or 1 MB
110         bufferSize = Math.min(bytesAvailable, maxBufferSize);
111
112         //setting the buffer as byte array of size of bufferSize
113         buffer = new byte[bufferSize];
114
115
116         //reads bytes from FileInputStream(from 0th index of
117         //buffer to buffersize)
118         bytesRead = fileInputStream.read(buffer, 0, bufferSize);
119
120
121         //loop repeats till bytesRead = -1, i.e., no bytes are
122         //left to read
123         while (bytesRead > 0) {
124
125             //write the bytes read from inputstream
126             dataOutputStream.write(buffer, 0, bufferSize);
127
128             bytesAvailable = fileInputStream.available();
129
130             bufferSize = Math.min(bytesAvailable, maxBufferSize)
131             ;

```

```

114         bytesRead = fileInputStream.read(buffer, 0,
115                                         bufferSize);
116
117         dataOutputStream.writeBytes(twoHyphens + boundary +
118                                     twoHyphens + lineEnd);
119
120         serverResponseCode = connection.getResponseCode();
121
122         InputStream in = new BufferedInputStream(connection.
123                                         getInputStream());
124         BufferedReader reader = new BufferedReader(new
125                                         InputStreamReader(in));
126
127         String line;
128         while ((line = reader.readLine()) != null) {
129             result.append(line);
130         }
131
132         //closing the input and output streams
133         fileInputStream.close();
134         dataOutputStream.flush();
135         dataOutputStream.close();
136
137     } catch (FileNotFoundException e) {
138         e.printStackTrace();
139         act.runOnUiThread(new Runnable() {
140             @Override
141             public void run() {
142                 Toast.makeText(act, "File Not Found", Toast.
143                               LENGTH_SHORT).show();
144             }
145         });
146     } catch (MalformedURLException e) {
147         e.printStackTrace();
148         Log.e(TAG, "URL error!");
149     } catch (IOException e) {
150         e.printStackTrace();
151         Log.e(TAG, "Cannot Read/Write File!");
152     }

```

```
151         return result.toString();
152     }
153 }
154
155     @Override
156     protected void onPostExecute(String result) {
157
158         get.processFinish(result);
159
160     }
161 }
```