PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
ESCOLA DE CIÊNCIAS EXATAS E DA COMPUTAÇÃO
GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

APLICAÇÃO DE *BLUETOOTH* ASSOCIADA AO *GEOFENCING*

HENRIQUE COELHO DOS SANTOS

GOIÂNIA
2021

HENRIQUE COELHO DOS SANTOS

APLICAÇÃO DE *BLUETOOTH* ASSOCIADA AO *GEOFENCING*

Trabalho de Conclusão de Curso apresentado à Escola de Ciências Exatas e da Computação, da Pontifícia Universidade Católica de Goiás, como parte dos requisitos para a obtenção do título de Bacharel em Engenharia de Computação.

Orientador:
Prof. Marcelo Antonio Adad de Araújo, M.E.E.

GOIÂNIA
2021

HENRIQUE COELHO DOS SANTOS

APLICAÇÃO DE *BLUETOOTH* ASSOCIADA AO *GEOFENCING*

Trabalho de Conclusão de Curso aprovado em sua forma final pela Escola de Ciências Exatas e da Computação, da Pontifícia Universidade Católica de Goiás, para obtenção do título de Bacharel em Engenharia de Computação, em 07 de junho de 2021.

_____

Prof. Ma. Ludmilla Reis Pinheiro dos Santos
Coordenadora de Trabalho de Conclusão de Curso

Banca examinadora:

_____

Orientador: Prof. Marcelo Antonio Adad de Araújo, M.E.E.

_____

Prof. Carlos Alexandre Ferreira de Lima, M.E.E.

_____

Prof. Fernando Gonçalves Abadia, M.E.E.

GOIÂNIA
2021

Com gratidão, a Deus por tudo que tem feito em minha vida.

Aos meus pais, sem eles nada seria possível, pelo amor e assistência nos momentos difíceis.

Aos meus familiares, minha base.

A todos que me ajudaram e estiveram presentes em minha trajetória acadêmica.

AGRADECIMENTOS

A Deus por me proporcionar a perseverança, me mantendo no caminho certo durante a elaboração deste trabalho, providenciando a força e saúde necessárias.

Aos meus pais Ana Maria Filha Coelho Santos e Faustino Gomes dos Santos pelo apoio incondicional ao longo de toda minha vida e por sempre investirem em mim e em minha educação.

Ao meu professor orientador Marcelo Antonio Adad de Araújo pela paciência e confiança depositada em mim, bem como todas as instruções necessárias para a realização deste trabalho.

Aos meus amigos Brenno Giovanini de Moura e Laryssa Salustiana de Oliveira Pires que me acompanharam por grande parte em minha trajetória acadêmica, pelo companheirismo, união e pelos bons momentos juntos.

A Isis Vilanova e Silva Lima, por me oferecer a estadia durante boa parte da minha formação acadêmica, tornando possível minha dedicação aos estudos em tempo integral.

RESUMO

Este trabalho apresenta a continuação do desenvolvimento de um aplicativo móvel para o sistema operacional Android, que utiliza a localização do usuário dentro de um perímetro virtual, denominado *geofence*, para a tomada de decisões. Esta nova versão tem por objetivo a realização da autenticação de um usuário através da utilização de *Bluetooth Low Energy* e obtenção do *Media Access Control* (MAC) do dispositivo. Para isso será realizada a montagem do componente de hardware, o microcontrolador ESP32, aliado ao aplicativo presente no dispositivo Android, adicionando ainda novas funcionalidades que incluem métodos de validação biométricas de impressão digital e facial, integração com o Google Firebase, remodelagem das interfaces gráficas e mudança na forma de interação do usuário com as mesmas. Além disso consta incluso o código-fonte da aplicação e explicação de todas as suas partes componentes, tanto da parte referente o Android quanto para o ESP32, separados de acordo com suas respectivas funções dentro do código.

Palavras-Chave: *Android; geofence; Bluetooth; microcontrolador; ESP32.*

ABSTRACT

This work presents the continuation of the development of a mobile application for the Android operating system, which uses the user's location within a virtual perimeter, called geofence, for decision making. This new version aims to carry out the authentication of a user through the use of Bluetooth Low Energy and obtaining the Media Access Control (MAC) of the device. For this, the hardware component ESP32 microcontroller will be assembled, together with the application present on the Android device, adding new features that include fingerprint and facial biometric validation methods, integration with Google Firebase, remodeling of the graphical interfaces and change in the way the user interacts with them. Also included is the application's source code and explanation of all its component parts, both the part referring to Android and ESP32, separated according to their respective functions within the code.

Keywords: *Android; geofence; Bluetooth; microcontoller; ESP32.*

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

Ma.	Mestra
M.E.E	Mestre em engenharia elétrica
Prof.	Professor
PUC	Pontifícia Universidade Católica

LISTA DE SIGLAS

IoT	*Internet of Things*
IEEE	*Institute of Electrical and Electronics Engineers*
Wi-Fi	*Wireless Fidelity*
RFID	*Radio Frequency Identification*
BLE	*Bluetooth Low Energy*
GHz	gigahertz
ISM	*Industrial, Scientific and Medical*
AES	*Advanced Encryption Standard*
GFSK	*Gaussian Frequency-Shift Keying*
MHz	megahertz
FHSS	*Frequency-Hopping Spread Spectrum*
Mbps	*megabits per second*
FH	*Frequency Hopping*
CPU	Unidade Central de Processamento
GPS	Sistema de Posicionamento Global
API	Interface de Programação de Aplicativos
SSID	*Service Set Identifier*
MAC	*Media Access Control*
ROM	*Read Only Memory*
NIC	*Network Interface Card*
ASCII	*American Standard Code for Information Interchange*
OUI	*Organizationally Unique Identifier*
RFC	*Request for Comments*
PIN	*Personal Identification Number*
AI	Inteligência Artificial
RF	Rádio Frequência
balun	*Balanced-unbalanced*
PCB	Placa de circuito impresso
IDE	*Integrated Development Environment*
APK	*Android Application Package*
XML	*eXtensible Markup Language*
SDK	*Software Development Kit*

URL	*Uniform Resource Locator*
JSON	*JavaScript Object Notation*
NoSQL	*Not Only Standard Query Language*
BSD	*Berkeley Software Distribution*
MMX	*MultiMedia eXtensions*
SSE	*Streaming SIMD Extensions*
MLKit	*Machine Learning Kit*
LED	Diodo Emissor de Luz
UUID	*Universally Unique Identifier*
GB	gigabytes
RAM	*Random Access Memory*

SUMÁRIO

1 INTRODUÇÃO

Este trabalho de conclusão de curso visa continuar o desenvolvimento de um aplicativo para o sistema operacional Android criado inicialmente por Paulo Victor Alexandre Alves (ALVES, 2018) e continuado posteriormente por Gabriel Sarmento Loureiro (LOUREIRO, 2019).

A *Internet of Things* (IoT), em português "Internet das coisas", é um conceito que utiliza redes sem fio para a integração de diversos dispositivos. Redes sem fio utilizam radiofrequência para a transmissão de dados e possuem uma série de protocolos para operar, possuindo características distintas para diversas aplicações. Um exemplo desse tipo de rede é o *Bluetooth Low Energy* (BLE), ou *Bluetooth* de baixa energia (SANTOS, 2016; TEIXEIRA 2016).

O uso de geolocalização permite que qualquer posição no globo terrestre seja determinada e, devido à maior abrangência desse tipo de serviço em smartphones atuais, tornou-se possível que aplicativos se beneficiassem disso. Tomando isso como base, é possível delimitar um perímetro virtual denominado *geofence* que possibilita que ações sejam tomadas em seu interior, pois antes desse fato as ações só poderiam ser tomadas de forma geral, ou seja, a *geofence* é um dos maiores avanços quando se pensa em comércio eletrônico localizado, estratégias de marketing, outras aplicações e segurança que será tratada no presente trabalho (MACHADO, 2015; WHITE, 2017).

A autenticação de usuários em aplicativos busca garantir que informações sejam acessadas pelos devidos proprietários, composta de duas etapas: de identificação e verificação. Existem vários métodos para que isso seja possível, incluindo a utilização do endereço MAC, diversos tipos de biometrias, como de impressão digital, da palma da mão, voz, dinâmica de digitação, de íris e facial (STALLINGS; BROWN, 2014).

Assim, o presente trabalho busca responder à seguinte questão de pesquisa:

- É possível validar um usuário de Android em uma *geofence* utilizando *Bluetooth*, juntamente com a obtenção do endereço MAC do dispositivo aliado às biometrias de digital ou facial?

1.2 Objetivos

Esta seção descreverá os objetivos geral e específicos do trabalho.

1.2.1 Objetivo Geral

O presente trabalho tem por objetivo realizar a autenticação de um usuário em um aplicativo Android, que se encontra dentro de uma *geofence*, com o uso do *Bluetooth* em conjunto com o endereço *Media Access Control* (MAC) do dispositivo móvel.

1.2.2 Objetivos Específicos

- Implementar funções de validação por *Bluetooth* no aplicativo Android;
- Delimitar a localidade de um veículo com uma *geofence*;
- Utilizar o *Bluetooth* para conectar o microcontrolador ESP32 dentro do carro com o dispositivo Android do proprietário;
- Validar o usuário do aplicativo através do endereço MAC do dispositivo;
- Validar o usuário do aplicativo através de confirmação por leitor biométrico de impressão digital e detecção facial;
- Simular o destrave da tranca da porta de um carro pela requisição de abertura feita pelo smartphone através do ESP32.

1.3 Justificativa

O presente trabalho justifica-se por possibilitar a integração de dispositivos eletroeletrônicos através da IoT, utilizando tecnologias presentes no smartphone Android juntamente com microcontroladores acessíveis e de baixo custo para a automatização de tarefas simples do cotidiano, de uma forma segura.

O aplicativo possibilitará ao usuário uma maior comodidade, providenciando a facilidade de ter o controle de dispositivos eletroeletrônicos remotamente, de forma que apenas o devido proprietário possa ser capaz de realizar ações em locais específicos.

1.4 Métodos

Primeiramente uma revisão bibliográfica será realizada baseada em artigos científicos, monografias, dissertações de mestrado, teses de doutorado, livros, revistas e outros meios de informações confiáveis. Feito isso, será feita montagem dos componentes de hardware juntamente com o que for implementado por software, agregando as novas funcionalidades ao aplicativo desenvolvido em trabalhos anteriores, realizando testes para se determinar que a aplicação do *Bluetooth* e das autenticações estão funcionando corretamente.

1.5 Resultados esperados

Espera-se com este trabalho que os resultados obtidos contribuam para o crescimento e aperfeiçoamento de soluções em IoT, em especial as que envolvem *geofencing* e utilização de redes sem fio para a autenticação de usuários em sistemas, com ênfase na automatização de atividades cotidianas, no que diz respeito a aplicativos móveis.

2 ESTADO DA ARTE

Este capítulo contém a fundamentação teórica necessária para a elaboração deste trabalho, com conceitos e definições importantes para a compreensão, incluindo trabalhos relacionados ao tema principal.

2.1 *Internet of Things*

A chegada da Internet possibilitou o acesso à informação de modo rápido e acessível para muitas pessoas, ocasionando diversas mudanças significativas no cotidiano, porém, a utilização da Internet se estende além da interação entre usuários (SANTOS, 2016).

A IoT é um conceito em que diversos dispositivos reúnem informações através de sensores, que se comunicam entre si utilizando redes sem fio. O sufixo “coisas” se refere a equipamentos que podem ser conectados uns aos outros, como smartphones, lâmpadas, tomadas, entre outros, não somente construídos pelo homem como naturais também (SANTOS, 2016).

Esse termo foi utilizado pela primeira vez em 1999 por Kevin Ashton em uma reunião que teve como tema central o etiquetamento eletrônico de produtos utilizando identificadores de rádio frequência (RFID) e então se popularizou rapidamente. A ideia principal foi integrar o mundo real ao mundo virtual, com o intuito de gerar maior eficiência (ASHTON, 2015).

A IoT possibilita que diversos tipos de problemas reais possam ser resolvidos através da interconexão de dispositivos inteligentes. Vários tipos de produtos são integrados com tecnologias que permitem a IoT, se estendendo às mais variadas aplicações e áreas, entre elas entretenimento, mobilidade, eletrodomésticos e automação (MAGRANI, 2018).

2.2 Redes sem fio

Com o avanço da tecnologia, diversos dispositivos eletroeletrônicos tornaram-se capazes de se comunicar de várias formas. A necessidade de otimização de tarefas e processos, bem como a criação de novos métodos para realização de tarefas, alguns até mesmo considerados impossíveis anteriormente, visando maior eficiência, são alguns dos motivos que levam aos avanços tecnológicos (TEIXEIRA, 2016).

A utilização de radiofrequência para transmissão de dados possibilitou um avanço nos equipamentos que até então necessitavam utilizar meios físicos como cabos e barramentos para comunicação. Esse método de transmissão é denominado rede sem fio (TEIXEIRA, 2016).

As redes sem fio utilizam de uma grande variedade de protocolos, possuindo características distintas que se adequam a diferentes situações, seja pela capacidade de transferência de dados, alcance ou velocidade de conexão (TEIXEIRA, 2016).

Para que haja um bom funcionamento dos equipamentos que fazem uso desses protocolos o *Institute of Electrical and Electronics Engineers* (IEEE), em português "Instituto de Engenheiros Eletroeletrônicos", criou uma série de normas (*standards*), que são consensos entre profissionais da área, para que haja a interoperabilidade entre as diferentes tecnologias associadas às redes sem fio. Como exemplos bastante conhecidos e difundidos tem-se o *Wireless Fidelity* (Wi-Fi) (IEEE 802.11) e o *Bluetooth* (IEEE 802.15) (TEIXEIRA, 2016).

2.2.1 Bluetooth Low Energy

A tecnologia *Bluetooth* possui uma variação chamada *Bluetooth Low Energy*. O BLE é uma tecnologia *wireless*, sem fio, que possui baixa potência por ser designada para a conexão de dispositivos que utilizam baterias, necessitando de um consumo reduzido de energia para manter o funcionamento por períodos extensos de uso de dias, meses ou anos (AFANEH, 2018). O BLE também é chamado de *Bluetooth Smart* e sua eficiência energética pode chegar a ser dez vezes maior do que o *Bluetooth* Clássico (BERNARDES, 2018).

Essa tecnologia foi introduzida ao mercado em 2010 com a chegada do *Bluetooth* 4.0. O BLE é uma tecnologia nova e não uma atualização da tecnologia *Bluetooth* já existente, com foco para a utilização na IoT e suas aplicações, fazendo a transferência de volumes de dados menores, mas utilizando a banda *Industrial, Scientific and Medical* (ISM) de 2,4 gigahertz (GHz) idêntica ao *Bluetooth* original, assim como o ZigBee e o Wi-Fi (AFANEH, 2018). A Tabela 1 ilustra as semelhanças e diferenças de algumas especificações técnicas entre o *Bluetooth* Clássico e o *Bluetooth Low Energy*.

Tabela 1 - Comparativo *Bluetooth* Clássico e *Bluetooth Low Energy*

Especificação	*Bluetooth* Clássico	*Bluetooth Low Energy* (BLE)
Criptografia	Chaves de 56 a 128 bits	Chave de 128 bits *Advanced Encryption Standard* (AES)
Esquema de modulação	*Gaussian Frequency-Shift Keying* (GFSK)	*Gaussian Frequency-Shift Keying* (GFSK)
Frequência	2400 a 24835 megahertz (MHz)	2400 a 24835 megahertz (MHz)
Índice de modulação	0,35	0,5
Largura de banda do canal	1 megahertz (MHz)	2 megahertz (MHz)
Quantidade de nós / escravos ativos permitidos	7	Ilimitado
Quantidade de canais	79	40
Robustez	*Frequency-Hopping Spread Spectrum* (FHSS)	*Frequency-Hopping Spread Spectrum* (FHSS)
Taxa de dados nominal	1 a 3 megabits *per second* (Mbps)	1 megabits *per second* (Mbps)
Taxa de transferência	0,7 a 2,1 megabits *per second* (Mbps)	Menor que 0,3 megabits *per second* (Mbps)
Técnica de modulação	*Frequency Hopping* (FH)	*Frequency Hopping* (FH)
Transferência de áudio	Aceita	Não aceita

Fonte: Zanchi, 2016.

O BLE não possui compatibilidade com o *Bluetooth* Clássico, portanto, não é possível haver comunicação direta entre as duas tecnologias, mesmo trabalhando na mesma faixa de frequência. Existem dispositivos que implementam ambas, mas funcionam de modo separado. O BLE possui diversos usos que podem ser em aplicações de automação residencial, como controle de luzes, climatização, trancas de portas, em aplicações relacionadas a prática de exercícios físicos, à computação pervasiva que utiliza de relógios e óculos inteligentes, além dos dispositivos médicos (AFANEH, 2018).

Para utilizar o BLE são necessários dois tipos de dispositivos: um central e um periférico. O dispositivo central é aquele que tem o maior poder computacional, com *Central Process Unit* (CPU), ou Unidade Central de Processamento, mais potente, maior capacidade de

memória e bateria. O dispositivo periférico possui os mesmos componentes, porém, mais limitados, em especial a quantidade de bateria. Devido à assimetria da tecnologia, a carga de trabalho se concentra no dispositivo central, fazendo com que o dispositivo periférico tenha a possibilidade de entrar em modo *sleep*, "adormecido" por períodos que podem ser extensos, preservando assim a bateria (AFANEH, 2018).

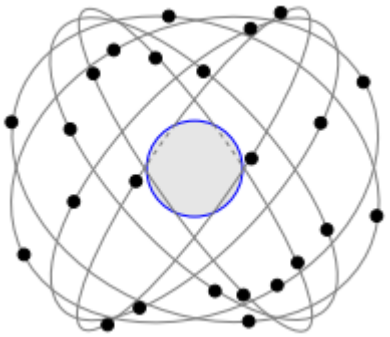
Existem muitos benefícios que fazem do BLE uma tecnologia viável, como o baixo consumo energético, a gratuidade de acesso às especificações e documentação do fabricante sem a necessidade de pagamentos ou filiação, o custo do hardware acessível e o fato de estar presente em grande parte dos modelos de smartphones atuais (AFANEH, 2018).

2.3 Geolocalização

A geolocalização consiste em determinar uma posição no globo terrestre. Para este fim o *Global Positioning System* (GPS), ou Sistema de Posicionamento Global, é um recurso utilizado atualmente que está presente em diversos equipamentos de uso cotidiano como smartphones, rastreadores e aparelhos que traçam rotas e envolvem direção. Através da maior abrangência desse recurso tornou-se possível o desenvolvimento de aplicativos que o utilizam para os mais diversos fins (MACHADO, 2015).

O GPS foi desenvolvido pelo Departamento de Defesa dos Estados Unidos e sua utilização se restringia a uso militar, posteriormente, tornou-se disponível para uso civil. Esse sistema funciona com base em 32 satélites, com pelo menos 24 deles em operação, distribuídos ao redor do globo em 6 planos orbitais e com altitudes de 20200 quilômetros acima da superfície, como demonstra a Figura 1 (FAGGIAN, 2019).

Figura 1 - Distribuição de satélites pelo globo terrestre.

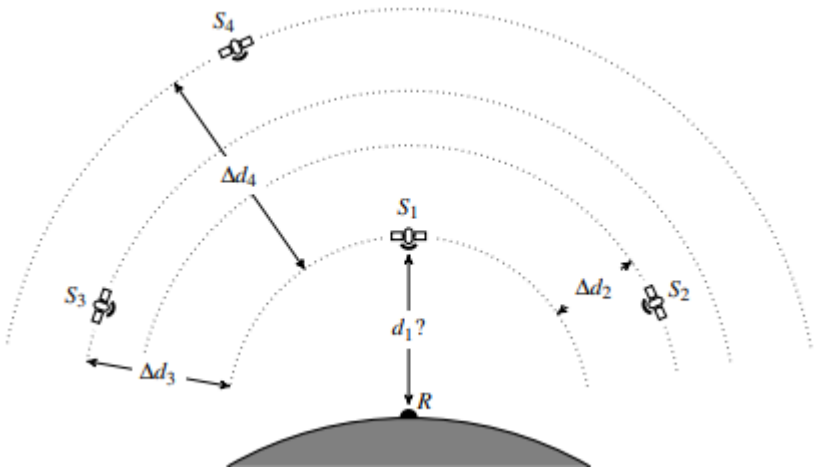


Fonte: Rousseau; Saint-Aubin, 2015.

Para serem determinados quaisquer pontos da superfície terrestre com precisão, são necessários que pelo menos quatro satélites tenham visão da área e não compartilhem o mesmo plano, conforme mostra a Figura 2. Os satélites têm suas rotas monitoradas por bases fixas e simultaneamente emitem sinais codificados que fornecem dados como posição, horário e código identificador (FAGGIAN, 2019).

Em relação à quantidade de satélites necessários, três deles são utilizados para a triangulação de coordenadas, enquanto o quarto faz-se necessário para a sincronização temporal, já que diferenças nos tempos de relógios ocorrem entre os satélites e o usuário, ocasionadas pelos diferentes materiais que os compõem (DOMPIERI *et al*., 2015).

Figura 2 - Organização de satélites para determinação de localização precisa.

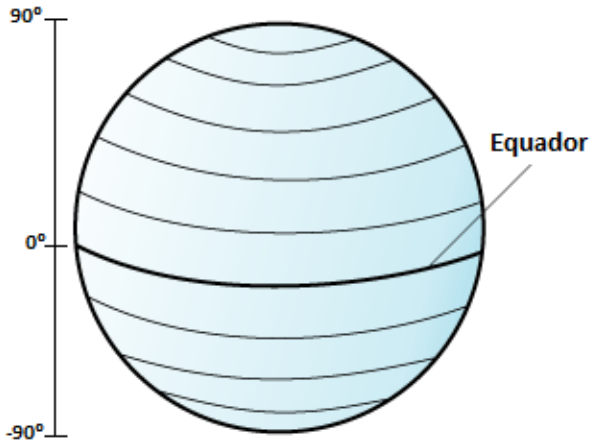


Fonte: Faggian, 2019.

O GPS utiliza um sistema de coordenadas geográficas que fazem uso de linhas imaginárias que subdividem o globo terrestre baseadas nas linhas do Equador e no Meridiano de *Greenwich*. Isso torna possível a utilização de códigos geográficos chamados de latitude e longitude (MACHADO, 2015).

Os paralelos são linhas imaginárias que formam círculos ao redor do globo e possuem um centro em comum, percorrem o sentido leste e oeste, possuindo latitude constante e são equidistantes, o que significa que possuem a mesma distância de uma linha para outra. A linha do Equador possui latitude de zero graus, as localizações que ficam ao sul dessa linha variam suas latitudes negativamente de 0 a -90 graus e as que ficam ao norte variam positivamente de 0 a 90 graus, conforme representado na Figura 3 (IBM KNOWLEDGE CENTER, 2018).

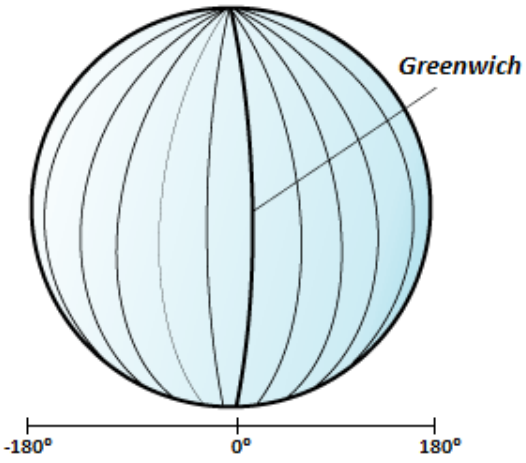
Figura 3 - Representação das linhas latitudinais.



Fonte: Adaptado de IBM Knowledge Center, 2018.

Os meridianos são linhas imaginárias que formam círculos ao redor do globo e possuem um centro relacional em comum, mas diferentemente dos paralelos, percorrem o sentido norte e sul, tem longitude constante e se cruzam nos polos. O Meridiano de *Greenwich* possui longitude de zero graus, as localizações que ficam ao oeste dessa linha variam suas longitudes negativamente de 0 a -180 graus até o meridiano antipodal, que corresponde a longitude de zero graus do outro lado do globo, e as que ficam ao leste variam positivamente de 0 a 180 graus igualmente até o meridiano antipodal, como mostra a Figura 4 (IBM KNOWLEDGE CENTER, 2018).

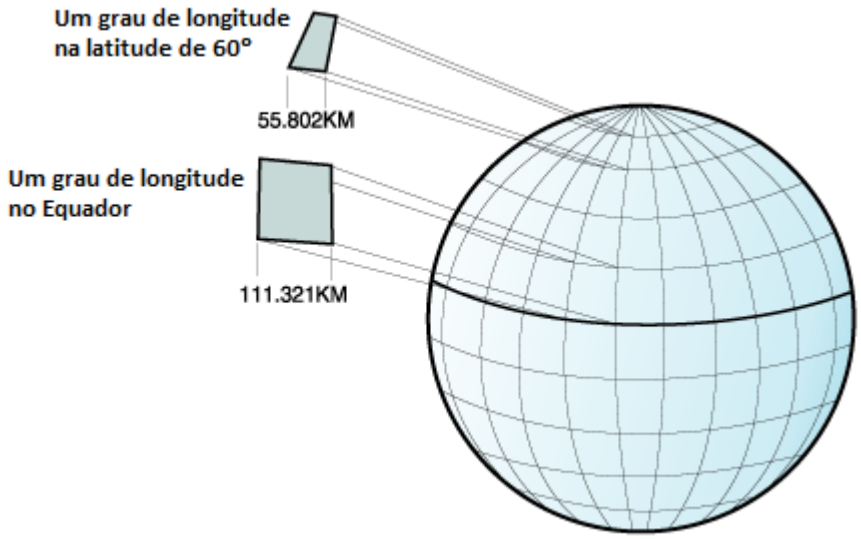
Figura 4 - Representação das linhas longitudinais.



Fonte: Adaptado de IBM Knowledge Center, 2018.

Fazendo o uso de latitudes e longitudes, pode-se gerar uma representação chamada gratícula, com sua origem no cruzamento das coordenadas de zero graus. Na gratícula, as distâncias de um grau de latitude e um grau de longitude possuem quase a mesma proporção no Equador, único local no qual isso ocorre, enquanto que em outras localizações, à medida que se aproxima dos polos, a distância de um grau de longitude é muito menor do que a de um grau de latitude, devido a diferença da distância entra os meridianos. A Figura 5 mostra um exemplo dos valores de um grau de longitude, em quilômetros, de latitudes distintas (IBM KNOWLEDGE CENTER, 2018).

Figura 5 - Diferenças nas dimensões em latitudes distintas.



Fonte: Adaptado de IBM Knowledge Center, 2018.

Para se representar latitude e longitude são utilizados valores em graus, além de minutos, segundos e milissegundos, caso seja necessário alto nível de precisão. Para que esses dados possam ser interpretados por sistemas de computação, essas representações necessitam ser convertidas em decimal ou valores de ponto flutuante, facilitando assim o desenvolvimento de softwares e os cálculos gerados por eles. Considerando que um grau equivale a 60 minutos e um minuto equivale a 60 segundos, um grau equivale a 3600 segundos com a fórmula para a conversão de valores dada por (MACHADO, 2015):

$$Latitude\ decimal = -\left(graus + \frac{minutos'}{60} + \frac{segundos''}{3600}\right) \quad (1)$$

$$Longitude\ decimal = -\left(graus + \frac{minutos'}{60} + \frac{segundos''}{3600}\right) \quad (2)$$

2.4 *Geofencing*

O *geofencing* consiste em utilizar a localização geoposicionada do usuário em um aplicativo, a partir de dados móveis obtidos por telefonia, GPS, Wi-Fi e outros tipos de tecnologias, para fazer com que ações sejam tomadas quando o usuário de dispositivo móvel entra dentro de uma cerca virtual configurada com base em coordenadas geográficas denominada *geofence* (WHITE, 2017).

Dependendo da maneira que é configurada a *geofence,* essa pode solicitar o envio de mensagens ou ainda desativar recursos do dispositivo, validar usuários, além de permitir o monitoramento de áreas que exigem segurança elevada, emitindo alertas quando um indivíduo entra ou sai da cerca, entre outras possibilidades se tratando de aplicações móveis (WHITE, 2017).

Geofences precisam necessariamente de serviços de localização, que faz com que sejam mais utilizadas em aplicativos móveis, porém, sua utilização não se restringe a apenas isso, o agronegócio, a indústria de pesca e os drones também fazem uso desse recurso. Em se tratando de drones, estes podem ser configurados para que mudem sua trajetória de voo, fazendo-os parar no ar, emitir sinais de advertência e fazer medições de área (WHITE, 2017).

Das áreas de aplicação destacam-se:

- Dispositivos eletroeletrônicos: em combinação com tecnologias como o *Bluetooth*, é possível configurar ações personalizadas como o ajuste de temperatura de um termostato de uma geladeira, por exemplo;
- Entretenimento: podem ser utilizadas *geofences* para informar pessoas em eventos como shows, teatros, feiras, entre outros;
- Marketing: as lojas podem direcionar anúncios de novidades nas mercadorias ou emitir alertas de promoções assim que o cliente entrar na loja;

- Recursos humanos: além do monitoramento de funcionários, a automatização do ponto eletrônico se torna possível registrando as horas de entrada e saída dos funcionários nas cercas;
- Redes sociais: filtros ou efeitos de fotos e vídeos podem ser configurados para estarem disponíveis em locais como shows por exemplo;
- Segurança: pode-se fazer o monitoramento de indivíduos que entram ou saiam da residência do proprietário ou configurar um dispositivo para que desbloqueie ao adentrar;
- Telemática: sinais de advertência podem ser enviados a operadores nos ambientes de produção (WHITE, 2017).

Para se calcular uma *geofence* é utilizada a Fórmula de Haversine, bastante conhecida nos meios de navegação, e que possibilita o cálculo da distância entre dois pontos, levando em consideração a curvatura da superfície terrestre utilizando latitudes e longitudes (STATLER, 2016). A fórmula é dada por:

$$hav(\theta) = sen^2\left(\frac{\theta}{2}\right) = \frac{1 - \cos(\theta)}{2} \tag{3}$$

$$hav\left(\frac{d}{r}\right) = (hav\Phi_2 - hav\Phi_1) + hav(\lambda_2 - \lambda_1)\cos(\Phi_1)\ \cos(\Phi_2) \tag{4}$$

Onde,

θ: é o ângulo formado pelos dois pontos;

d: é a distância entre dois pontos em uma esfera;

r: é o raio da esfera;

Φ_1, Φ_2: são as latitudes dos pontos 1 e 2 respectivamente;

λ_1, λ_2: são as longitudes dos pontos 1 e 2 respectivamente (ZIN *et al.*, 2016).

Calculadas as distâncias é possível gerar as cercas em formatos circulares, com base na distância entre os pontos para formar o diâmetro da circunferência. Utilizando uma Interface de Programação de Aplicativos (API) um desenvolvedor pode gerar o código para uma aplicação de *geofence* e algumas linguagens de programação possuem funções específicas para a manipulação no sistema operacional Android. A configuração de *geofences* também pode ser feita por usuários finais dentro dos aplicativos móveis, podendo selecionar onde as cercas devem ser posicionadas e quais devem ser removidas (WHITE, 2017).

Geofences são divididas em tipos, que podem ser ativo ou passivo. As do tipo ativo só funcionam quando o aplicativo está em uso, o GPS é inicializado para a obtenção de latitudes e longitudes de alta precisão e então é observado se o usuário se encontra dentro ou fora dos limites predefinidos para a *geofence*, os quais ficam armazenados de maneira local ou remota utilizando um banco de dados. Como o uso do GPS para localização precisa demanda um alto custo energético, é mais viável que o aplicativo solicite a localização quando se inicia e então pause logo após obtê-la (STATLER, 2016).

Caso sejam enviados dados sobre localização para o servidor, através da configuração de eventos, esse tipo de implementação promove dois benefícios importantes: a transferência do processamento dos dados para o servidor, que normalmente tem grande vantagem no poder de processamento e economiza a energia que seria gasta no dispositivo móvel, e a não exigência de atualização do aplicativo para adicionar novas *geofences*. O tipo ativo garante boas precisões, que não podem ser atingidas com o tipo passivo, e permite a criação de cercas pequenas, com a possibilidade de terem apenas 10 metros de diâmetro, aumentando a segurança da validação de usuários (STATLER, 2016). Pretende-se no presente trabalho melhorar a precisão utilizando *Bluetooth*.

As *geofences* do tipo passivo funcionam com o aplicativo em uso ou até mesmo encerrado, já que operam em segundo plano e isso é uma vantagem em relação ao tipo ativo, porém, a criação de cercas está restrita a extensões maiores que 100 metros de diâmetro, o que prejudica a precisão, devido à pouca utilização de GPS sendo mais dependente da combinação de Wi-Fi e dados móveis de telefonia (STATLER, 2016).

Algumas APIs nativas dos dispositivos Android fazem o aproveitamento de serviços de localização gerando maior autonomia de bateria. Esses dispositivos realizam conexões frequentes com torres de celular e em cada um deles existe uma lista contendo as localizações geográficas de cada torre, alocada em um banco de dados. À medida que se conectam, dispositivo e torre, pode-se inferir a possível localização do usuário, porém, sua precisão não é tão boa quanto a da *geofence* ativa, visto que observar-se que pode chegar a mais de 100 metros da localização real. A combinação com Wi-Fi permite a obtenção de mais pontos de referência para se determinar a localização atual (STATLER, 2016).

Redes Wi-Fi possuem um identificador chamado *Service Set Identifier* (SSID), ou identificador do conjunto de serviço. Caso o Wi-Fi esteja habilitado no dispositivo Android e o SSID não seja reconhecido pela Google, o GPS será ativado em segundo plano e a localização será salva no banco de dados. Não existe a necessidade de conexão com a rede para a obtenção do SSID, o dispositivo deve fazer uma varredura para reconhecer as redes disponíveis nas

proximidades e, considerando que grande parte deles está atrelada a uma localização já identificada, facilita o processo de identificar se o usuário está dentro dos limites da *geofence*. Caso o Wi-Fi esteja desabilitado, existe a possibilidade de obter precisão se o usuário abrir alguma aplicação que utilize GPS, permitindo assim que a *geofence* faça uso desses dados (STATLER, 2016).

2.5 Autenticação de usuários

Em se tratando de segurança em computadores, autenticar usuários em aplicações é um modo de garantir que os dados não serão acessados por alguém que não seja o devido proprietário. A autenticação de usuários é definida pela *Request for Comments* (RFC) 2828 como sendo um processo no qual se verifica a identidade de uma determinada entidade do sistema ou para uma outra entidade deste ou de outro sistema, sendo esse processo dividido em duas partes, etapa de identificação e etapa de verificação. A etapa de identificação consiste em fornecer ao sistema algum tipo de identificador, que precisa ser definido com cautela devido à utilização de dados autenticados em outros serviços que envolvem segurança. A etapa de verificação contribui para vincular identificadores às respectivas entidades do sistema (STALLINGS; BROWN, 2014).

Usuários podem ser autenticados em sistemas de quatro maneiras distintas, não havendo impedimento de utilização de técnicas em conjunto:

- Algo que o usuário sabe ou conhece: utiliza senhas (numéricas ou padrões desenhados), respostas para perguntas pré-configuradas ou *Personal Identification Number* (PIN);
- Algo que o usuário possui (*token*): utiliza chaves físicas, cartões eletrônicos com senha ou *smart cards*;
- Algo que o usuário faz (biometria dinâmica): utiliza o reconhecimento de padrões como traços de escrita, ritmo de digitação e voz;
- Algo que o usuário é (biometria estática): utiliza o reconhecimento de retina, de face ou impressão digital (STALLINGS; BROWN, 2014).

Para o presente trabalho serão utilizados os métodos de autenticação de biometria da face e digital e de algo que o usuário possui como um *token*, smartphone com *Media Access Control* (MAC), fazendo uso do *Bluetooth* para conectar o smartphone com o microcontrolador ESP32.

2.5.1 Endereço MAC de dispositivos

Dispositivos eletroeletrônicos que são aptos à conexão com rede possuem um código identificador de endereço físico chamado MAC, que fica armazenado em uma memória do tipo *Read Only Memory* (ROM), na placa de rede que também é chamada de *Network Interface Card* (NIC). O endereço MAC possui um tamanho de 48 bits (6 bytes), distribuídos de forma hexadecimal, sendo cada byte separado por um sinal de dois pontos do código *American Standard Code for Information Interchange*, ASCII 58, : (ROCHA, 2017).

Os três primeiros bytes do endereço MAC são referentes ao identificador do fabricante, fornecido pelo IEEE, chamado de *Organizationally Unique Identifier* (OUI) e os três últimos são escolhidos pelo fabricante para a identificação. Existe um único endereço MAC para identificar cada dispositivo fabricado, não havendo duplicações (ROCHA, 2017).

2.5.2 Biometria de impressão digital

Biometria, de modo geral, consiste em aplicar alguma forma de Inteligência Artificial (AI) ou métodos matemáticos, como análise e estatística quantitativa, em informações que advêm da biologia, gerando um bom grau de confiança para identificação de usuários em sistemas (SILVA, 2019).

Existem quatro fatores que contribuem para que sistemas que utilizam biometria sejam eficientes: critério quantitativo, permanência, unicidade e universalidade. O critério quantitativo faz com que os dados obtidos pela extração de características estejam dispostos em forma numérica, em quantidades que permitem informações precisas. A permanência diz respeito à baixa variabilidade, por exemplo, da impressão digital ao longo dos anos, não havendo mutações naturais que possam modificar o padrão. A unicidade garante que não haverá pessoas diferentes com mesmo padrão, por exemplo, de íris. A universalidade se refere ao fato de que essas características estão presentes em todos seres humanos praticamente desde antes do nascimento (SILVA, 2019).

A ideia de se utilizar impressões digitais para a identificação de indivíduos não é algo recente, porém, isso é feito de forma bastante consistente. Cada impressão digital é única para cada indivíduo e apresenta padrões de sulcos e saliências que a identifica. Um sistema que faz uso de biometria faz a verificação das características e as extrai, convertendo os dados obtidos

para o formato numérico de forma a constituir uma modelagem completa (STALLINGS; BROWN, 2014).

A estrutura das impressões digitais é formada por três componentes: linhas, deltas e núcleo. O conjunto das três estruturas são denominados singularidades e representa os detalhes globais da impressão digital. As linhas são a base para a criação das outras estruturas presentes na impressão digital. Os deltas são a união das linhas que formam um padrão característico que remete a um triângulo e que permite a divisão da impressão em classes e regiões denominadas basilar, marginal e nuclear. As linhas centrais da digital formam o núcleo, que tem uma grande importância e é geralmente caracterizado por um *loop* ou espiral, como ilustrado na Figura 6, podendo existir variações em que não seja bem definido, se esse for o caso, admite-se que o núcleo se encontra no interior da linha de curvatura mais acentuada (NETO *et al*., 2014; SILVA, 2019).

Figura 6 - Estruturas das singularidades

Fonte: Adaptado de Araújo, 2019; Férrer, 2008; Silva, 2019.

Singularidades delta são a base para se classificar impressões digitais e juntamente com a observação das outras singularidades, o então comissário da polícia de metrópole de Londres, Edward Henry, formulou em 1905 cinco classes, conforme mostrado na Figura 7, que possuem as seguintes características:

1) Arco Angular: possui um delta abaixo do núcleo, que por sua vez tem formato de tenda;
2) Arco Plano: não possui delta e é composta de arcos sobrepostos em camadas;
3) Presilha Externa: possui um delta à esquerda do núcleo, as linhas que o formam são curvas, possuem centro em comum, tem origem à direita da impressão digital e tentem a retornar para onde se iniciam;

4) Presilha Interna: possui padrão espelhado à presilha externa;
5) Verticilo: possui dois deltas cada um de um lado do núcleo, que se encontra exatamente no centro da impressão digital (SILVA, 2019).

Figura 7 - Classificação de impressões digitais

Fonte: Adaptado de Silva, 2019.

Nas impressões digitais se encontram presentes outros tipos de estruturas denominadas minúcias ou características de Galton, que são os detalhes locais. A identificação de indivíduos através de impressões digitais foi primeiramente pesquisada por Francis Galton em 1888 e seu estudo foi de grande importância para a identificação de cristas, que correspondem às elevações presentes na impressão digital e os espaços entre elas, que são os sulcos (SILVA, 2019).

Através das minúcias pode-se realizar a distinção entre indivíduos, já que os detalhes globais estão presentes em todas as impressões digitais. Existem seis tipos de minúcias sendo as principais para a identificação as terminações, também chamadas de cristas finais, e as bifurcações, que são tipos simples. As outras quatro minúcias são tipos compostos chamados de cristas curtas, cruzamentos, esporas e ilhas, como mostra a Figura 8 (VAL; MARCELINO; NETO, 2015; SILVA, 2019).

Figura 8 - Tipos de minúcias

Fonte: Adaptado de Silva, 2019.

2.5.3 Reconhecimento facial

Em se tratando de seres humanos, as características do rosto são as que permitem com mais facilidade a identificação de indivíduos, sendo plausível a utilização de reconhecimento facial para sistemas computacionais. Para extração de características comumente leva-se em consideração a forma e a localização relativa dos componentes da face como boca, nariz, olhos, sobrancelhas e formato do queixo. Alternativamente pode-se utilizar sensores infravermelhos a fim de produzir um termograma que se baseia no calor gerado pelo sistema vascular na face. (STALLINGS; BROWN, 2014).

Existem diferenciações nos conceitos de detecção e de reconhecimento facial. A detecção se refere à capacidade de se localizar um rosto de um indivíduo em uma imagem, enquanto o reconhecimento é caracterizado pela identificação do indivíduo através da análise dos componentes da face (ANDREZZA, 2015).

O processo de reconhecimento facial pode ser dividido em três modos distintos e suas utilizações dependem de como o sistema de reconhecimento vai operar, sendo eles: verificação, identificação e observação (COSTA, 2019).

O modo de verificação serve para constatar que determinado indivíduo é realmente quem ele declara ser, permitindo ou recusando acesso, por exemplo, em funcionalidades de aplicativos. Existe uma separação de grupos de indivíduos para a realização do teste de verificação, em grupos de clientes, que são realmente os detentores das identidades declaradas, e impostores, que buscam se passar por outra pessoa através de identidades falsas. Nessa fase imagens são comparadas e apenas uma correspondência de faces é admitida, considerando todas armazenadas no sistema (COSTA, 2019). O presente trabalho utilizará o modo de verificação para a autenticação de um usuário no dispositivo Android.

No modo de identificação as faces nas quais serão realizados os testes são conhecidas e o banco de dados pode ser formado pela junção de diversas bases de dados distribuídas por diversos locais em que o sistema se encontra em uso. Quanto maior o volume e a diversidade de faces, maior será a possibilidade de se reconhecer um indivíduo em ambientes com grandes aglomerações de pessoas e grande espaço físico (COSTA, 2019).

O modo de observação é semelhante ao de identificação, admitindo ainda faces desconhecidas que serão adicionadas em uma lista de acordo com o ambiente em que se encontram. Os indivíduos são observados ao surgirem no ambiente com a intenção de se constatar se existe correspondência com a lista (COSTA, 2019).

A eficiência do reconhecimento facial está atrelada ao fato de que não existe a necessidade de contribuição do usuário além da captura da própria face por uma câmera, que pode ser feita a distâncias relativamente grandes em se tratando de lugares amplos ou pela câmera de smartphone por exemplo, ocorrendo de maneira quase que imediata, pois o usuário sempre estará olhando para a tela do smartphone, diferentemente da biometria por reconhecimento de íris que necessita da contribuição do usuário posicionando os olhos em frente a um leitor e não necessidade de espera por longos períodos para ser realizado (GUIMARÃES, 2015).

O processo de identificação pode ter sua eficiência reduzida para alguns tipos de situações, como obstrução da face por algum tipo de objeto, mudanças estéticas muito acentuadas, expressões faciais exageradas, ângulo, entre outras, mas que podem ser evitadas através de condições a serem impostas aos usuários (GUIMARÃES, 2015).

A detecção e reconhecimento facial podem ser divididos em quatro abordagens:

1) Baseada em aparência: os algoritmos que usam essa abordagem fazem uso de aprendizado e treinamento com grandes bases de dados e retiram os dados necessários para a detecção das próprias imagens analisadas, considerando que a detecção facial faz parte do reconhecimento de padrões;
2) Baseada em características invariantes: utilizam características presentes nas faces que são consideradas invariantes, como textura da pele e tonalidade, para destacar a face do restante dos componentes da imagem analisada, e se baseiam na capacidade de reconhecimento que os seres humanos têm na identificação de características, não importando o contexto. Podem ser utilizadas cores em escala de cinza ou no espectro de cores completo e quanto mais cores mais informações podem ser utilizadas;
3) Baseada em conhecimento: uma base de regras é utilizada para determinar se o que consta em uma imagem é um rosto verdadeiramente, de acordo com as noções e conhecimento de quem criou essas regras. Se tratando de face humana, todos rostos possuem uma boca, um nariz e dois olhos e suas próprias localizações no espaço da face;
4) Baseada em *Templates*: as características faciais serão comparadas com modelos preestabelecidos. Esses modelos podem ser criados com base em formas geométricas, como círculos e triângulos por exemplo, e são chamados de *templates*. A detecção dependerá do grau de correspondência entre o

template e as formas do rosto presentes na imagem analisada (BISSI, 2018; COSTA, 2019).

2.6 Trabalhos relacionados

Esta seção apresentará trabalhos relacionados com o tema deste trabalho e suas análises.

2.6.1 Verificação automática georreferenciada (2018)

O trabalho em questão teve por objetivo o estudo e desenvolvimento de um aplicativo móvel para Android que possuísse a capacidade de fazer a verificação automática georreferenciada de um usuário, através da verificação e validação de sua geolocalização por meio de APIs (*geofence*), automatizando ações de um sistema embarcado e controlando acesso a locais ao destravar uma tranca (ALVES, 2018).

Para a construção do sistema foram selecionados os softwares para prototipação Arduino IDE, Android Studio e Android SDK *Tools*, as linguagens de programação C e Java, e o microcontrolador ESP88266-12E, através de estudos para que se tivesse a melhor exatidão, desempenho e segurança (ALVES, 2018).

O trabalho concluiu que o projeto demonstrou grande potencial para ser destinado a diversas aplicações que fazem uso de posição georreferenciada de usuários, devido à sua facilidade de utilização, estando presente em uma grande quantidade de dispositivos móveis atuais, viabilizando o seu uso aliado a utilização de sistemas embarcados que possuem baixo custo (ALVES, 2018).

2.6.2 Verificação automática georreferenciada (2019)

Este trabalho foi desenvolvido como uma continuação direta do trabalho feito por Paulo Victor Alexandre Alves (ALVES, 2018), com o objetivo de adicionar novas funcionalidades que incluíam a possibilidade de o usuário definir o local que uma nova *geofence* pudesse ser criada e o que automatizar em seu interior, como o ar condicionado de um ambiente de trabalho por exemplo, além de permitir ao usuário saber a última localização do seu veículo

particular e uma maior segurança para a abertura da tranca implementada na primeira versão (LOUREIRO, 2019).

A ideia inicial se manteve e foram adicionados novos requisitos de software e hardware, utilizando dois microcontroladores para a ampliação das capacidades de automatização de aparelhos, validação biométrica de usuário por impressão digital e aperfeiçoamento de código fonte. Obteve-se uma melhora significativa no aplicativo tornando possível que as novas funcionalidades de configuração fossem adicionadas de modo mais fácil (LOUREIRO, 2019).

3 PROPOSTA DE SOLUÇÃO

Este capítulo apresentará a proposta de solução, complementando trabalhos anteriores com a utilização do *Bluetooth*, de forma a melhorar a segurança do sistema.

3.1 Componentes de hardware

Esta seção descreverá o componente de hardware para a prototipação do trabalho.

3.1.1 Microcontrolador ESP32

Um microcontrolador se trata de um computador completo em tamanho reduzido constituído de circuitos integrados. Circuitos integrados são circuitos eletrônicos que utilizam semicondutores, geralmente encapsulados em Silício, ocupando menos espaço físico que circuitos com diversos componentes como resistores, transistores, diodos, entre outros. Um microcontrolador é composto de uma CPU e de componentes que permitem a execução de tarefas de modo autônomo, como conversores analógico/digital, portas programáveis de entrada e saída, memórias para leitura e escrita de dados, entre outros (ARAUJO; CAVALCANTE; SILVA, 2019). Este trabalho utilizará o ESP32.

O ESP32, apresentado na Figura 9, é um microcontrolador desenvolvido pela Espressif composto de um conjunto de chips, que possui Wi-fi e *Bluetooth* integrado, foi projetado para utilização de consumo ultrabaixo de energia e ao mesmo tempo atingir potência e desempenho satisfatórios de rádio frequência (RF) para aplicações móveis, IoT e computação pervasiva (ESPRESSIF, 2020).

Figura 9 - Microcontrolador ESP32

Fonte: RS Robótica, 2020.

Algumas das funções disponíveis: são o controle de *clock* de baixa granularidade, escalonamento de energia dinâmico e modos de energia múltiplos, características de chips de consumo reduzido, além de modo *sleep* que possibilita que o microcontrolador opere somente quando alguma condição preestabelecida seja acionada, ficando "adormecido" e limitando o consumo de energia. O amplificador de potência pode ser ajustado fazendo a compensação entre a taxa de dados, alcance do sinal e consumo energético (ESPRESSIF, 2020).

O ESP32 possibilita a integração de BLE 4.2 e Wi-Fi 802.11 a/b/g/n em aplicações de IoT, com 34 pinos de entrada e saída programáveis, aceleração de hardware criptográfico, com cerca de 20 tipos de componentes externos como amplificadores de potência, amplificadores de recepção de baixo ruído, filtros, interruptores de antena, módulos de gerenciamento de energia e RF *balanced-unbalanced* (balun), entre outros, com a vantagem de ocupar uma área pequena de placa de circuito impresso (PCB) (ESPRESSIF, 2020).

3.2 Componentes de software

Esta seção descreverá os componentes de software para a prototipação das novas funcionalidades a serem adicionadas ao aplicativo.

3.2.1 Android Studio

O Android Studio é um *Integrated Development Environment* (IDE), em português, ambiente de desenvolvimento integrado, para desenvolver aplicativos Android, possuindo

editor de código e ferramentas avançadas. Existem recursos como um emulador de alto desempenho, um ambiente que permite o desenvolvimento de aplicações para todos os tipos de dispositivos com Android, a possibilidade de aplicar alterações de código nos aplicativos enquanto estão em execução sem a necessidade de reiniciá-los, integração com o GitHub, ferramentas de teste e para detecção de possíveis problemas de compatibilidade entre versões, usabilidade, desempenho, entre outras funcionalidades (ANDROID DEVELOPERS, 2020).

O sistema de compilação do Android Studio é o *Gradle*, que é incorporado ao menu do Android Studio, trabalhando separadamente da linha de comando. O sistema propicia a criação de *Android Application Package* (APK) para aplicativos em um mesmo projeto e mesmos módulos, mas com recursos distintos, configuração, personalização e ampliação da programação e reutilização de código (ANDROID DEVELOPERS, 2020).

Os arquivos de projetos são organizados em módulos e podem ser vistos no formato de *Gradle Scripts*. O módulo *Manifests* contém o arquivo *AndroidManifest.xml*; o módulo Java contém os arquivos de código-fonte, incluindo o de teste da JUnit; e o módulo *Resources* contém partes do programa que não são código, como interfaces gráficas, imagens de bitmap e layouts *eXtensible Markup Language* (XML) (ANDROID DEVELOPERS, 2020). Para este trabalho será utilizada a linguagem de programação Kotlin.

3.2.2 Arduino IDE

O Arduino IDE possui editor de código, console, barras de ferramentas, área de notificações e diversos menus. Essa IDE serve para realizar a conexão com o hardware, fazendo upload de programas desenvolvidos e sua comunicação. Esses programas, chamados de *sketches*, são feitos no editor de texto e salvos em formato ".ino" (ARDUINO, 2015).

O editor permite que o texto do código seja escrito, colado, recortado, pesquisado ou substituído. O console serve para exibir informações sobre a compilação, erros e a saída em forma de texto. As barras de ferramentas possuem botões que permitem o carregamento e verificação de *sketches*, abrir, salvar, entre outras funcionalidades (ARDUINO, 2015).

O Arduino IDE permite que o ESP32 seja programado e para que isso seja possível é necessário que reconheça os modelos disponíveis de placa. Nas preferências da IDE deve-se configurar um *Uniform Resource Locator* (URL) que torna possível o acesso a uma base de dados no formato *JavaScript Object Notation* (JSON) contendo diversos modelos de placas. Instalada a placa "esp32 by Espressif Systems", a opção "ESp32 Dev Module" deve ser

selecionada no menu de placas para se programar o ESP32 (USINAINFO, 2019). Foi utilizada a linguagem C para a programação do ESP32 no Arduino IDE.

3.2.3 Google Firebase

Este trabalho utiliza o Google Firebase *Realtime Database* para o armazenamento de dados. O Firebase é um banco de dados que admite que dados sejam armazenados, manipulados e sincronizados em nuvem, sendo do tipo *Not Only Standard Query Language* (NoSQL). Essa abordagem de banco de dados permite grande versatilidade para armazenamento e recuperação de dados, possuindo as tabelas comumente encontradas em outros tipos de bancos de dados, e foram criados a alguns anos, ganhando destaque recentemente devido a popularização da big data, computação em nuvem, aplicativos móveis e web. Seus atributos como desempenho, escalabilidade e facilidade de uso são as principais razões pelas quais são utilizados (IBM, 2020; FIREBASE, 2019a).

Os dados armazenados são salvos em formato JSON e sincronizados em todas as aplicações cliente em tempo real, mantendo-se disponíveis inclusive em modo off-line. Mesmo que os clientes utilizem aplicativos criados em plataformas distintas, os dados serão atualizados de forma automática e estarão nas suas versões mais recentes (FIREBASE, 2019a).

O *Realtime Database* possui um conjunto de regras baseadas em expressão que são as regras de segurança, determinando quando os dados podem ser lidos, gravados e como devem ser estruturados. Aliado ao Firebase *Authentication*, pode ser definido pelo desenvolvedor como os dados podem ser acessados e quem pode acessá-los. Apenas operações que podem ser realizadas com rapidez são permitidas pela API do *Realtime Database*, garantindo que muitos clientes acessem dados em tempo real sem prejudicar os tempos de resposta (FIREBASE, 2019a).

Para que o Firebase seja adicionado nos projetos para Android é necessário que o Android Studio esteja instalado em sua versão mais recente. Também se faz necessário que o aplicativo para Android tenha pelo menos nível 16 da API (*Jelly Bean*) e o *Gradle* pelo menos versão 4.1 (FIREBASE, 2019b).

Para a conexão do aplicativo para Android ao Firebase existem duas opções: utilizar o Firebase Assistente do Android Studio, que requer uma configuração adicional, ou o Console do Firebase, que é a opção recomendada. (FIREBASE, 2019b).

3.3 Componentes da aplicação Android

Esta seção descreverá todos os métodos presentes no código-fonte e as funcionalidades desenvolvidas para o aplicativo.

3.3.1 Login Fragment

A *Login Fragment* contém a apresentação inicial ao se abrir o aplicativo, e como se trata de uma funcionalidade simples, não requer permissões especiais por parte do usuário. Ao ser incluída em um projeto a *Login Fragment template* gera dois arquivos, sendo eles:

- *fragment_login.xml*: este arquivo contém um único *fragment* que é responsável pela parte gráfica, com as instruções sobre o tipo de layout, suas dimensões e seus componentes visuais (ANDROID DEVELOPERS, 2021d);
- *LoginFragment.kt*: este arquivo faz a comunicação entre o a interface gráfica e o código que é executado de forma concomitante (ANDROID DEVELOPERS, 2021d).

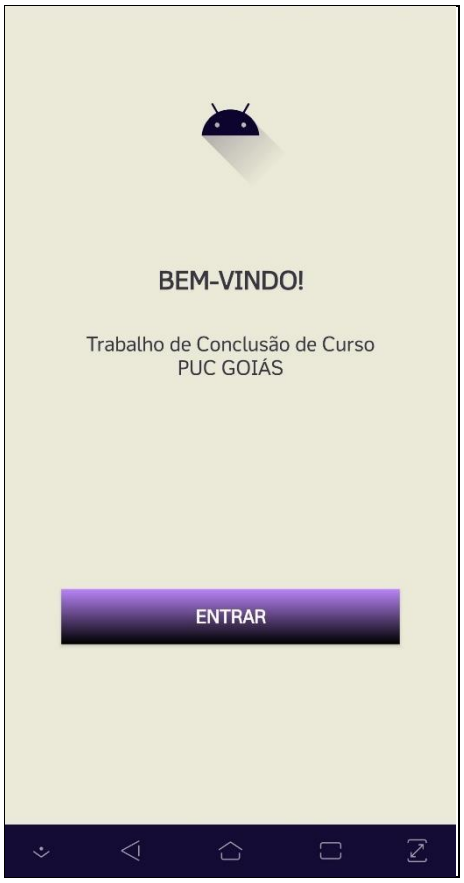
A implementação é composta pelos métodos sobrescritos pela diretiva *override*, que permite que métodos com nomes iguais possam executar parâmetros diferentes, sendo eles *onCreateView*() e *onViewCreated*(), além de dois métodos criados para a aplicação, *onLoading*() e *onResultSuccess*().

O método *onCreateView*() é responsável por "inflar" a interface gráfica, permitindo sua visualização pelo usuário. Sua utilização pode ser opcional, já que as *fragments* podem ter elementos não gráficos, ou seja, apenas código-fonte puro que não necessita ser visualizado pelo usuário. Após o retorno desse método, a *onViewCreated*() é chamado imediatamente, inicializando as subclasses e variáveis a serem utilizadas no decorrer do projeto (ANDROID DEVELOPERS, 2021d). No caso da *Login Fragment* são inicializados o botão "ENTRAR" e a tela de carregamento.

O método *onLoading*() é responsável por controlar o *status* de carregamento da tela ao se apertar o botão, definindo se o componente deve ser exibido ou não, caso esteja em *status* de carregamento a visibilidade é habilitada, caso contrário nada é feito.

E por fim o método *onResultSuccess*() realiza a navegação para a próxima tela, ao se apertar o botão, que será definida pela *Maps Fragment*. A Figura 10 mostra a interface da tela inicial. O código com a implementação se encontra no Apêndice B.

Figura 10 - Interface da *Login Fragment*



Fonte: Elaborado pelo autor.

3.3.2 Maps Fragment

A *Maps Fragment* contém os serviços de localização necessários para a implementação do *geofencing*, através do Google Maps. Para que o SDK do *Maps* para Android seja utilizado é necessária uma chave de API que está atrelada a um certificado digital que vincula ao programador o aplicativo por ele desenvolvido e que deve ser obtida através do registro do projeto na página da Google API e Serviços (ANDROID DEVELOPERS, 2021h).

Para que o uso da localização seja permitido é necessário adicionar as permissões android.permission.ACCESS_FINE_LOCATION, android.permission.ACCESS_COARSE_

LOCATION e android.permission.ACCESS_BACKGROUND_LOCATION no arquivo *AndroidManifest.xml* (ANDROID DEVELOPERS, 2021h). Ao ser incluída em um novo projeto a Google *Maps Fragment template* gera três arquivos:

- *google_maps_api.xml*(*debug*): este arquivo é utilizado para manter a chave de API (ANDROID DEVELOPERS, 2021h).
- *fragment_maps.xml*: este arquivo contém um único *fragment*, que é uma interface de usuário e uma seção modular da *Activity* à qual pertence possuindo seu comportamento e ciclo de vida próprios, podendo ser combinada com outras *fragments*, preenchendo a tela inteira fazendo com que a utilização do mapa no aplicativo seja facilitada, sendo o mapa gerado automaticamente ao se instanciar uma Google *Maps Fragment* (ANDROID DEVELOPERS, 2021d).
- *MapsFragment.kt*: este arquivo permite que as manipulações sejam feitas no mapa, contendo os diversos métodos que fazem com que o usuário tome controle de ações (ANDROID DEVELOPERS, 2021d).

A implementação dessa *fragment* é composta pelos métodos sobrescritos pela diretiva de compilação *override*, sendo eles *onCreateView*(), *onViewCreated*(), *onMapReady*(), *onNavigationItemSelected*(), *onRequestPermissionsResult*() e *onActivityResult*(), e métodos instanciados propriamente para a aplicação *setMapLongClick*(), *setMapStyle*(), *removeAllGeofences*(), *isPermissionGranted*() e *enableMyLocation*(), referentes ao mapa, *getFirebaseData*() e *getAllData*(), referentes ao banco de dados Google Firebase, e a classe interna *ConnectedThread*() com os métodos para a utilização de *Bluetooth*, que serão descritos a seguir.

Os métodos *onCreateView*() e *onViewCreated*() têm o mesmo objetivo dos métodos de mesmo nome presentes na *Login Fragment*. No caso da *Maps Fragment* são inicializados o mapa, uma barra na margem superior da tela denominada *toolbar* e um menu presente na lateral esquerda, denominado *navigation drawer.*

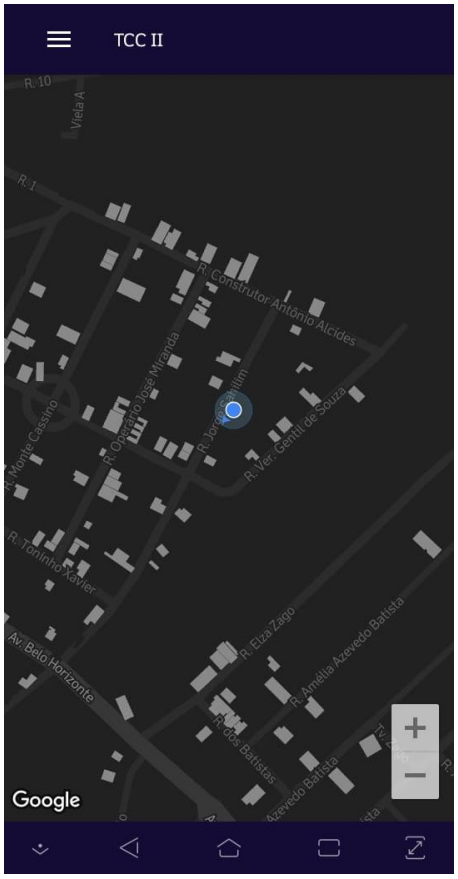
O método *onMapReady*() é chamado quando o mapa está pronto para uso. Para quesitos de teste foram instanciados nesse método os valores de latitude e longitude correspondente a uma residência. Além desses valores, o nível de zoom no mapa é selecionado, sendo que os valores de 1f, 5f, 10f, 15f e 20f correspondem respectivamente a visualização a nível de mundo, continentes, cidades, ruas e construções, as chamadas aos métodos restantes são realizadas e também as configurações da interface do mapa que permitem ao usuário final, manipulá-lo, como a exemplificar controle de zoom por tecla, habilitar bússola, zoom por gesto,

gesto de rolagem, possibilitar a inclinação e a rotação do mapa (ANDROID DEVELOPERS, 2021h).

O método *setMapStyle*() possibilita a estilização, com suas configurações de cores presentes no arquivo map_style.json, permitindo a customização das cores das ruas e avenidas categorizadas em diversos tipos de vias, terreno, regiões lacustres e pontos de interesse (ANDROID DEVELOPERS, 2021h).

O método *onRequestPermissionsResult*() verifica se a permissão de localização está ativada no arquivo do *AndroidManifest.xml* e funciona em conjunto com o método *enableMyLocation*(), que permite a localização do usuário do aplicativo em tempo real e *isPermissionGranted*(), que possibilita a localização precisa através dos serviços de GPS (ANDROID DEVELOPERS, 2021h). Após a criação de todos os componentes descritos anteriormente, a interface é criada como apresenta a Figura 11.

Figura 11 - Interface inicial da *Maps Fragment*



Fonte: Elaborado pelo autor.

O método *setMapLongClick*() faz com que um marcador seja gerado no local onde o usuário pressiona o dedo na tela por alguns segundos, armazenando os valores de latitude e

longitude (ANDROID DEVELOPERS, 2021h). Através do método *addMarker*() um marcador vai ser adicionado na tela que simboliza o centro de uma *geofence* e exibe seu identificador, podendo então ser instanciada uma *geofence* pelo método *addGeofence*().

O método *addGeofence*() gera um círculo que representa graficamente uma *geofence* através de um método chamado *addCircle*(), que utiliza os dados de raio da *geofence*, latitude e longitude, com seu centro na mesma coordenada da *geofence* a ser criada, adicionando ainda cor e espessura de borda. Parar criar efetivamente a *geofence* o método *addGeofence*() *utiliza* um arquivo que se encontra presente fora da *fragment*, o *GeofenceHelper.kt*, que é uma classe em Kotlin composta pelos métodos *getGeofencingRequest*(), *getGeofence*(), *getPendingIntentHelper*() e *getErrorString*(), responsáveis pela criação das *geofences* e sua manipulação (ANDROID DEVELOPERS, 2021c). A classe possui ainda um *companion object,* que é um objeto inicializado apenas quando a classe é carregada.

O método *getGeofencingRequest*() possui um construtor que cria uma *geofence,* que é monitorada pelo *geofencing service*, e especifica como as notificações devem ser relatadas, informando se um dispositivo se encontra dentro da cerca no momento que a mesma é criada (ANDROID DEVELOPERS, 2021c).

O método *getGeofence*() é responsável pela criação da área circular correspondente a *geofence*, a atribuição de um identificador, a criação de alertas de interesse para os tipos de transição especificados, o tempo necessário para se considerar que o dispositivo se encontra dentro da cerca, levando em consideração o momento em que se ultrapassa o limite de entrada, dado em milissegundos, e o tempo de duração da *geofence* (ANDROID DEVELOPERS, 2021c).

O método *getPendingIntentHelper*() utiliza um arquivo externo contendo uma classe em Kotlin chamada *GeofenceBroadcastReceiver.kt*, que possui apenas o método *onReceive*() e um *companion object*. Esse método categoriza os tipos de transição já mencionados anteriormente, sendo eles GEOFENCE_TRANSITION_ENTER, que informa se um dispositivo adentrou a *geofence*, GEOFENCE_TRANSITION_DWELL, que informa se o dispositivo permanece dentro da cerca por um período de tempo e GEOFENCE_TRANSITION_EXIT, que informa se o dispositivo saiu da geofence (ANDROID DEVELOPERS, 2021c).

O método *getErrorString*() reconhece exceções, informando caso uma *geofence* não esteja disponível, ocasionada pela desativação dos serviços de localização nas configurações através da ação do usuário, se o limite de *geofences* foi alcançado, a API permite a utilização de até cem *geofences* simultaneamente, para que novas cercas sejam utilizadas é necessário a

remoção das que não estão em uso, e se estão sendo utilizados mais *Pending Intents* do que o permitido, sendo no total de cinco, estes sendo *token*s fornecidos a uma aplicação externa que permitem a utilização de permissões da aplicação local para a execução de código-fonte pré definido (ANDROID DEVELOPERS, 2021c).

Após a chamada ao método *addGeofence*() ser realizada, uma instância do banco de dados Google Firebase é criada, permitindo que a *geofence* tenha seus dados armazenados no *Realtime Database* utilizando-se dos mesmos parâmetros, sendo eles o identificador, as coordenadas de latitude e longitude e o raio.

Um objeto do tipo *DatabaseReference* cria o caminho para o armazenamento utilizando o comando *.child*, que é cumulativo e recebe um identificador. Esses identificadores possibilitam a manipulação dos dados advindos do *Realtime Database*. A Figura 12 mostra a organização das *geofences* no banco de dados (FIREBASE, 2020a).

Figura 12 - Organização dos dados armazenados no *Realtime Database*

Fonte: Elaborado pelo autor.

O primeiro *child* trata do identificador geral dos blocos de *geofences*, chamado *Geofences,* que será utilizado para a manipulação dos dados, os outros *childs* são gerados automaticamente com o auxílio de um contador, conferindo o formato de texto de *Geofences* somado a um contador inteiro que é incrementado toda vez que uma *geofence* nova é criada.

Para que o valor seja inserido no *child* o mesmo é estendido com o comando *.setValue*(), que recebe como parâmetro um objeto do tipo *LocationData*, que está configurado em um arquivo separado, em Kotlin, contendo uma classe chamada *LocationData.kt*, que cria

as variáveis que serão utilizadas no armazenamento e seu construtor, que é chamado sempre que uma nova geofence for armazenada no banco de dados (FIREBASE, 2020a).

Tanto *fragments* quanto *activities* possuem um ciclo de vida, que se encerra quando ocorre a navegação entre telas ou se há uma mudança de orientação de retrato para paisagem por exemplo (ANDROID DEVELOPERS, 2021d). Visto que as *geofences* precisam ser mantidas no mapa por tempo indeterminado até que o usuário da aplicação deseje removê-las, é necessário que sejam salvas para que possam ser geradas novamente e automaticamente quando o usuário retornar para a tela principal da *Maps Fragment*. Para isso é utilizado o método *getFirebaseD*ata().

Esse método se encontra dentro de outro método, o *onViewCreated*(), que vai executar todos os comandos presentes em seu interior para a criação das características visuais. O *getFirebaseData*() vai acessar o *Realtime Database* e procurar pelo *child* referente a *Geofences*. Quando for encontrado, o método sobrescrito por *override*, *onDataChange*(), vai verificar se o conteúdo do banco de dados nesse *child* é vazio e caso seja, a *Maps Fragment* será inicializada normalmente, caso contrário o método *getAllData*() será invocado e vai procurar pelos dados até que todos tenham sido lidos, e à medida que isso ocorre, o método *addGeofence*() vai adicionar cada uma das geofences presentes no banco de dados.

O método *removeAllGeofences*() é responsável por remover todas as *geofences* presentes no mapa e suas respectivas representações gráficas.

O método *onNavigationItemSelected*() é utilizado para o gerenciamento de todos os botões do *navigation drawer*, fazendo a ligação entre os botões e os respectivos métodos que fazem com que o usuário possa realizar ações dentro do aplicativo. Cada item do menu possui um identificador próprio e que, caso o usuário clique em algum botão correspondente, será disparado um *itemId* que será então reconhecido, e para cada um deles uma ação distinta ocorrerá, sendo elas criação de novas *geofences* com ou sem a integração com o ESP32, remoção de geofences ou navegação entre telas, tanto para autenticação quanto para sair do aplicativo.

Dentro deste método o item relacionado ao carro possui características em comum aos demais botões para a criação de *geofences*, sendo eles de casa e trabalho, porém a maneira como é criada difere das demais devido a utilização do *Bluetooth* para tal. Para que o uso do *Bluetooth* seja possível são necessárias que as permissões android.permission.BLUETOOTH, android.permission.BLUETOOTH_ADMIN e android.permission.LOCAL_MAC_ADDRESS sejam adicionadas no arquivo do *AndroidManifest.xml* (ANDROID DEVELOPERS, 2021i).

O objeto correspondente ao adaptador de *Bluetooth* é iniciado no método *onViewCreated*() e através dele é verificado se o mesmo existe no dispositivo ao qual o aplicativo está instalado. Caso não exista, uma mensagem irá informar o usuário da inexistência do adaptador. Caso o *Bluetooth* esteja desativado, um *token* será disparado requisitando a sua ativação, que será analisada no método *onActivityResult*() e retornará a mensagem informando se o dispositivo se encontra com o *Bluetooth* ativado após a confirmação do usuário (ANDROID DEVELOPERS, 2021i).

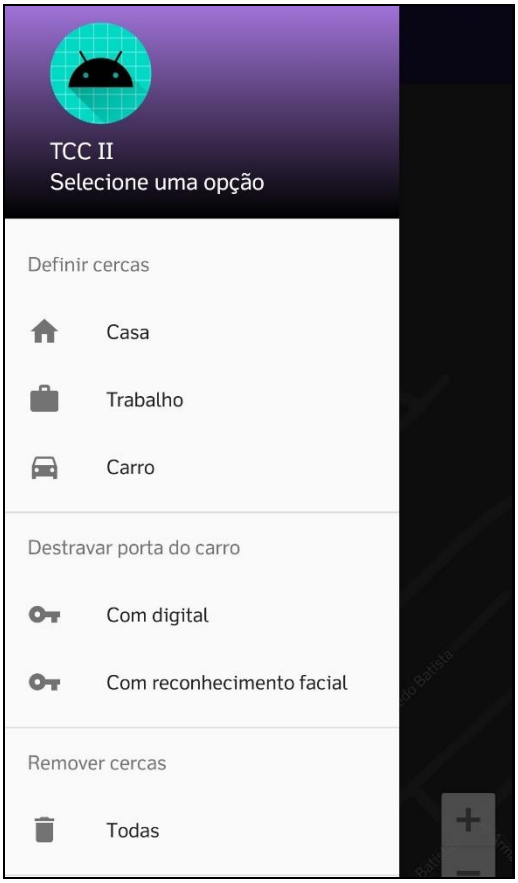
Após a ativação, será feita a tentativa de se conectar o dispositivo com o ESP32 e caso ocorra de maneira correta, o endereço MAC será enviado através do método *write*() presente na classe *ConnectedThread*(), que recebe o endereço e o transforma em um vetor de *bytes* para ser enviado (ANDROID DEVELOPERS, 2021i). Realizado o envio é aguardado o retorno da confirmação da validação do endereço MAC pelo ESP32, que se for bem-sucedida, irá enviar novamente um código de confirmação para o ESP32 fazer a abertura da porta e criar a *geofence* pelo próprio aplicativo através do método *addGeofence*(), na última localização conhecida do usuário.

Uma instância do banco de dados será criada para salvar a *geofence,* e a navegação para a tela de autenticação por impressão digital será feita, retornando para a tela principal da *MapsFragment* caso seja autenticado.

Descritos todos os métodos ocultos da visualização do usuário final, as funcionalidades serão descritas a seguir.

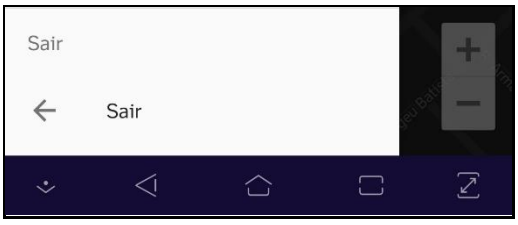
O *navigation drawer* pode ser aberto ao se deslizar o dedo no sentido da margem esquerda para a margem direita ou ao se clicar no componente gráfico composto de 3 barras alinhadas horizontalmente, e possui uma série de opções clicáveis que permitem a definição de *geofences* atribuídas a casa, ao trabalho e ao carro do proprietário, além das opções de destravar a porta do carro por biometria de impressão digital ou reconhecimento facial, a remoção de todas as *geofences* e sair do aplicativo. As opções ocultas da visualização do usuário podem ser acessadas ao rolar o menu para baixo, como mostra as Figuras 13a e Figura 13b.

Figura 13a - Interface do *navigation drawer*



Fonte: Elaborado pelo autor.

Figura 13b - Interface do *navigation drawer*

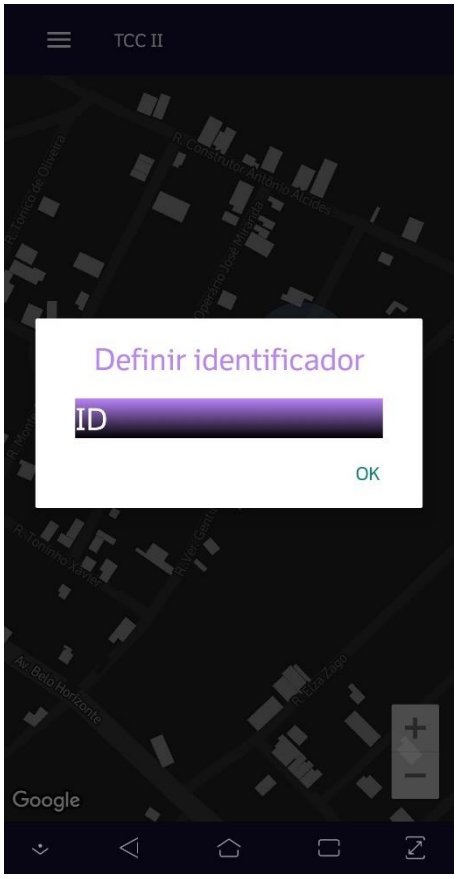


Fonte: Elaborado pelo autor.

As *geofences* da casa e do trabalho poderão ter seus locais definidos pelo usuário livremente, porém a *geofence* relacionada ao carro será definida automaticamente ao se parear o dispositivo a um microcontrolador ESP32 através de *Bluetooth*.

Ao se clicar em Casa ou Trabalho, o *navigation drawer* será fechado e uma janela *pop-up* será aberta, solicitando que o usuário insira um identificador para a geofence pretendida, não sendo permitido que o usuário retorne a tela anterior enquanto não definir o identificador e apertar o botão “OK”. A interface da janela é mostrada na Figura 14.

Figura 14 - Janela para definição de identificador de *geofence*



Fonte: Elaborado pelo autor.

Feito isso, o usuário poderá executar um clique longo sobre o mapa, que vai criar a *geofence*, tanto visualmente como virtualmente, e será exibida uma mensagem comprovando que a *geofence* foi adicionada com sucesso.

Ao se clicar em Carro, o usuário deverá inserir um identificador para a *geofence* assim como ocorre nas demais, porém ao invés de ser utilizado o clique longo sobre o mapa, uma mensagem pedindo a permissão para a ativação do *Bluetooth* será exibida, e após a ativação, o usuário deverá conectar o dispositivo com o ESP32 que se encontra dentro do carro do proprietário. O sucesso da operação será confirmado com o acendimento de um Diodo Emissor de Luz (LED) na cor branca fria.

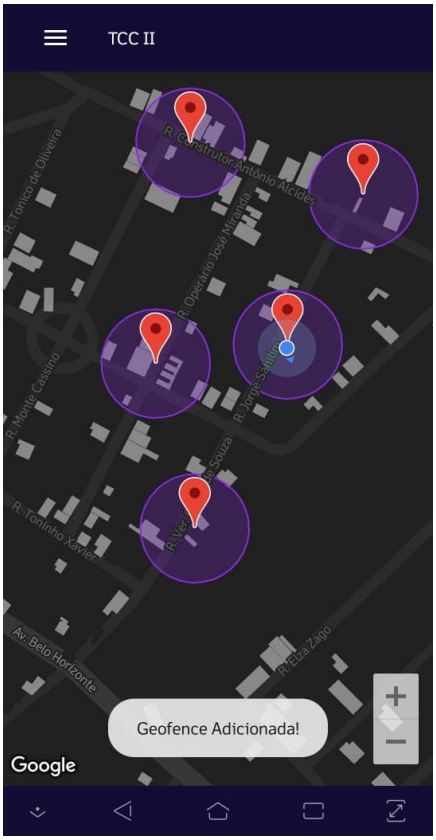
Quando realizada a conexão, o dispositivo enviará automaticamente seu endereço MAC para o ESP32 por meio de *Bluetooth* para ser comparado com o endereço inserido em seu código-fonte. Caso os endereços sejam iguais, um LED na cor azul acenderá indicando que o dispositivo foi autenticado e então uma mensagem será enviada de volta ao dispositivo, contendo o caractere “1”, que é a confirmação para que o aplicativo crie a *geofence* na última localização conhecida do usuário.

Após o término do procedimento, o usuário necessita ainda de comprovar sua identidade através da autenticação por impressão digital, para que finalmente possa retornar a tela que contém o mapa. Será enviado também de volta ao ESP32 o caractere “1”, para que possa realizar o destrave da porta e o LED na cor branca quente será aceso para indicar o destrave da porta do carro. Caso qualquer operação no código do ESP32 indicar qualquer incoerência, o usuário será desconectado automaticamente.

Para que seja possível a conexão com o ESP32, o usuário deve estar o mais próximo possível do carro devido ao alcance limitado do sinal de *Bluetooth*, contribuindo para o aumento da segurança na realização do processo de abertura da porta.

As *geofences* da casa e trabalho possuem ambas um raio de 40 metros, enquanto a *geofence* do carro possui um raio de 10 metros. O raio da circunferência gerada possui tamanho fixo e não pode ser alterado. As *geofences* definidas no mapa são demonstradas pela Figura 15.

Figura 15 - *Geofences* adicionadas no mapa



Fonte: Elaborado pelo autor.

Ao se clicar no botão “Todas” no tópico remover cercas, as *geofences* presentes no mapa serão deletadas, assim como no *Realtime Database*, através da seleção do *child* principal,

o *Geofences*, e utilizando o comando *removeValue*(). O contador de *geofences* será reiniciado para que, ao serem criadas novamente, possa iniciar na *geofence* de número um. O código com a implementação completa se encontra no Apêndice C.

3.3.3 FingerPrint Fragment

A *FingerPrint Fragment* contém parte da implementação destinada à validação do usuário do aplicativo, realizando a verificação biométrica por impressão digital. Por motivos de simplicidade e reaproveitamento do software embutido nos dispositivos alvo da aplicação, as impressões digitais já cadastradas pelo usuário no seu dispositivo serão utilizadas para validar ações dentro do app, sem a necessidade de implementar novamente tal função.

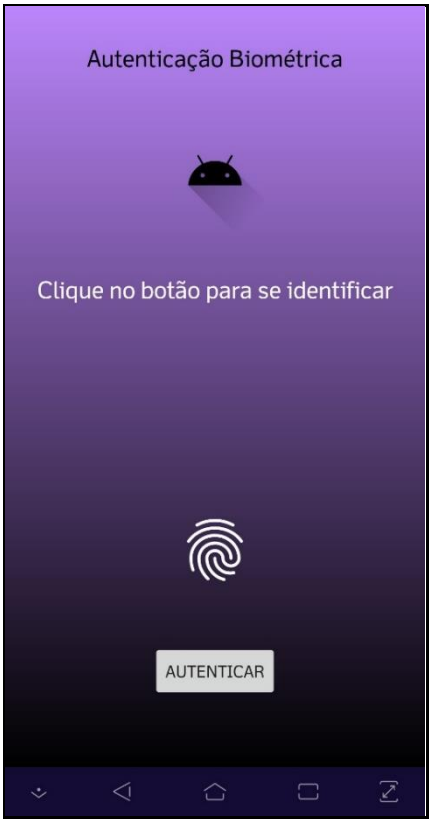
Para que a verificação biométrica seja utilizada é necessário adicionar a permissão android.permission.USE_BIOMETRIC no arquivo *AndroidManifest.xml* (ANDROID DEVELOPERS, 2021f). Ao ser incluída em um projeto a *FingerPrint Fragment template* gera dois arquivos, sendo eles:

- *fragment_finger_print.xml*: este arquivo contém um único *fragment* que é responsável pela janela de autenticação biométrica, com as instruções sobre o tipo de layout, suas dimensões e seus componentes visuais (ANDROID DEVELOPERS, 2021d);
- *FingerPrintFragment.kt*: este arquivo faz a comunicação entre o a interface gráfica e o código que é executado de forma concomitante (ANDROID DEVELOPERS, 2021d).

A implementação da autenticação biométrica na *FingerPrint Fragment* é feita pelos métodos sobrescritos pela diretiva *override*, *onCreateView*(), *onViewCreated*(), onAuthenticationError(), onAuthenticationSucceeded() e onAuthenticationFailed().

A inicialização da *FingerPrint Fragment* ocorre da mesma maneira que na *Maps Fragment,* com os métodos *onCreateView*() e *onViewCreated*(), realizando a integração dos identificadores da interface gráfica com o código para a manipulação dos dados e a verificação de forma a possibilitar a continuidade do código. A interface exibida na Figura 16 será chamada quando o usuário realizar a solicitação para a abertura da porta do carro.

Figura 16 - Interface da *FingerPrint Fragment*

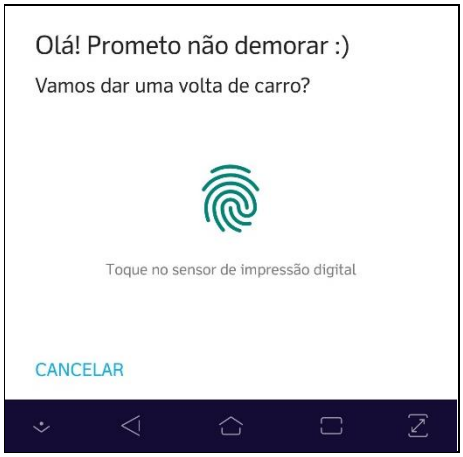


Fonte: Elaborado pelo autor.

Após a abertura da interface é verificada a possibilidade de autenticação, que gerará um *token* para cada uma das condições existentes, sendo eles:

BIOMETRIC_SUCCESS, indicando que todos os requisitos para a utilização do sensor estão sendo cumpridos. Ao se apertar o botão "autenticar" será exibido um *prompt*, que é gerado automaticamente pela classe *BiometricPrompt* , como mostra a Figura 17, para que o usuário possa tocar o sensor e avançar no aplicativo (ANDROID DEVELOPERS, 2021f);

Figura 17 - Janela para inserção de impressão digital



Fonte: Elaborado pelo autor.

BIOMETRIC_ERROR_NO_HARDWARE, sinalizando que o dispositivo não possui leitor de impressão digital. A frase de subtítulo “Clique no botão para se identificar” será substituída por “O dispositivo não possui leitor de impressão digital.”. O botão “autenticar” não estará mais disponível, repetindo o padrão para as demais condições (ANDROID DEVELOPERS, 2021f);

BIOMETRIC_ERROR_HW_UNAVAILABLE, indicando que o hardware para a leitura se encontra indisponível no momento, sendo o subtítulo substituído por “Os sensores estão atualmente indisponíveis.” (ANDROID DEVELOPERS, 2021f);

BIOMETRIC_ERROR_NONE_ENROLLED, determinando que não existem impressões digitais cadastradas. O subtítulo será alterado para “Impressões digitais não cadastradas! Por favor as cadastre nas configurações de segurança.”, sendo, portanto, necessário adicionar pelo menos uma impressão digital nas configurações do dispositivo para que o usuário possa prosseguir (ANDROID DEVELOPERS, 2021f);

E por fim, *BIOMETRIC_ERROR_SECURITY_UPDATE_REQUIRED*, acusando que o dispositivo possui baixos níveis de segurança, substituindo o subtítulo por “Vulnerabilidade de segurança encontrada! Por favor as cadastre uma senha para o dispositivo.”, sugerindo que seja cadastrado pelo menos um método de bloqueio de tela no dispositivo para prosseguir (ANDROID DEVELOPERS, 2021f).

Os métodos *onAuthenticationError*(), *onAuthenticationSucceeded*() e *onAuthenticationFailed*() são responsáveis por verificar o status da autenticação, sendo utilizados respectivamente para indicar se um possível erro ocorreu e exibi-lo, se a autenticação foi realizada com sucesso e se houve alguma falha no processo (ANDROID DEVELOPERS,

2021f). Finalizado todo o processo, a *FingerPrint Fragment* será encerrada e retornará a *Maps Fragment.* O código com a implementação se encontra no Apêndice D.

3.3.4 FacialRecognition Fragment

A *FacialRecognition Fragment* possui a implementação de código para a detecção facial do usuário, utilizando o *Machine Learning Kit Face Detection* (MLKit) do Firebase. Inicialmente foi idealizada a utilização desta *fragment* para realização do reconhecimento facial do usuário, porém a implementação se restringe a detecção do rosto devido as limitações de tempo para a apresentação de um trabalho com esse nível de complexidade.

Para que possa ser utilizado é necessário a permissão para a utilização da câmera do dispositivo, que pode ser obtida adicionando-se a permissão android.permission.CAMERA no arquivo *AndroidManifest.xml* (ANDROID DEVELOPERS, 2021e). Ao ser incluída em um projeto a *FacialRecognition Fragment template* gera dois arquivos:

- *fragment_facial_recognition.xml*: este arquivo contém um único *fragment* que é responsável pela janela de reconhecimento facial, com as instruções sobre à utilização da câmera e um botão (ANDROID DEVELOPERS, 2021d);
- *FacialRecognitionFragment.kt*: este arquivo faz a comunicação entre a interface gráfica e o código que é executado de forma concomitante (ANDROID DEVELOPERS, 2021d).

O MLKit é um SDK que permite a utilização de técnicas de aprendizado de máquina em dispositivos móveis. A API *Vision* utilizada na programação deste aplicativo permite a detecção de rostos, identificando características para a obtenção dos contornos faciais. Essa API não realiza o reconhecimento de pessoas, mas sim detecta rostos em imagens, podendo ser fotos estáticas ou vídeo (ANDROID DEVELOPERS, 2021e). É possível utilizar o MLKit para realizar a implementação de reconhecimento de pessoas, todavia o mesmo não foi implementado neste trabalho.

Das funcionalidades que o MLKit oferece, tem-se a localização das características faciais através das coordenadas dos lábios, nariz, olhos, orelhas e bochechas, para cada rosto a ser detectado e seus contornos, o reconhecimento de expressões como olhos fechados ou sorrisos, o rastreio de rostos em vídeo com a atribuição de um identificador que permite que o rosto de uma pessoa específica possa ser manipulado durante o fluxo e a possibilidade de ser utilizado em aplicações de vídeo em tempo real (ANDROID DEVELOPERS, 2021e).

Seguindo o padrão das outras *fragments* a *FacialRecognition Fragment* possui os métodos *onCreateView*() e *onViewCreated*(), esse último inicializa o botão para ser efetuada a detecção facial ao ser clicado, o *alert dialog*, que exibe uma janela *pop-up* solicitando que o usuário aguarde enquanto a detecção é processada e o *graphic overleay*, que faz a renderização dos gráficos, sendo implementado a parte dentro do arquivo *GraphicOverlay.kt* que possui uma classe de mesmo nome, utilizada na *FacialRecognition Fragment*. Esta *fragment* também utiliza o arquivo *RectOverlay.kt*, que terão seus métodos descritos a seguir.

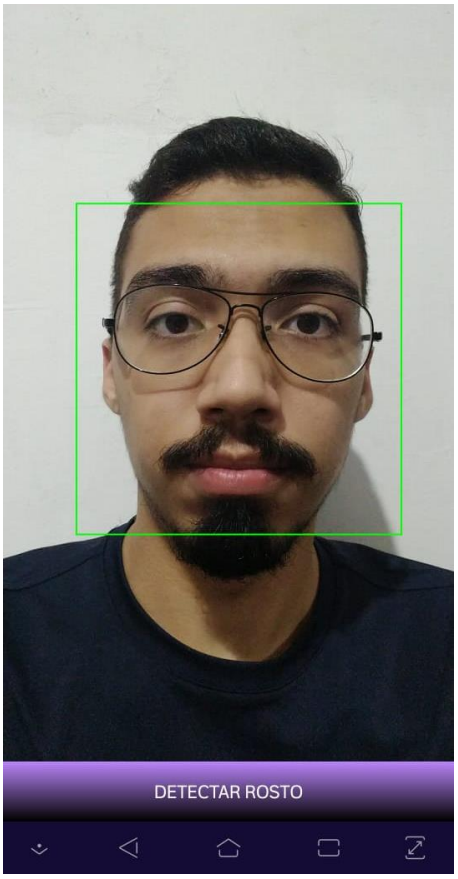
Existem outros métodos sobrescritos por *override*, sendo eles *onEvent*(), *onError*(), *onVideo*(), *onPause*(), utilizado para pausar o processo e *onResume*(), para retornar de onde parou (ANDROID DEVELOPERS, 2021e)..

O método *onImage*() faz o mapeamento de bits proveniente da câmera após ter o botão "DETECTAR ROSTO" apertado, exibe a janela *pop-up* solicitando que o usuário aguarde o carregamento, pausa a utilização da câmera em segundo plano, faz a chamada ao método *processFaceDetection*(), que utiliza a API *Vision* do Firebase para a detecção da face e que, caso tenha êxito, chama o método *getFaceResults*(), que retira a janela da tela e chama o método *onResultSuccess*(), exibindo uma mensagem confirmando a detecção com sucesso e navegando de volta para a *Maps Fragment* (ANDROID DEVELOPERS, 2021e).

A classe *GraphicOverlay* possui a classe abstrata *Graphic* que implementa os métodos *draw*(), para desenhar o gráfico na tela fornecida, *scaleX*(), que ajusta a escala da imagem de acordo com o eixo X da tela, *scaleY*(), que ajusta de acordo com o eixo Y, *translateX*(), que ajusta o sistema de coordenadas do eixo X da tela, *translateY*(), que ajusta as do eixo Y, *clear*(), que remove todos os gráficos gerados, *add*(), que adiciona todos os gráficos a tela, *remove*(), que remove um gráfico específico, *setCameraInfo*(), que define características como o tamanho da face e sua direção para a transformação de coordenadas e *onDraw*(), que faz a sobreposição de gráficos. Todos esses métodos atuam em conjunto para que haja o funcionamento correto do código (ANDROID DEVELOPERS, 2021e).

A classe *RectOverlay* possui apenas um método sobrescrita por *override*, a *draw*(), que se limita a desenhar um quadrado de bordas verdes sobre o rosto detectado, atuando em conjunto com a *GraphicOverlay*. A Figura 18 mostra a interface da *FacialRecognition Fragment* após o usuário apertar o botão e ter seu rosto detectado. Podem ser detectados mais de um rosto caso existam mais pessoas no alcance da câmera (ANDROID DEVELOPERS, 2021e). O código com a implementação completa se encontra no Apêndice E.

Figura 18 - Interface pós detecção facial



Fonte: Elaborado pelo autor.

3.3.5 Main Activity

A *Main Activity* difere das demais estruturas do código, pois não se trata de um *fragment*. Ambos *fragments* e *activities* possuem elementos gráficos e características em comum, porém os *fragments* precisam, necessariamente, estar associados a uma *activity* para que possam ser apresentados, o contrário não ocorre (ANDROID DEVELOPERS, 2021a).

Ao se utilizar *fragments* é possível obter uma maior flexibilidade, devido a maior capacidade de reutilização de código e permitindo a criação de interfaces de aplicação mais complexas ao centralizá-los em uma única *activity*. Ao ser incluída em um novo projeto a *Main Activity template* gera dois arquivos:

- *activity_main.xml*: este arquivo é responsável pela configuração da navegação entre as telas, referenciando o arquivo *main_graph.xml* que faz a conexão das telas (ANDROID DEVELOPERS, 2021a);

- *MainActivity.kt*: este arquivo faz a comunicação entre o a interface gráfica e o código que é executado de forma concomitante (ANDROID DEVELOPERS, 2021a).

A inicialização da *Main Activity* ocorre de maneira semelhante à das outras *fragments*, porém utilizando o método *onCreate*() ao invés dos outros dois métodos, não existindo nenhum *layout* que precise ser inflado, como ocorre nas *fragments* (ANDROID DEVELOPERS, 2021a).

Além desse método existem mais dois que também são sobrescritos pela diretiva *override*, *onSupportNavigateUp*(), é acionado a cada vez que o usuário opta por navegar entre as telas e *onConfigurationChanged*(), que faz com que o conteúdo da *Activity* não seja recriado ao se modificar a orientação da tela de retrato para paisagem e vice-versa (ANDROID DEVELOPERS, 2021b). Cada vez que é chamada, a *Activity* se destrói e é recriada novamente por padrão, sendo necessário salvar o estado para que os dados não sejam perdidos. O código com a implementação se encontra no Apêndice F.

3.4 Componentes da aplicação do ESP32

Esta seção descreverá o que foi feito para a utilização do ESP32 em conjunto com o aplicativo de Android.

3.4.1 Hardware em conjunto com o sistema embarcado

Para se fazer uso do ESP32 alguns componentes de hardware foram utilizados em conjunto, sendo eles uma matriz de contatos de 1660 pontos, para que o ESP32 pudesse ser acoplado juntamente com resto dos componentes, 7 cabos *wire jumper* de 20 centímetros do tipo micro-banana macho-macho, 3 LEDs nas cores branco frio, azul e branco quente, um de cada, com tensão nominal de 3 a 3,4 volts e corrente de operação de 20 a 25 miliampères e 3 resistores com resistência de 150 ohms, que foram dimensionados de acordo com a Lei de Ohm, visto que o ESP32 possui tensão de saída de 3,3 volts.

3.4.2 Software do sistema embarcado

A implementação do software consiste em utilizar como base a implementação de exemplo presente na própria Arduino IDE. Ao se adicionar o módulo ESP32 Dev Module como placa alvo para o desenvolvimento, os exemplos para a placa em questão ficam disponíveis, ente eles o BLE_uart, transformando o ESP32 em um servidor de *Bluetooth*, que ao receber uma conexão poderá enviar dados de volta a quem enviou à medida que estiverem conectados, na arquitetura cliente-servidor, sendo o dispositivo do usuário a aplicação cliente (ANDROID DEVELOPERS, 2021h).

Cada atributo no BLE é identificado exclusivamente por um identificador chamado *Universally Unique Identifier* (UUID), que tem seu formato padronizado com o tamanho de 128 bits em formato de texto, *string*, para categorizar a informação, podendo ser característica ou serviço. As características contêm valores e podem ser consideradas um tipo, como uma classe, já os serviços são um conjunto de características (ANDROID DEVELOPERS, 2021h).

As operações dentro do código são feitas por meio de *callbacks*, que vão determinar os estados de conexão e escrita de dados. O método *BLEServerCallbacks* contém o código responsável pelo servidor, determinando se existe um dispositivo conectado ou não, caso esteja conectado o LED branco frio será ligado, uma variável irá receber o endereço MAC, confirmar se o endereço está correto, através da validação, e informar o estado, caso não seja o dispositivo que se procura, o mesmo será desconectado. Caso não esteja conectado, o LED permanecerá apagado.

Para lidar com as características o método *BLECharacteristicCallbacks* vai receber o MAC e exibi-lo. Caso a variável *state* contenha o caractere “1”, que foi inserido ao se validar o MAC, o LED azul vai acender indicando que foi autenticado e notificará o aplicativo, enviando também o caractere “1” para que possa ser criada a *geofence* pelo aplicativo. Criada a *geofence*, o ESP32 receberá novamente o caractere “1”, para que possa realizar a abertura da porta, após a autenticação no aplicativo por impressão digital, acendendo o LED branco quente. Finalizado o processo todos os LEDs serão apagados.

No bloco de código *setup* serão configurados os LEDs para que possam ser utilizados como saída, assim como a definição do nome do ESP32 visível para a conexão com o usuário, a configuração para manipulação de características, as *callbacks* e a inicialização do servidor. O código com a implementação se encontra no Apêndice A

4 CONSIDERAÇÕES FINAIS

O presente trabalho propôs-se a dar prosseguimento ao trabalho feito por Gabriel Loureiro (LOUREIRO, 2019), tomando como base as sugestões de trabalhos futuros, visando a utilização de um microcontrolador ESP32, *Bluetooth*, e validação de usuário por técnicas de biometria de impressão digital ou facial.

Ao longo do desenvolvimento da aplicação consegue-se perceber claramente como a linguagem de programação Kotlin é uma ferramenta poderosa para o desenvolvimento, com uma sintaxe de fácil entendimento que permite maior clareza na manipulação dos dados.

O IDE do Android Studio mostrou-se um ambiente de desenvolvimento muito completo, com amplo acesso a informação a respeito da linguagem de programação, seus métodos e classes, a integração das APIs é feita de modo facilitado, bem como a criação das telas e navegação. A comunidade dos desenvolvedores Android também se mostrou bastante prestativa, tanto em sites de terceiros como o da própria Google, proprietária do Android, com soluções precisas para grande parte dos problemas enfrentados.

O Firebase foi de grande importância para este trabalho, visto que devido a sua utilização é possível salvar os dados utilizados no *geofencing* remotamente e em tempo real, suprimindo a necessidade de salvar os dados localmente no dispositivo do usuário. Sua documentação é de fácil acesso e o armazenamento de dados na nuvem através do código é feita por comandos simples. A configuração das regras para a leitura e escrita de dados do *Realtime Database*, na página do Firebase Console, são de fácil entendimento e o plano gratuito para o armazenamento atende as necessidades do projeto.

Houve um grande avanço em relação à versão anterior do aplicativo, agora escrito em Kotlin, linguagem recomendada pela Google para o desenvolvimento de aplicativos para Android, bem como melhorias na apresentação visual e em segurança, devido as autenticações de usuário por biometria, a remodelagem de algumas ações dentro do aplicativo, como o menu lateral, e a adição de novas funcionalidades relacionadas a criação das *geofences*.

Das dificuldades enfrentadas cabe citar a dificuldade na obtenção do endereço MAC proveniente do dispositivo Android, visto que a partir da versão 6.0 *Marshmallow* mudanças foram realizadas a fim de prover maior segurança aos usuários, removendo a possibilidade de obtenção de ambos os endereços MAC, de Wi-Fi e *Bluetooth*, através de codificação, retornando um valor padrão fixo de 02:00:00:00:00:00, que não corresponde ao endereço real. Todavia, é possível obter esses endereços ao se conectar a um dispositivo por *Bluetooth*, ficando

destacado o endereço abaixo dos dispositivos que podem ser conectados, ou observando as configurações do smartphone na parte destinada as informações do sistema, no caso do endereço MAC de Wi-Fi (ANDROID DEVELOPERS, 2021g).

Para que esse problema fosse contornado, a obtenção do endereço MAC programaticamente foi substituída pela passagem do endereço em formato de texto diretamente para o ESP32, observado ao se parear o dispositivo com o ESP32 na realização os testes de conectividade. Os testes foram realizados com o smartphone do desenvolvedor, com especificações técnicas: o sistema operacional Android 9.0 *Pie*, leitor de impressão digital, câmeras frontal e traseira, com processador *octacore* de 2,21 GHz, 4 gigabytes (GB) de memória *Random Access Memory* (RAM) e 128 GB de armazenamento.

Este trabalho possibilitou um grande aumento do conhecimento, tanto a respeito do funcionamento dos dispositivos Android quanto ao desenvolvimento de aplicações móveis utilizando a linguagem Kotlin, além de tudo o que foi utilizado como base teórica para a criação do aplicativo. O projeto apresenta um grande potencial para ter seu desenvolvimento continuado, abrindo margem para a adição de novas funcionalidades no que diz respeito ao IoT. Os objetivos propostos foram cumpridos.

4.1 Trabalhos futuros

Como trabalhos futuros sugere-se:

- Realizar a implementação completa da autenticação por reconhecimento facial;
- Modificar a opção de abertura da porta do carro para que ambos os métodos de autenticação, por impressão digital e reconhecimento facial, estejam presentes em um mesmo botão, e sejam chamados de modo que o usuário não possa escolher como quer se autenticar para aumentar a segurança;
- Criar a opção de remover uma *geofence* específica de acordo com seu identificador;
- Implementar o *login* e o *logout* do usuário no aplicativo através de sua conta Google;
- Implementar a possibilidade de o usuário poder definir o tamanho do raio da *geofence* dinamicamente;
- Utilizar de métodos de criptografia no código para aumentar a segurança dos dados.

REFERÊNCIAS

AFANEH, Mohammad. **Ellisys *Bluetooth Video 1: Intro to Bluetooth Low Energy***. Youtube: Ellisys, 2018. (8:05 min.), son., color. Legendado. Disponível em: https://www.bluetooth.com/bluetooth-resources/intro-to-bluetooth-low-energy/. Acesso em: 14 set. 2020.

ALVES, Paulo Victor Alexandre. **Verificação automática georreferenciada**. 2018. 77 f. TCC (Graduação) - Curso de Engenharia de Computação, Escola de Ciências Exatas e da Computação, Pontifícia Universidade Católica de Goiás, Goiânia, 2018.

ANDREZZA, Igor Lucena Peixoto. **Análise de técnicas de normalização aplicadas ao reconhecimento facial**. 2015. 75 f. Dissertação (Mestrado) - Curso de Programa de Pós-Graduação em Informática, Centro de Informática, Universidade Federal da Paraíba, João Pessoa, 2015. Disponível em: https://repositorio.ufpb.br/jspui/handle/tede/7883. Acesso em: 26 out. 2020.

ANDROID DEVELOPERS. **Activity**. 2021a. Disponível em: https://developer.android.com/reference/android/app/Activity. Acesso em: 10 mar. 2021.

ANDROID DEVELOPERS. **Android Studio: Conheça o Android Studio**. 2020. Disponível em: https://developer.android.com/studio/intro?hl=pt-br. Acesso em: 09 nov. 2020.

ANDROID DEVELOPERS. **Atualizar componentes da IU com NavigationUI**. 2021b. Disponível em: https://developer.android.com/guide/navigation/navigation-ui?hl=pt-br. Acesso em: 17 abr. 2021.

ANDROID DEVELOPERS. **Criar e monitorar fronteiras geográficas virtuais**. 2021c. Disponível em: https://developer.android.com/training/location/geofencing. Acesso em: 30 mar. 2021.

ANDROID DEVELOPERS. **Fragmentos**. 2021d. Disponível em: https://developer.android.com/guide/components/fragments?hl=pt-br. Acesso em: 10 mar. 2021.

ANDROID DEVELOPERS. **MLKit: Face Detection**. 2021e. Disponível em: https://developers.google.com/ml-kit/vision/face-detection. Acesso em: 10 maio 2021.

ANDROID DEVELOPERS. **Mostrar caixa de diálogo de autenticação biométrica**. 2021f

Disponível em: https://developer.android.com/training/sign-in/biometric-auth. Acesso em: 30 abr. 2021.

ANDROID DEVELOPERS. **Mudanças do Android 6.0**. 2021g. Disponível em: https://developer.android.com/about/versions/marshmallow/android-6.0-changes.html#behavior-hardware-id. Acesso em: 15 abr. 2021.

ANDROID DEVELOPERS. **Advanced Android in Kotlin 04.1: Android Google Maps**. 2021h. Disponível em: https://developer.android.com/codelabs/advanced-android-kotlin-training-maps#0. Acesso em: 10 mar. 2021.

ANDROID DEVELOPERS. **Visão geral do *Bluetooth* de baixa energia**. 2021i. Disponível em: https://developer.android.com/guide/topics/connectivity/bluetooth-le?hl=pt-br. Acesso em: 10 abr 2021.

ARAÚJO, Izaura. **Impressões digitais**. 2019. Escola Educação. Disponível em: https://escolaeducacao.com.br/impressoes-digitais/. Acesso em: 22 out. 2020.

ARAUJO, Warley Monteiro; CAVALCANTE, Maxwell Machado; SILVA, Rogério Oliveira da. **Visão geral sobre microcontroladores e prototipagem com Arduino**. Revista Tecnologias em Projeção, Brasília, v. 10, n. 1, p. 36-46, 2019. Disponível em: http://revista.faculdadeprojecao.edu.br/index.php/Projecao4/article/view/1357. Acesso em: 06 nov. 2020.

ARDUINO. **Arduino Software (IDE)**. 2015. Disponível em: https://www.arduino.cc/en/guide/Environment#toc4. Acesso em: 09 nov. 2020

ASHTON, Kevin. **Kevin Ashton: entrevista exclusiva com o criador do termo "Internet das Coisas"**. Financiadora de Inovação e Pesquisa (FINEP), entrevista publicada em: 13 jan. 2015. Disponível em: http://finep.gov.br/noticias/todas-noticias/4446-kevin-ashton-entrevista-exclusiva-com-o-criador-do-termo-internet-das-coisas. Acesso em: 05 set. 2020.

BERNARDES, Rodrigo Camargos. **Tecnologias para apoiar o desenvolvimento de aplicações de internet das coisas**. 2018. 42 f. Monografia (Especialização) - Curso de Curso de Especialização em Internet das Coisas, Departamento Acadêmico de Eletrônica, Universidade Tecnológica Federal do Paraná, Curitiba, 2018. Disponível em: http://repositorio.roca.utfpr.edu.br/jspui/bitstream/1/13881/1/CT_CEIOT_I_2018_12.pdf. Acesso em: 05 set. 2020.

BISSI, Thelry David. **Reconhecimento Facial com os algoritmos Eigenfaces e Fisherfaces**. 2018. 40 f. TCC (Graduação) - Curso de Ciência da Computação, Faculdade de Computação,

Universidade Federal de Uberlândia, Uberlândia, 2018. Disponível em: http://repositorio.ufu.br/handle/123456789/22158. Acesso em: 26 out. 2020.

COSTA, Vambaster José da. **Reconhecimento de padrões faciais: uma síntese**. 2019. 97 f. TCC (Graduação) - Curso de Ciência da Computação, Coordenação de Ciência da Computação, Universidade Tecnológica Federal do Paraná, Santa Helena, 2019. Disponível em: http://repositorio.roca.utfpr.edu.br/jspui/handle/1/11953. Acesso em: 26 out. 2020.

DOMPIERI, Márcia Helena Galina; SILVA, Marcos Aurélio Santos da; JÚNIOR, Lauro Rodrigues Nogueira. **Sistemas de referência terrestre e posicionamento por satélite**. 2015. 35 f. Embrapa, Embrapa Tabuleiros Costeiros, Aracaju, 2015. Disponível em: https://www.infoteca.cnptia.embrapa.br/infoteca/bitstream/doc/1042182/1/Doc197.pdf. Acesso em: 07 out. 2020.

ESPRESSIF. **ESP32 Series**: *Datasheet*. 2020. Disponível em: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf. Acesso em: 06 nov. 2020.

FAGGIAN, Hugo César. **Geometria e GPS**. 2019. 60 f. Dissertação (Mestrado) - Curso de Matemática em Rede Nacional, Instituto de Ciências Matemáticas e de Computação ICMC-USP, Universidade de São Paulo (USP), São Carlos, 2019. Disponível em: https://www.teses.usp.br/teses/disponiveis/55/55136/tde-17092019150542/publico/HugoCesarFraggian_revisada.pdf. Acesso em: 26 set. 2020.

FÉRRER, Rômulo. **A impressão digital e sistemas biométricos**. 2008. Disponível em: https://sites.google.com/site/romuloferrer/projects-1/digital-image-processing/a-impress%C3%A3o-digital-e-sistemas-biom%C3%A9tricos. Acesso em: 21 out. 2020.

FIREBASE. ***Firebase Realtime Database***. 2020a. Disponível em: https://firebase.google.com/docs/database?hl=pt-br. Acesso em: 09 nov. 2020.

FIREBASE. **Adicionar o Firebase ao projeto para Android**. 2020b. Disponível em: https://firebase.google.com/docs/android/setup?hl=pt-br. Acesso em: 09 nov. 2020.

GUIMARÃES, Rafael Miranda. **Desenvolvimento de um protótipo de software de reconhecimento facial de tempo real para registro eletrônico de ponto em ambientes *indoor* com utilização do dispositivo Kinect**. 2015. 196 f. Dissertação (Mestrado) - Curso de Sistemas de Informação e Gestão do Conhecimento, Faculdade de Ciências Empresariais - FACE, Universidade FUMEC, Belo Horizonte, 2015. Disponível em: http://www.fumec.br/revistas/sigc/article/view/3037. Acesso em: 26 out. 2020.

IBM. **NoSQL *Databases***. 2020. Disponível em: https://www.ibm.com /cloud/learn/nosql-databases#:~:text=NoSQL%2C%20which%20stands%20for%20%E2%80%9Cnot,structures%20found%20in%20relational%20databases. Acesso em: 09 nov. 2020.

IBM KNOWLEDGE CENTER. ***Geographic coordinate systems***. 2018. Disponível em: https://www.ibm.com/support/knowledgecenter/SSEPGG_11.5.0/com.ibm.db2.luw.spatial.topics.doc/doc/csb3022a.html. Acesso em 26 set. 2020.

LOUREIRO, Gabriel Sarmento. **Verificação automática georreferenciada**. 2019. 79 f. TCC (Graduação) - Curso de Engenharia de Computação, Escola de Ciências Exatas e da Computação, Pontifícia Universidade Católica de Goiás, Goiânia, 2019.

MACHADO, Everto Fábio da Silva. **Desenvolvimento de sistemas de geolocalização e rastreamento para a plataforma Android – COMPASS**. 2015. 55 f. Monografia (Especialização) - Curso de Desenvolvimento de Sistemas para a Internet e Dispositivos Móveis, Coordenação de Licenciatura em Informática, Universidade Tecnológica Federal do Paraná, Francisco Beltrão, 2015. Disponível em: http://repositorio.roca.utfpr.edu.br/jspui/bitstream/1/6914/1/FB_DESIDM_I_2014_01.pdf. Acesso em: 22 set. 2020.

MAGRANI, Eduardo. **A Internet das coisas**. Rio de Janeiro: Fgv Editora, p. 46, 2018. 192 f. Disponível em: http://bibliotecadigital.fgv.br/dspace/bitstream/handle/10438/23898/A%20internet%20das%20coisas.pdf?sequence=1&isAllowed=y. Acesso em: 31 ago. 2020.

NETO, Esequias Aquino Duarte; JOB, Ricardo de Sousa; VENCESLAU, Amanda Drielly Pires; SILVA, Samuel Alves da; LIRA, Valnyr Vasconcelos. **Sistemas automáticos de impressões digitais integrando Java e Arduino**. Revista Principia: Divulgação Científica e Tecnológica do IFPB, João Pessoa, n. 24, p. 32-41, jun. 2014. Disponível em: periodicos.ifpb.edu.br/index.php/principia/article/download/101/116. Acesso em: 21 out. 2020.

ROCHA, Chirlando Weliton de Souza. **Detecção e localização de clonagem de endereço MAC em redes cabeadas: Uma proposta para o IFMG-SJE**. 2017. 96 f. Dissertação (Mestrado) - Curso de Ciência da Computação, Pós-Graduação em Ciência da Computação, Universidade Federal de Pernambuco, Recife, 2017. Disponível em: https://repositorio.ufpe.br/handle/123456789/26715. Acesso em: 15 out. 2020.

ROUSSEAU, Christiane; SAINT-AUBIN, Yvan. **Matemática e atualidade**. v. 1, tradução de Miguel V. S. Frasson, Rio de Janeiro: SBM, 2015.

RS ROBÓTICA. **ESP32 Modulo WIFI e *Bluetooth* ESP-WROOM-32**. 2020. Disponível em: https://www.rsrobotica.com.br/esp32-modulo-wifi-e-bluetooth-esp-wroom-32. Acesso em: 01 nov. 2020.

SANTOS, Pedro Miguel Pereira. **Internet das coisas: o desafio da privacidade.** Dissertação (mestrado em sistemas de informação organizacionais) - Escola Superior de Ciências Empresariais, Instituto Politécnico de Setúbal, 2016. Disponível em: https://comum.rcaap.pt/bitstream/10400.26/17545/1/Disserta%C3%A7%C3%A3o%20Pedro%20Santos%20140313004%20MSIO.pdf. Acesso em: 31 ago. 2020.

SILVA, Fellipe Eduardo Gonçalves da. **Sistema de identificação biométrica baseado em extração de minúcias e redes neurais artificiais**. 2019. 66 f. TCC (Graduação) - Curso de Engenharia de Controle e Automação, Departamento de Engenharia de Controle e Automação e Computação, Universidade Federal de Santa Catarina, Blumenau, 2019. Disponível em: https://repositorio.ufsc.br/bitstream/handle/123456789/197848/TCC_20191_Fellipe_Silva.pdf?sequence=1&isAllowed=y. Acesso em: 21 out. 2020.

STALLINGS, William; BROWN, Lawrie. **Segurança de Computadores: Princípios e Práticas**: Autenticação de usuários. 2. ed. Elsevier, 2014. 858 p. Tradução de *Computer Security*: *Principles and Practice*, 2nd. ed.

STATLER, Stephen. *Geofencing: Everything You Need to Know*. In: STATLER, Stephen. ***Beacon Technologies***. San Diego, Califórnia, Apress, 2016. p. 307-316.

TEIXEIRA, André Manuel Rodrigues. **Análise e utilização de protocolos de redes de sensores sem fios**. 2016. 76 f. Dissertação (Mestrado) - Curso de Sistemas de Informação, Escola Superior de Tecnologia e de Gestão, Instituto Politécnico de Bragança, Bragança, 2016. Disponível em: https://bibliotecadigital.ipb.pt/bitstream/10198/14027/1/Andr%C3%A9_Teixeira_MSI_2016.pdf. Acesso em: 05 set. 2020.

USINAINFO. **Programar ESP32 com a IDE Arduino**: Tutorial Completo. 2019. Disponível em: https://www.usinainfo.com.br/blog/programar-esp32-com-a-ide-arduino-tutorial-completo/. Acesso em: 09 nov. 2020.

VAL, Fernanda Baumgarten Ribeiro do; MARCELINO, Priscila Ribeiro; NETO, João José. **Uso de Técnicas Adaptativas no Reconhecimento Biométrico por Impressão Digital**. 2015. p. 80-85. Curso de Engenharia Elétrica Com Ênfase em Computação, Escola Politécnica da Universidade de São Paulo, Universidade de São Paulo, São Paulo, 2015. Disponível em: https://www.researchgate.net/profile/Claudia_Zapata2/publication/275032442_Construccion_de_un_Compilador_de_Asertos_de_Programacion_Metodica_Utilizando_El_Metodo_De_Automatas_Adaptativos/links/552fd54f0cf20ea0a06ecdda/Construccion-de-un-Compilador-de-Asertos-de-Programacion-Metodica-Utilizando-El-Metodo-De-Automatas-Adaptativos.pdf#page=92. Acesso em: 21 out. 2020

WHITE, Sarah K. ***What is geofencing? Putting location to work***. CIO Magazine, IDG Communications, Inc. 01 nov. 2017. Disponível em: https://www.cio.com/article/2383123/geofencing-explained.html. Acesso em: 07 out. 2020.

ZANCHI, Marta Gaia. LitePoint Corporation. ***Bluetooth Low Energy***. 2016. Disponível em: https://www.litepoint.com/wp-content/uploads/2018/12/Bluetooth-Low-Energy-White-Paper-102416.pdf. Acesso em: 22 set. 2020.

ZIN, Mohd Shahril Izuan Mohd; NURJI, M. F. M.; ISA, Azmi Awang Md; ISA, Mohd Sa'Ari Mohamad. ***Geofencing -based Auto-Silent Mode Application***. 2016. p. 199-204, 8 v. Tese (Doutorado) - *Faculty Of Electronic And Computer Engineering* (FKEKK), *Centre Of Telecommunication Research And Innovation* (CETRI), *Universiti Teknikal Malaysia Melaka* (UTEM), Malásia, 2016. Disponível em: https://journal.utem.edu.my/index.php/jtec/article/view/1398/880. Acesso em: 07 out. 2020.

6 APÊNDICES

Apêndice A

Implementação em C do código presente no ESP32

```
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>
#include <BLEAddress.h>
#include "esp_bt_defs.h"
#include "esp_gatts_api.h"

#define LED_BRANCO_FRIO 13
#define LED_AZUL 26
#define LED_BRANCO_QUENTE 14

#define MAC_ADDRESS "0C:9D:92:7C:88:48"

#define SERVICE_UUID           "6E400001-B5A3-F393-E0A9-E50E24DCCA9E"
#define CHARACTERISTIC_UUID_RX "6E400002-B5A3-F393-E0A9-E50E24DCCA9E"
#define CHARACTERISTIC_UUID_TX "6E400003-B5A3-F393-E0A9-E50E24DCCA9E"

BLECharacteristic *pCharacteristic;
bool deviceConnected = false;
uint8_t receivedMac[17];
char state = '0';
String converted;

class ServerCallbacks: public BLEServerCallbacks {
   void onConnect(BLEServer* pServer, esp_ble_gatts_cb_param_t *param) {
     gatts_connect_evt_param* p = param->connect;
     esp_bd_addr_t* mac = param->connect;
     converted = (char*)mac;

     Serial.print("Dispositivo conectado!");
     digitalWrite(LED_BRANCO_QUENTE, HIGH);
     if (converted == MAC_ADDRESS) {
       deviceConnected = true;
       state = '1';
     }
     else {
       pServer->disconnect(p->conn_id);

     }
   };
   void onDisconnect(BLEServer* server) {
```

```
    deviceConnected = false;
    digitalWrite(LED_BRANCO_QUENTE, LOW);
  }
};

class MyCallbacks: public BLECharacteristicCallbacks {
  void onWrite(BLECharacteristic *pCharacteristic) {
    std::string rxValue = pCharacteristic->getValue();
    if (rxValue.length() > 0) {
      Serial.print("Valor recebido: ");
      for (int i = 0; i < rxValue.length(); i++) {
        Serial.print(rxValue[i]);
      }
      if (state == '1') {
        digitalWrite(LED_AZUL, HIGH);
        Serial.print("MAC validado!");
        characteristic->setValue(state);
        characteristic->notify();
        delay(20);
        if (caracteristic->getValue() == '1') {
          digitalWrite(LED_BRANCO_FRIO, HIGH);
        }
        else {
          Serial.print("Autenticacao falhou! Desconectando...");
          deviceConnected = false;
          digitalWrite(LED_BRANCO_QUENTE, LOW);
        }
      }
    }
    else {
      Serial.print("Endereco MAC nao cadastrado! Acesso negado. Desconectando...");
      deviceConnected = false;
      digitalWrite(LED_BRANCO_QUENTE, LOW);
    }
    digitalWrite(LED_BRANCO_QUENTE, LOW);
    digitalWrite(LED_AZUL, LOW);
    digitalWrite(LED_BRANCO_FRIO, LOW);
  }
};

void setup() {
  Serial.begin(115200);
  pinMode(LED_BRANCO_FRIO, OUTPUT);
  pinMode(LED_AZUL, OUTPUT);
  pinMode(LED_BRANCO_QUENTE, OUTPUT);

  BLEDevice::init("ESP32")
  BLEServer *pServer = BLEDevice::createServer();
  pServer->setCallbacks(new ServerCallbacks());
```

```
  BLEService *pService = pServer->createService(SERVICE_UUID);

  pCharacteristic = pService->createCharacteristic(
                      CHARACTERISTIC_UUID_TX,
                      BLECharacteristic::PROPERTY_NOTIFY
                    );

  pCharacteristic->addDescriptor(new BLE2902());

  BLECharacteristic* pCharacteristic = pService->createCharacteristic(
                                 CHARACTERISTIC_UUID_RX,
                                 BLECharacteristic::PROPERTY_WRITE
                               );

  pCharacteristic->setCallbacks(new MyCallbacks());
  pService->start();
  pServer->getAdvertising()->start();

}

void loop() {}
```

Apêndice B

Implementação em Kotlin da *Login Fragment*

LoginFragment.kt

```
import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.appcompat.widget.AppCompatButton
import androidx.constraintlayout.widget.Group
import androidx.fragment.app.Fragment
import androidx.navigation.fragment.findNavController

class LoginFragment : Fragment() {

    private lateinit var loadingGroup: Group
    private lateinit var appCompatButton: AppCompatButton

    override fun onCreateView(
            inflater: LayoutInflater,
            container: ViewGroup?,
            savedInstanceState: Bundle?,
```

```
    ): View? {
        return inflater.inflate(R.layout.fragment_login, container, false)
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        loadingGroup = view.findViewById(R.id.loadingGroup)
        appCompatButton = view.findViewById(R.id.btnLogin)
        appCompatButton.setOnClickListener { onResultSuccess() }
        onLoading(false)
    }

    private fun onLoading(loading: Boolean) {
        if(loading)
            loadingGroup.visibility = View.VISIBLE
        else
            loadingGroup.visibility = View.GONE
    }

    private fun onResultSuccess() {
        findNavController().navigate(R.id.action_loginFragment_to_mapsFragment)
    }

}
```

Apêndice C

Implementação em Kotlin da *Maps Fragment*

MapsFragment.kt

```
import android.Manifest
import android.annotation.SuppressLint
import android.app.Activity
import android.app.PendingIntent
import android.bluetooth.BluetoothAdapter
import android.bluetooth.BluetoothDevice
import android.bluetooth.BluetoothSocket
import android.content.*
import android.content.pm.PackageManager
import android.content.res.Resources
import android.graphics.Color
import android.location.Location
import android.os.Bundle
import android.os.Handler
import android.os.Message
import android.util.Log
import android.view.LayoutInflater
```

```
import android.view.MenuItem
import android.view.View
import android.view.ViewGroup
import android.widget.Toast
import androidx.appcompat.app.ActionBarDrawerToggle
import androidx.appcompat.app.AlertDialog
import androidx.appcompat.app.AppCompatActivity
import androidx.appcompat.widget.Toolbar
import androidx.core.app.ActivityCompat
import androidx.core.content.ContextCompat
import androidx.core.view.GravityCompat
import androidx.drawerlayout.widget.DrawerLayout
import androidx.fragment.app.Fragment
import androidx.navigation.fragment.findNavController
import com.google.android.gms.location.*
import com.google.android.gms.maps.CameraUpdateFactory
import com.google.android.gms.maps.GoogleMap
import com.google.android.gms.maps.OnMapReadyCallback
import com.google.android.gms.maps.SupportMapFragment
import com.google.android.gms.maps.model.CircleOptions
import com.google.android.gms.maps.model.LatLng
import com.google.android.gms.maps.model.MapStyleOptions
import com.google.android.gms.maps.model.MarkerOptions
import com.google.android.gms.tasks.OnFailureListener
import com.google.android.gms.tasks.OnSuccessListener
import com.google.android.material.navigation.NavigationView
import com.google.android.material.textfield.TextInputEditText
import com.google.firebase.database.*
import java.io.IOException
import java.io.InputStream
import java.io.OutputStream
import java.lang.StringBuilder
import java.util.*

class MapsFragment : Fragment(), OnMapReadyCallback,
NavigationView.OnNavigationItemSelectedListener {

    private lateinit var map: GoogleMap
    private val TAG = MapsFragment::class.java.simpleName
    private val REQUEST_LOCATION_PERMISSION = 1
    private lateinit var mapFragment: SupportMapFragment

    private lateinit var toolbar: Toolbar
    private lateinit var toggle: ActionBarDrawerToggle
    private lateinit var drawerLayout: DrawerLayout

    private val geofencingClient: GeofencingClient? = null
    private val geofenceHelper: GeofenceHelper? = null
    private var GEOFENCE_RADIUS = 100f
    private lateinit var GEOFENCE_ID: String
```

```
    private val FINE_LOCATION_ACCESS_REQUEST_CODE = 10001
    private val BACKGROUND_LOCATION_ACCESS_REQUEST_CODE = 10002

    private lateinit var navigationView: NavigationView
    private lateinit var menuItem: MenuItem

    private lateinit var textInputEditText: TextInputEditText

    private var pendingIntent: PendingIntent? = null

    private lateinit var databaseReference: DatabaseReference

    private var bluetoothAdapter: BluetoothAdapter? = null
    private var bluetoothDevice: BluetoothDevice? = null
    private var bluetoothSocket: BluetoothSocket? = null
    private val REQUEST_ENABLE_BLUETOOTH = 1
    private var ANDROID_MAC_ADDRESS:String = "0C:9D:92:7C:88:48"
    private var ESP32_MAC_ADDRESS:String = "24:0A:C4:5B:7F:EA"
    private var connection: Boolean = false
    private var connectedThread: ConnectedThread? = null
    private lateinit var handler: Handler
    private var MESSAGE_READ = 3
    private lateinit var stringBuilder: StringBuilder
    private var uuid:UUID = UUID.fromString("00001101-0000-1000-8000-00805f9b34fb")
    private var counter = 1
    private lateinit var fusedLocationClient: FusedLocationProviderClient

    companion object {
        val EXTRA_ADDRESS: String = "Device_Address"
    }

    override fun onCreateView(
            inflater: LayoutInflater,
            container: ViewGroup?,
            savedInstanceState: Bundle?,
    ): View? {
        return inflater.inflate(R.layout.fragment_maps, container, false)
    }

    @SuppressLint("InflateParams")
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        mapFragment = childFragmentManager.findFragmentById(R.id.map) as
SupportMapFragment
        mapFragment.getMapAsync(this)
        fusedLocationClient =
LocationServices.getFusedLocationProviderClient(requireContext())
```

```
    toolbar = view.findViewById(R.id.materialToolbar)
    if(activity is AppCompatActivity){
        (activity as AppCompatActivity).setSupportActionBar(toolbar)
    }

    drawerLayout = view.findViewById(R.id.drawerLayout)
    navigationView = view.findViewById(R.id.lateralMenu)
    navigationView.setNavigationItemSelectedListener(this)

    toggle = ActionBarDrawerToggle(
            requireActivity(),
            drawerLayout,
            toolbar,
            R.string.navigation_drawer_open,
            R.string.navigation_drawer_close
    )
    drawerLayout.addDrawerListener(toggle)
    toggle.syncState()

    databaseReference = FirebaseDatabase.getInstance().reference

    getFirebaseData()

    bluetoothAdapter = BluetoothAdapter.getDefaultAdapter()

}

override fun onMapReady(googleMap: GoogleMap) {
    map = googleMap
    val latitude = -16.374152
    val longitude = -48.955747
    val home = LatLng(latitude, longitude)
    val zoomLevel = 20f
    /* 1: World
    5: Landmass/continent
    10: City
    15: Streets
    20: Buildings */
    map.addMarker(MarkerOptions().position(home).title("Casa"))
    map.moveCamera(CameraUpdateFactory.newLatLngZoom(home, zoomLevel))
    setMapStyle(map)
    enableMyLocation()
    map.uiSettings.isZoomControlsEnabled = true
    map.uiSettings.isCompassEnabled = true
    map.uiSettings.isZoomGesturesEnabled = true
    map.uiSettings.isScrollGesturesEnabled = true
    map.uiSettings.isTiltGesturesEnabled = true
    map.uiSettings.isRotateGesturesEnabled = true
}
```

```
@SuppressLint("CutPasteId", "MissingPermission", "InflateParams")
override fun onNavigationItemSelected(menuItem: MenuItem) : Boolean {
   when (menuItem.itemId) {
      R.id.item1Casa -> {
         val dialogView = layoutInflater.inflate(R.layout.set_id_geofence, null)
         textInputEditText = dialogView.findViewById(R.id.setGeoId)
         val alertDialog = AlertDialog.Builder(requireContext())
         alertDialog.setView(dialogView)
            .setCancelable(false)
            .setPositiveButton("OK", { dialogInterface: DialogInterface, i: Int -> })
            .show()

         GEOFENCE_ID = textInputEditText.layoutParams.toString()
         setMapLongClick(map, 40f, GEOFENCE_ID) // 12f 450 m^2 35f
      }
      R.id.item2Trabalho -> {
         val alertDialog = AlertDialog.Builder(requireContext())
         val dialogView = layoutInflater.inflate(R.layout.set_id_geofence, null)
         textInputEditText = dialogView.findViewById(R.id.setGeoId)
         alertDialog.setView(dialogView)
            .setCancelable(false)
            .setPositiveButton("OK", { dialogInterface: DialogInterface, i: Int -> })
            .show()

         GEOFENCE_ID = textInputEditText.text.toString()
         setMapLongClick(map, 40f, GEOFENCE_ID)  // 12f 450 m^2 35f
      }
      R.id.item3Carro -> {
         databaseReference.child("Geofences").child("Geofence Car").removeValue()
         val alertDialog = AlertDialog.Builder(requireContext())
         val dialogView = layoutInflater.inflate(R.layout.set_id_geofence, null)
         textInputEditText = dialogView.findViewById(R.id.setGeoId)
         alertDialog.setView(dialogView)
            .setCancelable(false)
            .setPositiveButton("OK", { dialogInterface: DialogInterface, i: Int -> })
            .show()
         GEOFENCE_ID = textInputEditText.text.toString()

         if (bluetoothAdapter == null) {
            Toast.makeText(
               requireContext(),
               "Este dispositivo não possui suporte a Bluetooth.",
               Toast.LENGTH_SHORT
            ).show()
         }

         if (!bluetoothAdapter!!.isEnabled) {
            val enableBluetoothIntent =
Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE)
            startActivityForResult(enableBluetoothIntent,
```

```
REQUEST_ENABLE_BLUETOOTH)
            }

            bluetoothDevice =
bluetoothAdapter!!.getRemoteDevice(ESP32_MAC_ADDRESS)
            try{
               bluetoothSocket =
bluetoothDevice?.createRfcommSocketToServiceRecord(uuid)
               bluetoothSocket?.connect()

               connection = true

               connectedThread = ConnectedThread(bluetoothSocket!!)
               connectedThread!!.start()

               Toast.makeText(
                  requireContext(),
                  "Dispositivo conectado com o ESP32.",
                  Toast.LENGTH_SHORT
               ).show()

            }
            catch (e: IOException) {
               connection = false
               Log.e(TAG, "Ocorreu um erro: ", e)
            }

            connectedThread?.write(ANDROID_MAC_ADDRESS)

            handler = @SuppressLint("HandlerLeak")
            object : Handler() {
               override fun handleMessage(msg: Message) {
                  super.handleMessage(msg)

                  if (msg.what == MESSAGE_READ) {
                     val received = msg.obj
                     stringBuilder.append(received)

                     if(stringBuilder[0] == '1'){
                        connectedThread?.write("1")
                        fusedLocationClient.lastLocation.addOnSuccessListener { location:
Location? ->
                           val latLng = LatLng(location!!.latitude, location.longitude)
                           val radius = 20f

                           addGeofence(GEOFENCE_ID, latLng, radius)

                           val snippet = String.format(
                              Locale.getDefault(),
                              "Lat: %1$.5f, Long: %2$.5f",
```

```
                        latLng.latitude,
                        latLng.longitude
                    )
                    map.addMarker(
                        MarkerOptions()
                            .position(latLng)
                            .title(GEOFENCE_ID)
                            .snippet(snippet)
                    )

                    val locationData2 = LocationData(GEOFENCE_ID, latLng.latitude,
latLng.longitude, radius)
                    databaseReference.child("Geofences").child("Geofence
Car").setValue(locationData2)
                    Toast.makeText(requireContext(), "Geofence adicionada!\n   Salva no
Firebase.", Toast.LENGTH_SHORT).show()

                        }
                    }
                    stringBuilder.delete(0,0)
                }
            }
findNavController().navigate(R.id.action_mapsFragment_to_fingerPrintFragment)
            }
        else {
            Toast.makeText(requireContext(), "Erro ao realizar operação com Bluetooth.",
Toast.LENGTH_SHORT).show()
        }
        }
        R.id.item4PortaCarroDigital -> {
            findNavController().navigate(R.id.action_mapsFragment_to_fingerPrintFragment)
        }
        R.id.item5PortaCarroFacial -> {

findNavController().navigate(R.id.action_mapsFragment_to_facialRecognitionFragment)

        }
        R.id.item6Todas -> {
            removeAllGeofences()
            databaseReference.child("Geofences").removeValue()
            counter = 1
        }
        R.id.item7Logout -> {
            findNavController().navigate(R.id.action_mapsFragment_to_loginFragment)
        }
    }
    drawerLayout.closeDrawer(GravityCompat.START)
    return true
}
```

```
    private fun setMapLongClick(map: GoogleMap, radius: Float, ID: String) {
        map.setOnMapLongClickListener { latLng ->
            val snippet = String.format(
                    Locale.getDefault(),
                    "Lat: %1$.5f, Long: %2$.5f",
                    latLng.latitude,
                    latLng.longitude
            )
            map.addMarker(
                    MarkerOptions()
                            .position(latLng)
                            .title(ID)
                            .snippet(snippet)
            )
            addGeofence(ID, latLng, radius)
            val locationData = LocationData(ID, latLng.latitude, latLng.longitude, radius)

databaseReference.child("Geofences").child("Geofence$counter").setValue(locationData)
            Toast.makeText(requireContext(), "Geofence adicionada!\n   Salva no Firebase.",
Toast.LENGTH_SHORT).show()
            counter++
        }
    }

    private fun addGeofence(id: String, latLng: LatLng, radius: Float) {
        val geofence: Geofence? = geofenceHelper?.getGeofence(
                id,
                latLng,
                radius,
                Geofence.GEOFENCE_TRANSITION_ENTER or
Geofence.GEOFENCE_TRANSITION_DWELL or
Geofence.GEOFENCE_TRANSITION_EXIT
        )
        val geofencingRequest: GeofencingRequest? =
geofenceHelper?.getGeofencingRequest(geofence)
        pendingIntent = geofenceHelper?.getPendingIntentHelper()
        if (ActivityCompat.checkSelfPermission(
                        requireContext(),
                        Manifest.permission.ACCESS_FINE_LOCATION
                ) != PackageManager.PERMISSION_GRANTED
        ) {
            return
        }
        geofencingClient?.addGeofences(geofencingRequest!!,
pendingIntent!!)?.addOnSuccessListener(OnSuccessListener<Void?> {
            Log.i(TAG, "Ok!")

        })?.addOnFailureListener(OnFailureListener { e ->
            val errorMessage: String? = geofenceHelper?.getErrorString(e)
            Log.i(TAG, "Erro: $errorMessage")
```

```
    })

    map.addCircle(
        CircleOptions()
            .center(latLng)
            .radius(radius.toDouble())
            .strokeColor(Color.argb(255, 138, 43, 226))
            .fillColor(Color.argb(64, 138, 43, 226))
            .strokeWidth(4f)
    )
}

private fun removeAllGeofences(){
    geofencingClient?.removeGeofences(pendingIntent!!)
    map.clear()
    Toast.makeText(requireContext(),
            "Todas Geofences Removidas!",
            Toast.LENGTH_SHORT).show()
}

private fun getFirebaseData() {
    databaseReference.child("Geofences").addListenerForSingleValueEvent(
        object : ValueEventListener {
            override fun onDataChange(dataSnapshot: DataSnapshot) {
                if(dataSnapshot.value != null)
                getAllData(dataSnapshot.value as HashMap<String, Any>)
            }
            override fun onCancelled(databaseError: DatabaseError) {
                Log.w(TAG, "loadPost:onCancelled", databaseError.toException())
            }
        })
}

private fun getAllData(locations: HashMap<String, Any>) {
    for ((key,value) in locations) {
        val ID = value as HashMap<*, *>
        val singleLocation = value as HashMap<*, *>
        val latLng = LatLng(
            (singleLocation["latitude"] as Double?)!!,
            (singleLocation["longitude"] as Double?)!!
        )
       val radius = 40f

        addGeofence(ID.toString(), latLng, radius)

        val snippet = String.format(
            Locale.getDefault(),
            "Lat: %1$.5f, Long: %2$.5f",
            latLng.latitude,
            latLng.longitude
```

```
            )
            map.addMarker(
                MarkerOptions()
                    .position(latLng)
                    .title(ID.toString())
                    .snippet(snippet)
            )
        }
    }

    private fun setMapStyle(map: GoogleMap) {
        try {
            val success = map.setMapStyle(
                    MapStyleOptions.loadRawResourceStyle(
                            requireContext(),
                            R.raw.map_style
                    )
            )
            if (!success) {
                Log.e(TAG, "Aplicação do estilo falhou.")
            }
        } catch (e: Resources.NotFoundException) {
            Log.e(TAG, "Estilo não encontrado. Erro: ", e)
        }
    }

    private fun isPermissionGranted(): Boolean {
        return ContextCompat.checkSelfPermission(
                requireContext(),
                Manifest.permission.ACCESS_FINE_LOCATION
        ) == PackageManager.PERMISSION_GRANTED
    }

    private fun enableMyLocation() {
        if (isPermissionGranted()) {
            if (ActivityCompat.checkSelfPermission(
                            requireContext(),
                            Manifest.permission.ACCESS_FINE_LOCATION
                    ) != PackageManager.PERMISSION_GRANTED &&
ActivityCompat.checkSelfPermission(
                            requireContext(),
                            Manifest.permission.ACCESS_COARSE_LOCATION
                    ) != PackageManager.PERMISSION_GRANTED
            ) {
                return
            }
            map.isMyLocationEnabled = true
        } else {
            ActivityCompat.requestPermissions(
                    requireActivity(),
```

```
                arrayOf<String>(Manifest.permission.ACCESS_FINE_LOCATION),
                REQUEST_LOCATION_PERMISSION
            )
        }
    }

    override fun onRequestPermissionsResult(
        requestCode: Int,
        permissions: Array<String>,
        grantResults: IntArray
    ){
        if (requestCode == REQUEST_LOCATION_PERMISSION) {
            if (grantResults.contains(PackageManager.PERMISSION_GRANTED)) {
                enableMyLocation()
            }
        }
    }

    override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
        super.onActivityResult(requestCode, resultCode, data)
        if(requestCode == REQUEST_ENABLE_BLUETOOTH) {
            if(resultCode == Activity.RESULT_OK) {
                if (bluetoothAdapter!!.isEnabled) {
                    Toast.makeText(requireContext(),
                        "Bluetooth está ativado.",
                        Toast.LENGTH_SHORT)
                        .show()
                } else {
                    Toast.makeText(
                        requireContext(),
                        "Bluetooth está desativado.",
                        Toast.LENGTH_SHORT
                    ).show()
                }
            }else if (resultCode == Activity.RESULT_CANCELED) {
                Toast.makeText(
                    requireContext(),
                    "Ativação do Bluetooth foi cancelada.",
                    Toast.LENGTH_SHORT
                ).show()
            }
        }
    }

    private inner class ConnectedThread(private val bluetoothSocket: BluetoothSocket) :
Thread() {

        private val inStream: InputStream = bluetoothSocket.inputStream
        private val outStream: OutputStream = bluetoothSocket.outputStream
```

```
        override fun run() {
            val buffer = ByteArray(1024)
            var numBytes: Int

            while (true) {
                // Read from the InputStream.
                try {
                    numBytes = inStream.read(buffer)
                    val bluetoothData = String(buffer, 0, numBytes)
                    val readMsg = handler.obtainMessage(
                        MESSAGE_READ, numBytes, -1,
                        bluetoothData)
                    readMsg.sendToTarget()
                } catch (e: IOException) {
                    Log.d(TAG, "Input stream was disconnected", e)
                    break
                }
            }
        }

        fun write(sendDataToEsp32: String) {
            val msgBuffer: ByteArray = sendDataToEsp32.toByteArray()
            try {
                outStream.write(msgBuffer)
            } catch (e: IOException) {
                Log.e(TAG, "Algum erro ocorreu ao enviar o arquivo.", e)
            }
        }

    }

}
```

GeofenceHelper.kt

```
import android.app.PendingIntent
import android.content.Context
import android.content.ContextWrapper
import android.content.Intent
import android.widget.Toast
import com.google.android.gms.common.api.ApiException
import com.google.android.gms.location.Geofence
import com.google.android.gms.location.GeofenceStatusCodes
import com.google.android.gms.location.GeofencingRequest
import com.google.android.gms.maps.model.LatLng
import com.google.firebase.FirebaseApp
import com.google.firebase.database.DatabaseReference
import com.google.firebase.database.FirebaseDatabase

class GeofenceHelper(base: Context?) : ContextWrapper(base) {
```

```
    private lateinit var pendingIntent: PendingIntent
    private lateinit var databaseReference: DatabaseReference

    fun getGeofencingRequest(geofence: Geofence?): GeofencingRequest {
        return GeofencingRequest.Builder()
            .addGeofence(geofence!!)
            .setInitialTrigger(GeofencingRequest.INITIAL_TRIGGER_ENTER)
            .build()
    }

    fun getGeofence(ID: String, latLng: LatLng, radius: Float, transitionTypes: Int): Geofence
{
        return Geofence.Builder()
            .setCircularRegion(latLng.latitude, latLng.longitude, radius)
            .setRequestId(ID)
            .setTransitionTypes(transitionTypes)
            .setLoiteringDelay(2000)
            .setExpirationDuration(Geofence.NEVER_EXPIRE)
            .build()
    }

    fun getPendingIntentHelper(): PendingIntent {
        if (pendingIntent != null) {
            return pendingIntent
        }
        val intent = Intent(this, GeofenceBroadcastReceiver::class.java)
        pendingIntent =
            PendingIntent.getBroadcast(this, 2607, intent,
PendingIntent.FLAG_UPDATE_CURRENT)
        return pendingIntent
    }

    fun getErrorString(e: Exception): String {
        if (e is ApiException) {
            when (e.statusCode) {
                GeofenceStatusCodes.GEOFENCE_NOT_AVAILABLE -> return "Geofence
Indisponível"
                GeofenceStatusCodes.GEOFENCE_TOO_MANY_GEOFENCES -> return
"Número máximo de geofences excedido"
                GeofenceStatusCodes.GEOFENCE_TOO_MANY_PENDING_INTENTS -> return
"Número de pending intents excedido"
            }
        }
        return e.localizedMessage!!
    }

    companion object {
        private const val TAG = "GeofenceHelper"
```

```
    }
}
```

GeofenceBroadcastReceiver.kt

```
import android.app.Activity
import android.content.BroadcastReceiver
import android.content.ContentValues.TAG
import android.content.Context
import android.content.Intent
import android.util.Log
import android.widget.Toast
import androidx.core.content.ContentProviderCompat.requireContext
import com.google.android.gms.location.Geofence
import com.google.android.gms.location.GeofenceStatusCodes
import com.google.android.gms.location.GeofencingEvent

class GeofenceBroadcastReceiver : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) {

        val notificationHelper = NotificationHelper(context)

        val geofencingEvent = GeofencingEvent.fromIntent(intent)
        if (geofencingEvent.hasError()) {

            val errorMessage = GeofenceStatusCodes
                .getStatusCodeString(geofencingEvent.errorCode)
            Log.d(TAG, "Erro $errorMessage ao receber o evento de geofence.")
            return
        }
        val geofenceList = geofencingEvent.triggeringGeofences

        for (geofence in geofenceList) {
            Log.d(TAG, "onReceive: " + geofence.requestId)
        }
        //       Location location = geofencingEvent.getTriggeringLocation();

        when (geofencingEvent.geofenceTransition) {
            Geofence.GEOFENCE_TRANSITION_ENTER -> {
                Toast.makeText(context, "GEOFENCE_TRANSITION_ENTER",
Toast.LENGTH_SHORT).show()
                notificationHelper.sendHighPriorityNotification(
                    "GEOFENCE_TRANSITION_ENTER",
"GEOFENCE_TRANSITION_ENTER",
                    MapsFragment::class.java
                )
            }
            Geofence.GEOFENCE_TRANSITION_DWELL -> {
                Toast.makeText(context, "GEOFENCE_TRANSITION_DWELL",
Toast.LENGTH_SHORT).show()
```

```
            notificationHelper.sendHighPriorityNotification(
                "GEOFENCE_TRANSITION_DWELL", "",
                MapsFragment::class.java
            )
        }
        Geofence.GEOFENCE_TRANSITION_EXIT -> {
            Toast.makeText(context, "GEOFENCE_TRANSITION_EXIT",
Toast.LENGTH_SHORT).show()
            notificationHelper.sendHighPriorityNotification(
                "GEOFENCE_TRANSITION_EXIT", "",
                MapsFragment::class.java
            )
        }
    }
}

companion object {
    private const val TAG = "GeofenceBroadcastReceiver"
}
}
```

LocationData.kt

```
package com.example.myapplication2

import com.google.android.gms.maps.model.LatLng

class LocationData{
    private var ID: String = ""
    private var latitude: Double = 0.0
    private var longitude: Double = 0.0
    private var radius: Float = 0.0f

    constructor()
    constructor(ID: String, latitude: Double, longitude: Double, radius: Float){
        this.ID = ID
        this.latitude = latitude
        this.longitude = longitude
        this.radius = radius
    }

    fun getId() : String {
        return ID
    }

    fun setId(ID: String) {
        this.ID = ID
    }

    fun getLatitude() : Double {
```

```
        return latitude
    }

    fun setLatitude(latitude: Double) {
        this.latitude = latitude
    }

    fun getLongitude() : Double {
        return longitude
    }

    fun setLongitude(longitude: Double) {
        this.longitude = longitude
    }

    fun getRadius() : Float {
        return radius
    }

    fun setRadius(radius: Float) {
        this.radius = radius
    }
}
```

Apêndice D

Implementação em Kotlin da *FingerPrint Fragment*

FingerPrintFragment.kt

```
import android.annotation.SuppressLint
import android.graphics.Color
import androidx.biometric.BiometricManager
import androidx.biometric.BiometricPrompt
import android.os.Build
import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.Toast
import androidx.annotation.RequiresApi
import androidx.appcompat.widget.AppCompatButton
import androidx.appcompat.widget.AppCompatTextView
import androidx.core.content.ContextCompat
import androidx.fragment.app.Fragment
import androidx.navigation.fragment.findNavController
```

```
import java.security.*
import java.util.concurrent.Executor

class FingerPrintFragment : Fragment() {

    private lateinit var executor: Executor
    private lateinit var biometricPrompt: BiometricPrompt
    private lateinit var promptInfo: BiometricPrompt.PromptInfo
    private lateinit var biometricManager: BiometricManager
    private lateinit var appCompatButton: AppCompatButton
    private lateinit var appCompatTextView: AppCompatTextView

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?,
    ): View? {
        return inflater.inflate(R.layout.fragment_finger_print, container, false)
    }

    @SuppressLint("SetTextI18n")
    @RequiresApi(api = Build.VERSION_CODES.R)
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        appCompatTextView = view.findViewById(R.id.subtitleFingerPrint)
        appCompatButton = view.findViewById(R.id.fingerButton)

        biometricManager = BiometricManager.from(requireContext())
        when (biometricManager.canAuthenticate()) {
            BiometricManager.BIOMETRIC_SUCCESS -> {
                appCompatTextView.text = "Clique no botão para se identificar"
                appCompatTextView.setTextColor(Color.parseColor("#FFFFFFFF"))
            }
            BiometricManager.BIOMETRIC_ERROR_NO_HARDWARE -> {
                appCompatTextView.text = "O dispositivo não possui leitor de impressão digital."
                appCompatTextView.setTextColor(Color.parseColor("#E60000"))
                appCompatButton.visibility = View.GONE
            }
            BiometricManager.BIOMETRIC_ERROR_HW_UNAVAILABLE -> {
                appCompatTextView.text = "Os sensores estão atualmente indisponíveis."
                appCompatTextView.setTextColor(Color.parseColor("#E60000"))
                appCompatButton.visibility = View.GONE
            }
            BiometricManager.BIOMETRIC_ERROR_NONE_ENROLLED -> {
                appCompatTextView.text =
                    "Impressões digitais não cadastradas! \nPor favor as cadastre nas configurações
de segurança."
                appCompatTextView.setTextColor(Color.parseColor("#E60000"))
                appCompatButton.visibility = View.GONE
```

```
            }
            BiometricManager.BIOMETRIC_ERROR_SECURITY_UPDATE_REQUIRED -> {
                appCompatTextView.text =
                    "Vulnerabilidade de segurança encontrada! Por favor as cadastre uma senha para
o dispositivo."
                appCompatTextView.setTextColor(Color.parseColor("#E60000"))
                appCompatButton.visibility = View.GONE
            }
        }

        executor = ContextCompat.getMainExecutor(requireContext())
        biometricPrompt = BiometricPrompt(requireActivity(), executor, object:
BiometricPrompt.AuthenticationCallback() {

            override fun onAuthenticationError(errorCode: Int, errString: CharSequence) {
                super.onAuthenticationError(errorCode, errString)
                Toast.makeText(requireContext(),
                    "Erro de autenticação: $errString!", Toast.LENGTH_SHORT)
                    .show()
            }

            override fun onAuthenticationSucceeded(result:
BiometricPrompt.AuthenticationResult) {
                super.onAuthenticationSucceeded(result)
                Toast.makeText(
                    requireContext(),
                    "Autenticado com sucesso!",
                    Toast.LENGTH_SHORT
                ).show()
                onResultSuccess()
            }

            override fun onAuthenticationFailed() {
                super.onAuthenticationFailed()
                Toast.makeText(requireContext(), "Autenticação falhou!",
                    Toast.LENGTH_SHORT)
                    .show()
            }
        })

        promptInfo = BiometricPrompt.PromptInfo.Builder()
            .setTitle("Olá! Prometo não demorar :)")
            .setDescription("Vamos dar uma volta de carro?")
            .setNegativeButtonText("Cancelar")
            .build()

        appCompatButton.setOnClickListener { biometricPrompt.authenticate(promptInfo) }

    private fun onResultSuccess() {
        findNavController().navigate(R.id.action_fingerPrintFragment_to_mapsFragment)
```

```
    }
}
```

Apêndice E

Implementação em Kotlin da *FacialRecognition Fragment*

FacialRecognitionFragment.kt

```
import android.app.AlertDialog
import android.graphics.Bitmap
import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.Toast
import androidx.appcompat.widget.AppCompatButton
import androidx.fragment.app.Fragment
import androidx.navigation.fragment.findNavController
import com.google.firebase.ml.vision.FirebaseVision
import com.google.firebase.ml.vision.common.FirebaseVisionImage
import com.google.firebase.ml.vision.face.FirebaseVisionFace
import com.google.firebase.ml.vision.face.FirebaseVisionFaceDetectorOptions
import com.wonderkiln.camerakit.*
import dmax.dialog.SpotsDialog
import kotlinx.android.synthetic.main.fragment_facial_recognition.*

class FacialRecognitionFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?,
    ): View? {
        return inflater.inflate(R.layout.fragment_facial_recognition, container, false)
    }

    private lateinit var faceDetectButton: AppCompatButton
    private lateinit var graphicOverlay: GraphicOverlay
    private lateinit var cameraView: CameraView
    private lateinit var alertDialog: AlertDialog

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
```

```
        faceDetectButton = view.findViewById(R.id.detectFaceButton)
        graphicOverlay = view.findViewById(R.id.graphicOverlay)
        cameraView = view.findViewById(R.id.cameraView)

        alertDialog = SpotsDialog.Builder()
            .setContext(requireContext())
            .setMessage("Por favor aguarde, carregando...")
            .setCancelable(false)
            .build()

        faceDetectButton.setOnClickListener(View.OnClickListener {
            cameraView.start()
            cameraView.captureImage()
            graphicOverlay.clear()
        })

        cameraView.addCameraKitListener(object : CameraKitEventListener {
            override fun onEvent(cameraKitEvent: CameraKitEvent) {}
            override fun onError(cameraKitError: CameraKitError) {}
            override fun onImage(cameraKitImage: CameraKitImage) {
                alertDialog.show()
                var bitmap = cameraKitImage.bitmap
                bitmap = Bitmap.createScaledBitmap(
                    bitmap!!,
                    cameraView.width,
                    cameraView.height,
                    false
                )
                processFaceDetection(bitmap)
                cameraView.stop()

            }
            override fun onVideo(cameraKitVideo: CameraKitVideo) {}
        })

    }

    private fun processFaceDetection(bitmap: Bitmap) {
        val firebaseVisionImage = FirebaseVisionImage.fromBitmap(bitmap)
        val firebaseVisionFaceDetectorOptions =
FirebaseVisionFaceDetectorOptions.Builder().build()
        val firebaseVisionFaceDetector = FirebaseVision.getInstance()
            .getVisionFaceDetector(firebaseVisionFaceDetectorOptions)
        firebaseVisionFaceDetector.detectInImage(firebaseVisionImage)
            .addOnSuccessListener { firebaseVisionFaces -> getFaceResults(firebaseVisionFaces)
}
            .addOnFailureListener { e ->
                Toast.makeText(
                    requireContext(),
                    "Error: " + e.message,
```

```
                Toast.LENGTH_SHORT
            ).show()
        }
    }

    private fun getFaceResults(firebaseVisionFaces: List<FirebaseVisionFace>) {
        var counter = 0
        for (face in firebaseVisionFaces) {
            val rect = face.boundingBox
            val rectOverlay = RectOverlay(graphicOverlay, rect)
            graphicOverlay.add(rectOverlay)
            counter += 1
        }
        alertDialog.dismiss()
        onResultSuccess()
    }

    override fun onPause() {
        super.onPause()
        cameraView.stop()
    }

    override fun onResume() {
        super.onResume()
    }

    private fun onResultSuccess() {
        Toast.makeText(requireContext(),
            "Reconhecido com sucesso!",
            Toast.LENGTH_SHORT).show()

findNavController().navigate(R.id.action_facialRecognitionFragment_to_mapsFragment)
    }

}
```

GraphicOverlay.kt

```
import android.content.Context
import android.graphics.Canvas
import android.util.AttributeSet
import android.view.View
import com.example.myapplication2.GraphicOverlay.Graphic
import com.google.android.gms.vision.CameraSource
import java.util.*

class GraphicOverlay(context: Context?, attrs: AttributeSet?) :
    View(context, attrs) {
    private val mLock = Any()
```

```
private var mPreviewWidth = 0
private var mWidthScaleFactor = 1.0f
private var mPreviewHeight = 0
private var mHeightScaleFactor = 1.0f
private var mFacing = CameraSource.CAMERA_FACING_BACK
private val mGraphics: MutableSet<Graphic> = HashSet()

    abstract fun draw(canvas: Canvas?)

    fun scaleX(horizontal: Float): Float {
        return horizontal * mOverlay.mWidthScaleFactor
    }

    fun scaleY(vertical: Float): Float {
        return vertical * mOverlay.mHeightScaleFactor
    }

    fun translateX(x: Float): Float {
        return if (mOverlay.mFacing == CameraSource.CAMERA_FACING_FRONT) {
            mOverlay.width - scaleX(x)
        } else {
            scaleX(x)
        }
    }

    fun translateY(y: Float): Float {
        return scaleY(y)
    }

    fun postInvalidate() {
        mOverlay.postInvalidate()
    }
}

fun clear() {
    synchronized(mLock) { mGraphics.clear() }
    postInvalidate()
}

fun add(graphic: Graphic) {
    synchronized(mLock) { mGraphics.add(graphic) }
    postInvalidate()
}

fun remove(graphic: Graphic) {
    synchronized(mLock) { mGraphics.remove(graphic) }
    postInvalidate()
}

fun setCameraInfo(previewWidth: Int, previewHeight: Int, facing: Int) {
```

```
        synchronized(mLock) {
            mPreviewWidth = previewWidth
            mPreviewHeight = previewHeight
            mFacing = facing
        }
        postInvalidate()
    }

    override fun onDraw(canvas: Canvas) {
        super.onDraw(canvas)
        synchronized(mLock) {
            if (mPreviewWidth != 0 && mPreviewHeight != 0) {
                mWidthScaleFactor = canvas.width.toFloat() / mPreviewWidth.toFloat()
                mHeightScaleFactor = canvas.height.toFloat() / mPreviewHeight.toFloat()
            }
            for (graphic in mGraphics) {
                graphic.draw(canvas)
            }
        }
    }

}
```

RectOverlay.kt

```
import android.graphics.*

class RectOverlay(graphicOverlay: GraphicOverlay, rect: Rect) :
    GraphicOverlay.Graphic(graphicOverlay) {
    private val rectColor = Color.GREEN
    private val strokeWidth = 4.0f
    private val rectPaint: Paint
    private val graphicOverlay: GraphicOverlay
    private val rect: Rect

    override fun draw(canvas: Canvas?) {
        val rectF = RectF(rect)
        rectF.left = translateX(rectF.left)
        rectF.right = translateX(rectF.right)
        rectF.top = translateX(rectF.top)
        rectF.bottom = translateX(rectF.bottom)
        canvas?.drawRect(rectF, rectPaint)
    }

    init {
        rectPaint = Paint()
        rectPaint.color = rectColor
        rectPaint.style = Paint.Style.STROKE
        rectPaint.strokeWidth = strokeWidth
```

```
        this.graphicOverlay = graphicOverlay
        this.rect = rect
    }

}
```

Apêndice F

Implementação em Kotlin da *Main Activity*

MainActivity.kt

```
import android.content.res.Configuration
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.util.Log
import androidx.navigation.NavController
import androidx.navigation.findNavController

class MainActivity : AppCompatActivity() {

    private val navController: NavController by lazy {
findNavController(R.id.navHostFragment) }
    private val COMMON_TAG: String = "OrientationChange"
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }

    override fun onSupportNavigateUp(): Boolean {
        return navController.navigateUp()
    }

    override fun onConfigurationChanged(newConfig: Configuration) {
        super.onConfigurationChanged(newConfig)
        if(newConfig.orientation == Configuration.ORIENTATION_LANDSCAPE){
            Log.i(COMMON_TAG, "Landscape")
        }
        else if(newConfig.orientation == Configuration.ORIENTATION_PORTRAIT){
            Log.i(COMMON_TAG, "Portrait")
        }
    }

}
```