**UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ**

**GIOVANNI FORASTIERI**

# SERVIÇO DE MENSAGERIA UTILIZANDO REDES TOLERANTES A ATRASO

**CURITIBA**

**2022**

**GIOVANNI FORASTIERI**

# SERVIÇO DE MENSAGERIA UTILIZANDO REDES TOLERANTES A ATRASO

## Messaging app using Delay Tolerant Networks

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia da Computação do Curso de Bacharelado em Engenharia da Computação da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Mauro Sergio Pereira Fonseca

**CURITIBA**

**2022**

**GIOVANNI FORASTIERI**

# SERVIÇO DE MENSAGERIA UTILIZANDO REDES TOLERANTES A ATRASO

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia da Computação do Curso de Bacharelado em Engenharia da Computação da Universidade Tecnológica Federal do Paraná.

Data de aprovação: 16/dezembro/2022

---

Mauro Sergio Pereira Fonseca
Doutorado
Universidade Tecnológica Federal do Paraná

---

Anelise Munaretto Fonseca
Doutorado
Universidade Tecnológica Federal do Paraná

---

Bogdan Tomoyuki Nassu
Doutorado
Universidade Tecnológica Federal do Paraná

---

Leyza Elmeri Baldo Dorini
Doutorado
Universidade Tecnológica Federal do Paraná

**CURITIBA**
**2022**

## AGRADECIMENTOS

Agradeço a Deus, nosso Senhor, por mais essa conquista e por se fazer presente em todos os momentos da minha vida.

Agradeço também à minha família por todo o carinho, apoio e suporte que me foi dado, mesmo em uma situação onde tivemos que ficar fisicamente distantes durante todo o decorrer deste curso.

Aos colegas e amigos que fiz durante o curso por toda ajuda, companheirismo e troca de conhecimentos. Sem vocês minha trajetória neste curso seria, sem dúvidas, mais difícil.

Ao corpo docente desta universidade pelo excelente serviço prestado a todos os alunos, incluindo a mim.

*“What we do now, echoes in eternity”*
– Marcus Aurelius

## RESUMO

O problema dos principais aplicativos mensageiros atuais é a sua dependência de conexão com a Internet. Esta, por sua vez, depende de infraestrutura, que pode, por exemplo, ser comprometida por interferência de desastres naturais, censurada ou vigiada pelos provedores de Internet (ISP), não alcançar locais remotos, além de vir com obrigações financeiras atreladas aos provedores de internet, fator limitante para pessoas de baixa renda. A proposta desse projeto é o desenvolvimento de um aplicativo mensageiro, para dispositivos *Android*, onde a comunicação é feita de forma criptografada, descentralizada, *peer-to-peer*, independente da infraestrutura de Internet, através da tecnologia *Wi-Fi Direct*. Cada *smartphone* age como um nó em uma DTN (rede tolerante a atraso), utilizando-se de outros nós suficientemente próximos fisicamente para os saltos, fazendo assim com que a infraestrutura de comunicação seja obtida unicamente a partir dos *smartphones*.



**Palavras-chave:** dtn (redes tolerantes a atraso); *wi-fi direct*; desenvolvimento android.

**ABSTRACT**

The problem with current messaging applications, is that it is dependant on Internet connection. This, in turn, is dependant on infrastructure, which can be, for example, compromised from interference of natural disasters, censored or surveilled by providers (ISP), be unable to reach remote locations, and comes with financial obligations, a limiting factor for low-income people. The proposal of this project is to develop a messaging application for Android devices, where the communication is done in an encrypted, decentralized, peer-to-peer and Internet-independent manner. This is done by using the Wi-Fi Direct technology, where every smartphone acts like a node in a DTN (delay tolerant network), using other physically close enough nodes for the hops, creating a communication infrastructure only from the smartphones themselves.

**Keywords:** dtn (delay tolerant networks); *wi-fi direct*; android development.

## LISTA DE FIGURAS

## LISTA DE TABELAS

## LISTAGEM DE CÓDIGOS FONTE

## LISTA DE ABREVIATURAS E SIGLAS

### Siglas

| | |
|---|---|
| ACK | Mensagem de reconhecimento, forma reduzida da palavra em inglês *acknowledgement* |
| AES | Um algoritmo de criptografia simétrica |
| API | Interface de programação de aplicação, do inglês *Application Programming Interface* |
| DTN | Redes tolerantes a atraso, do inglês *Delay Tolerant Networks* |
| GO | Proprietário de grupo, do inglês *Group Owner* |
| IDE | Ambiente de desenvolvimento integrado, do inglês *Integrated Development Environment* |
| P2P | Ponto-a-ponto, do inglês *peer-to-peer* |
| QR | Código de resposta rápida, do inglês *Quick Response Code* |
| RSA | Um algoritmo de criptografia assimétrica |
| TTL | Tempo de vida, do inglês *Time To Live* |
| UML | Linguagem de modelagem unificada, do inglês *Unified Modeling Language* |

**SUMÁRIO**

# 1 INTRODUÇÃO

Comunicação é algo essencial para os seres humanos, e estes às vezes se encontram em situações onde a sua habilidade de se comunicar à distância foi comprometida, como por exemplo em uma trilha, onde o sinal de Internet não chega, ou até mesmo situações mais críticas como em guerras, resgates, acidentes, eventual censura de governos, etc; onde pode existir falta de conectividade, porém ainda assim, existe a necessidade de comunicação.

Soluções para esses problemas já existem, como por exemplo o uso de rádios. Porém, para que a comunicação aconteça, é preciso que os integrantes possuam esses rádios previamente, e nem sempre indivíduos que se encontram nas situações descritas acima vão ter tempo e/ou oportunidade de obter esse *hardware* extra. Além disso, rádios convencionais não implementam criptografia, e toda comunicação decorre em frequências abertas, onde todos conseguiriam receber mensagens de todos. Mas existe um problema ainda mais fundamental: rádios não são exatamente intuitivos para muitas pessoas.

Por outro lado, aplicativos mensageiros como o *WhatsApp* são extremamente populares, a ponto de que 99% dos celulares no Brasil possuem o aplicativo instalado, e 93% utilizam o aplicativo todos os dias (VENTURA, 2020). Além disso, *smartphones* são dispositivos que se tornaram amplamente disponíveis para a maior parte da população, de forma que hoje há mais de um *smartphone* por brasileiro (LAGO, 2020).

Os três aplicativos mensageiros mais utilizados pelos brasileiros são o *WhatsApp*, o *Telegram* e o *Messenger* (GOOGLE, 2021). Todos esses possuem a capacidade de troca de mensagens criptografadas, funcionalidade essa que já se tornou perfeitamente intuitiva para os brasileiros. Porém, todos os três aplicativos citados acima dependem de infraestrutura de Internet, não sendo possível utilizá-los de forma P2P (Ponto-a-ponto, do inglês *peer-to-peer*) em caso de não ser possível conectar à Internet.

Uma alternativa ao uso da Internet seria a utilização de DTN (redes tolerantes a atraso, do inglês *Delay Tolerant Networks*), visto que essas não dependem necessariamente de infraestrutura, resolvendo o problema da comunicação onde o acesso a internet foi comprometido.

As DTNs trabalham de forma eficiente a transferência de dados em ambientes de difícil conexão ou grandes atrasos, através da comutação de mensagens e constante armazenamento e envio de dados pelos nós. Dessa forma, as DTNs permitem que o destinatário receba a mensagem independente de estar ligado à rede ou não, uma vez que um ou mais nós podem armazenar a mensagem até que o destino seja conectado à rede novamente.

Uma das formas de maximizar a taxa de entrega das mensagens em uma DTN é utilizar de um protocolo de roteamento conhecido como epidêmico, o qual, por meio de inundação, cada nó, ao receber uma mensagem, a armazena em um *buffer* e encaminha uma cópia para cada e todo nó adjacente que não possui aquela mensagem (UFRJ, 2021).

Neste trabalho é apresentado o desenvolvimento de um aplicativo mensageiro para *smartphones Android* que utiliza da tecnologia *Wi-Fi Direct*, presente em boa parte dos

*smartphones* atuais, para conectar os dispositivos em uma DTN epidêmica, de forma a que seja possível a troca de mensagens criptografadas em caso de não existir conectividade com a Internet.

Os capítulos seguintes encontram-se organizados da seguinte forma: O capítulo 2 aborda a fundamentação teórica em torno das tecnologias implementadas no projeto; O capítulo 3 apresenta os materiais e as metodologias empregadas no desenvolvimento do aplicativo; O capítulo 4 discorre sobre as implementações feitas e os resultados alcançados, além das limitações do aplicativo; Por último, o capítulo 5 contém as considerações finais em relação ao projeto, assim como sobre o que poderia ser feito para a continuação do trabalho.

# 2 FUNDAMENTAÇÃO TEÓRICA

Os temas e conceitos apresentados abaixo complementam e possibilitam o pleno entendimento do trabalho aqui descrito, pois foram fundamentais para a implementação deste projeto.

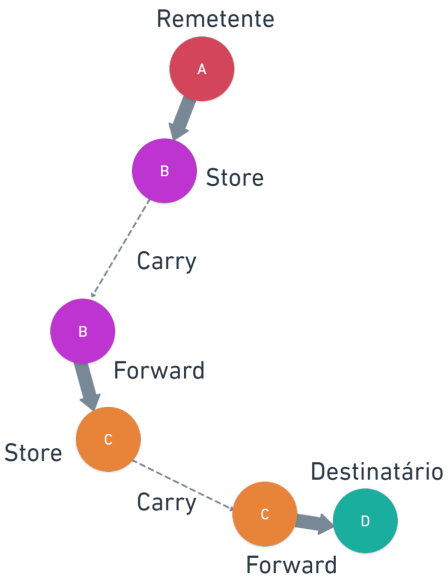
## 2.1 Redes tolerantes a atraso

Redes tolerantes a atraso, do inglês *Delay Tolerant Networks* (DTN), são redes projetadas para operar em condições extremas como guerras, resgates, acidentes, etc., onde pode existir atrasos e/ou rompimentos (falta de conectividade, resultando numa falta de caminho fim-a-fim instantâneo) (WIKIPEDIA, 2021).

As DTNs solucionam o problema dos atrasos utilizando protocolos de roteamento que utilizam a estratégia “armazenar e encaminhar” (*store-and-forward*), onde os dados são movidos incrementalmente e armazenados pelos nós da rede, esperando-se que eles irão eventualmente atingir o seu destino. Dessa forma, o destinatário de uma mensagem tem a possibilidade de recebê-la independente de estar ligado à rede ou não, uma vez que os nós armazenam a mensagem até que o destino esteja conectado novamente (NASA, 2022).

Além disso, as DTNs permitem que os nós se movam no espaço físico carregando mensagens em seu *buffer* (conceito denominado de *carry*) e repassem as mensagens em seu *buffer* a nós adjacentes, de forma que a comunicação alcance longas distâncias, através dos nós móveis intermediários — distâncias essas muito maiores do que conseguiriam caso fossem apenas dois nós na rede (COUTINHO, 2021).

**Figura 1 – Store, carry e forward**




**Fonte: Autoria própria (2022).**

Como visto na Figura 1, o nó A não alcança o nó D (o destinatário) diretamente, porém, com aplicação dos conceitos de DTN, foi possível que A se comunicasse com D através dos nós intermediários.

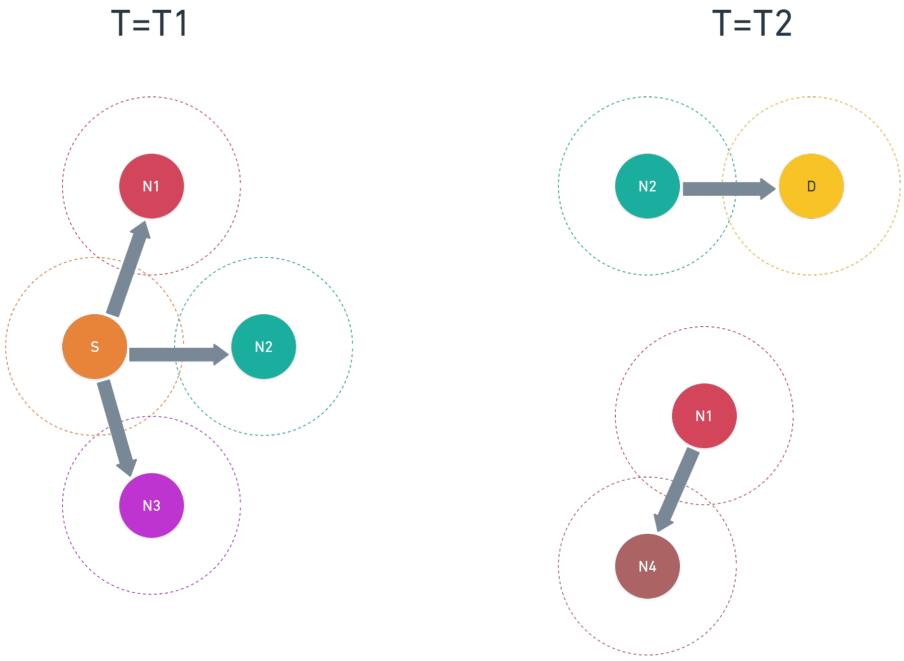
### 2.1.1 Roteamento epidêmico

Uma técnica usada para maximizar a probabilidade de uma mensagem ser transmitida com sucesso em uma rede DTN é replicar várias cópias dela mesma entre os nós, na esperança de que uma dessas cópias atingirá o seu destino com sucesso.

O roteamento epidêmico é o protocolo que utiliza desse princípio baseado em inundação, descrito acima. A principal vantagem é que dessa forma maximiza-se a taxa de sucesso na entrega das mensagens, porém a desvantagem é o rápido crescimento, em quantidade, das mensagens armazenadas pelos nós, o que se torna um problema se os nós possuem pouco espaço de armazenamento (SAID *et al.*, 2015).

Nesse protocolo, o nó que recebe uma mensagem, a armazena em um *buffer* e encaminha uma cópia para cada nó adjacente que encontra. Então a mensagem é difundida na rede por nós móveis até que todos os nós tenham os mesmos dados.

**Figura 2 – Roteamento epidêmico**




**Fonte: Autoria própria (2022).**

Na Figura 2, podemos ver um exemplo do roteamento epidêmico. Neste exemplo, o nó remetente S, envia uma mensagem para o nó destinatário D em um tempo T=T1. Os nós adjacentes a S no momento do envio eram os nós N1, N2 e N3, portanto estes recebem a mensagem de S e a mantêm em seu *buffer*. Em T=T2, os nós encontram-se em outro arranjo, e N2

agora é adjacente ao nó D, e portanto, faz o envio da mensagem de S. Enquanto isso, os outros nós, como o N1 por exemplo, continuam executando o algoritmo de roteamento epidêmico, espalhando a mensagem pela rede.

Dessa forma, podemos perceber que a mensagem de S foi entregue não por S, mas por outro nó participante da rede, de forma que S não precisou se locomover até D para entregar sua mensagem, esse trabalho foi realizado pelo nó N2.

## 2.2 Wi-Fi Direct

Baseado no fato de que todo dispositivo capaz de se conectar em uma rede *Wi-Fi* possui uma antena e um controlador, o *Wi-Fi Direct* é um sistema de conexão entre dispositivos que dispensa o uso de um roteador, utilizando apenas as antenas e controladores dos dispositivos para a conexão. Isso significa que é possível conectar aparelhos de forma *peer-to-peer*, assim como no *Bluetooth*, porém criando uma conexão muito mais rápida e segura, justamente por usar a mesma interface e alcançar as mesmas velocidades e distâncias típicas do *Wi-Fi* (ALLIANCE, 2022). Em contrapartida, a eficiência energética do *Bluetooth* é cerca de 30% maior quando comparado ao *Wi-Fi Direct* (PUTRA *et al.*, 2017).

A literatura diz que cada dispositivo teria um alcance de aproximadamente 200 metros em campo aberto e seria capaz de alcançar velocidades de até 250 Mbps (ALLIANCE, 2022). O *Wi-Fi Direct* também permite descobrir todos os dispositivos que suportam a tecnologia que estão próximos.

Além disso, o *Wi-Fi Direct* permite criar redes 1-para-N, em uma topologia estrela. O dispositivo central é chamado de servidor ou GO (Proprietário de grupo, do inglês *Group Owner*), e os outros de clientes. Um conjunto de dispositivos conectados a um GO é chamado de grupo. Vários grupos podem coexistir em um mesmo ambiente, porém não é possível a comunicação entre grupos distintos, e nem entre clientes do mesmo grupo diretamente. A comunicação se dá sempre entre cliente e GO, ou vice versa.

Os principais métodos implementados na API *Wi-Fi Direct* desenvolvida pelo *Google* para *smartphones Android* são: discovery(), que permite com que o dispositivo em questão descubra e seja descoberto por outros dispositivos, e o connect(), que permite que um dispositivo requisite a conexão a outro. Uma particularidade do método *connect* é que ao determinar qual dispositivo será o GO, o *Wi-Fi Direct* examina os recursos e capacidades dos dispositivos, como por exemplo o nível de bateria, e usa essas informações para escolher aquele que pode lidar com as responsabilidades de GO de forma mais eficiente (GOOGLE, 2022).

## 2.3 Criptografia

Criptografia é um mecanismo de segurança e privacidade que torna determinada comunicação (textos, imagens, vídeos e etc) ininteligível para quem não tem acesso à chave de "tradução" da mensagem (MAZIERO, 2020).

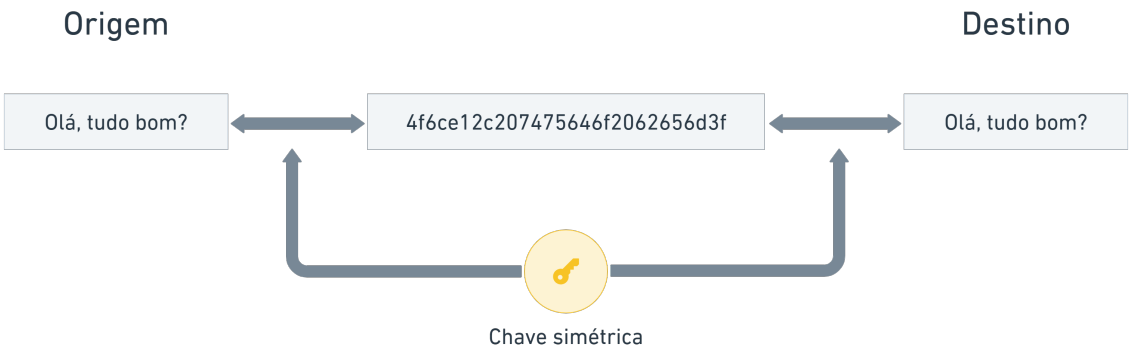
Nas comunicações digitais, a criptografia auxilia na proteção de todos os conteúdos transmitidos entre duas ou mais fontes, evitando a intercepção por parte de cibercriminosos, hackers e espiões, por exemplo.

### 2.3.1 Criptografia simétrica

Algoritmos de chave simétrica são algoritmos para criptografia que usam a mesma chave criptográfica para codificação de texto puro e decodificação do texto cifrado, como podemos ver na Figura 3. A chave, representa um segredo compartilhado entre duas ou mais partes que pode ser usado para manter a troca informação entre eles privada (MAZIERO, 2020).

Este requisito de que ambas as partes possuam acesso à mesma chave secreta é uma das principais desvantagens da criptografia de chave simétrica.

**Figura 3 – Criptografia simétrica**




**Fonte: Autoria própria (2022).**

Suas vantagens são principalmente a velocidade, pois os algoritmos de chave simétrica geralmente possuem tempo de resposta menor em relação aos algoritmos de chave assimétrica (MAZIERO, 2020). Exemplos dessa criptografia são: *Twofish*, *Serpent*, *Blowfish* e AES.

### 2.3.2 Criptografia assimétrica

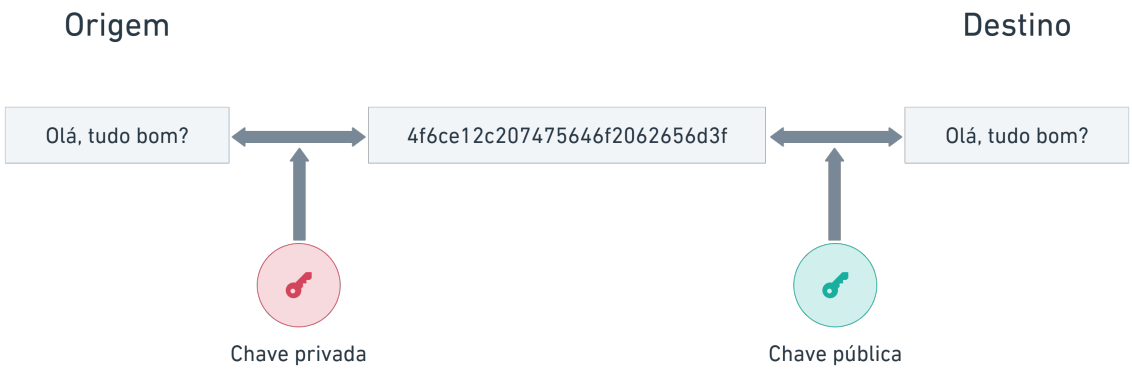
Criptografia assimétrica, é qualquer sistema criptográfico que usa um pares de chaves para suas operações: chaves públicas, que podem ser amplamente disseminadas, e chaves privadas que são conhecidas apenas pelo proprietário.

O termo assimétrica é devido ao fato das chaves realizarem funções opostas, uma a inversa da outra. Isto é, uma informação codificada com uma chave pública só pode ser deco-

dificada pela chave privada correspondente, como exemplificado pela Figura 4. Como contrapartida, na criptografia simétrica, a mesma chave para realiza ambas operações (MAZIERO, 2020).

Em um sistema de criptografia de chave pública, as chaves publicas podem ser amplamente disseminadas, sem risco de comprometer a segurança do sistema. Basta manter a chave privada em segurança, pois é esta que se utiliza para revelar as mensagens criptografadas.

**Figura 4 – Criptografia assimétrica**



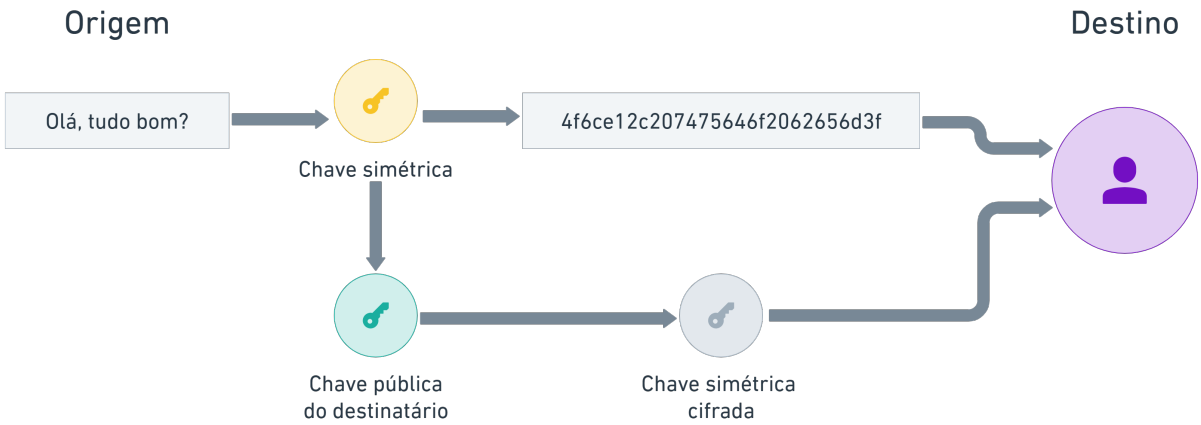

**Fonte: Autoria própria (2022).**

Exemplos dessa criptografia são: *Diffie-Hellman*, DSS, curvas elípticas e o RSA.

### 2.3.3 Criptografia híbrida

Em criptografia, um sistema criptográfico híbrido é aquele que combina a conveniência de um sistema criptográfico de chave assimétrica com a eficiência de um sistema criptográfico de chave simétrica. O funcionamento desse sistema encontra-se descrito na Figura 5.

**Figura 5 – Criptografia híbrida**




**Fonte: Autoria própria (2022).**

## 2.4 QR Code

O código QR (Código de resposta rápida, do inglês *Quick Response Code*) é um tipo de código de barras que pode codificar uma ampla variedade de informações. Diferentemente de códigos de barras tradicionais, que são utilizados para codificar apenas caracteres alfanuméricos (no caso do padrão 39, por exemplo), a grande vantagem do código QR é a possibilidade de codificação de várias formatos de dados como: numéricos, alfanuméricos, binário, *kanji* e inclusive *emojis* (ISO, 2015).

Outra diferença entre eles é que, enquanto o código de barras comum é lido apenas na horizontal, o código QR possui componentes tanto na vertical quanto na horizontal, o que reflete na aparência final do código e os tornam facilmente reconhecíveis e distinguíveis por humanos.

Atualmente a maior parte dos *smartphones* implementam de forma nativa a leitura de códigos QR direto do aplicativo de câmera padrão.

A Figura 6 exemplifica a diferença entre os formatos de codificação (barras e QR) ao codificar uma mesma informação.

**Figura 6 – Texto 'Hello World' codificado em ambos os formatos**

**(a) Code 39 (barras)** **(b) QR code**


**Fonte: Autoria própria (2022).**

# 3 MATERIAIS E MÉTODOS

A ênfase deste capítulo está em reportar o que foi feito, e como foi feito, para alcançar os objetivos deste trabalho. Este capítulo está subdividido em duas seções, uma que reporta os materiais utilizados e outra os métodos aplicados.

## 3.1 Materiais

Para o desenvolvimento do aplicativo proposto foi utilizado o ambiente de desenvolvimento integrado (IDE) oficial do *Android*, o *Android Studio*, em sua versão 2020.3.1 (*Arctic Fox*), rodando em uma máquina *macOS* versão 10.15.7 (*Catalina*). A linguagem de programação utilizada foi o Java, e os testes foram executados em três *smartphones* físicos com as características descritas na Tabela 1.

**Tabela 1 – Dispositivos utilizados no desenvolvimento do sistema**

| Marca | Modelo | Ano de lançamento | Versão do Android |
|---|---|---|---|
| Motorola | Moto X4 | 2017 | 9 |
| Motorola | Moto G6 | 2018 | 9 |
| Samsung | Galaxy S20 | 2020 | 11 |

**Fonte: Autoria própria (2022).**

Além disso, para facilitar os testes de capacidade de distância do aplicativo, usou-se um instrumento *range finder*, da marca *Wildgame Innovations*, modelo Halo R400. Esse dispositivo utiliza laser para medir distâncias de até 400 jardas (aproximadamente 365 metros) com uma precisão de +- 1 jarda (aproximadamente 90 centímetros). A Figura 7 apresenta o dispositivo.

**Figura 7 – Range finder Halo R400**


**Fonte: USA (2022).**

## 3.2 Métodos

Os métodos definem um plano geral do trabalho, com as principais atividades realizadas antes do processo de implementação em si. O que foi obtido com a realização dessas atividades está no Capítulo 4.

Para nortear a implementação do sistema, primeiramente foram definidos os requisitos funcionais e não funcionais. Os funcionais descrevem capacidades que o sistema deve possuir enquanto os não funcionais descrevem como serão feitos.

Após o levantamento de requisitos, foi possível dividir o projeto em 7 partes: identificação dos dispositivos, conexão entre os dispositivos, comunicação entre dispositivos de um mesmo grupo, comunicação entre dispositivos de grupos distintos, comunicação criptografada, cadastro de contatos e a interface do usuário. Cada uma dessas partes é explicada da subseção 3.2.3 à subseção 3.2.9.

### 3.2.1 Requisitos funcionais

RF1. O aplicativo deve ser capaz de identificar nós próximos;

RF2. O aplicativo deve ser capaz de se conectar automaticamente a um nó próximo identificado;

RF3. O aplicativo deve ser capaz de perceber uma desconexão e reiniciar o processo descrito na RF2;

RF4. O aplicativo deve ser capaz de conectar múltiplos dispositivos em topologia estrela, formando um grupo;

RF5. O aplicativo deve ser capaz de enviar mensagens de texto;

RF6. O aplicativo deve ser capaz de criptografar as mensagens;

RF7. O aplicativo deve ser capaz de exibir as mensagens recebidas;

RF8. O aplicativo deve ser capaz de trocar mensagens de texto entre cliente e GO (0 *hop*);

RF9. O aplicativo deve ser capaz de trocar mensagens entre dois clientes de um grupo (1 *hop*);

RF10. O aplicativo deve ser capaz de armazenar mensagens destinadas para fora do grupo em um *buffer* (N *hop*);

RF11. O aplicativo deve ser capaz de encaminhar mensagens do *buffer* conforme o roteamento epidêmico de DTNs (N *hop*);

RF12. O aplicativo deve ser capaz de permitir a geração de um novo par de chaves assimétricas;

RF13. O aplicativo deve ser capaz de exportar um par de chaves gerado;

RF14. O aplicativo deve ser capaz de importar e restaurar um par de chaves exportado anteriormente;

RF15. O aplicativo deve ser capaz de permitir adicionar um contato;

### 3.2.2 Requisitos não funcionais

RNF1. As mensagens devem ser criptografadas usando criptografia hibrida AES + RSA;

RNF2. O aplicativo deve exibir as mensagens recebidas e enviadas em uma única lista, ordenada pelo horário de recebimento de cada mensagem;

RNF3. O aplicativo deve usar códigos QR para a rotina de adicionar usuários;

RNF4. O aplicativo deve ser desenvolvido para dispositivos *Android* utilizando Java, utilizando o paradigma de orientação a objetos;

RNF5. O aplicativo deve possuir apenas uma tela a fim de facilitar a implementação;

### 3.2.3 Identificação dos dispositivos

Para que usuários consigam se comunicar e trocar mensagens entre si (Requisito funcional RF5) em uma rede de possíveis múltiplos usuários, é preciso que cada um tenha um identificador único e que exista uma forma de compartilhar esse identificador entre eles. Para isso, neste projeto, foi utilizada a chave pública, do par de chaves assimétrica, como meio de identificar tanto os usuários quanto a origem e o destino de suas mensagens.

O aplicativo então permite que um novo par de chaves seja gerado pelo usuário ou que este importe uma chave gerada previamente, a fim de se identificar. O processo de definição das chaves será chamado, neste trabalho, de inicializar. Além disso foi desenvolvido uma forma de compartilhar a chave pública através de códigos QR (Requisitos RF12, RF13, RF14 e RNF3).

### 3.2.4 Conexão entre os dispositivos

Para que dois usuários, já inicializados, se conectem usando o *Wi-Fi Direct*, precisa-se ativar o método de descoberta *discovery* (RF1), disponibilizada pela API do *Android*, e escolher um dos dispositivos encontrados para se conectar com o método de conexão *connect* (RF2).

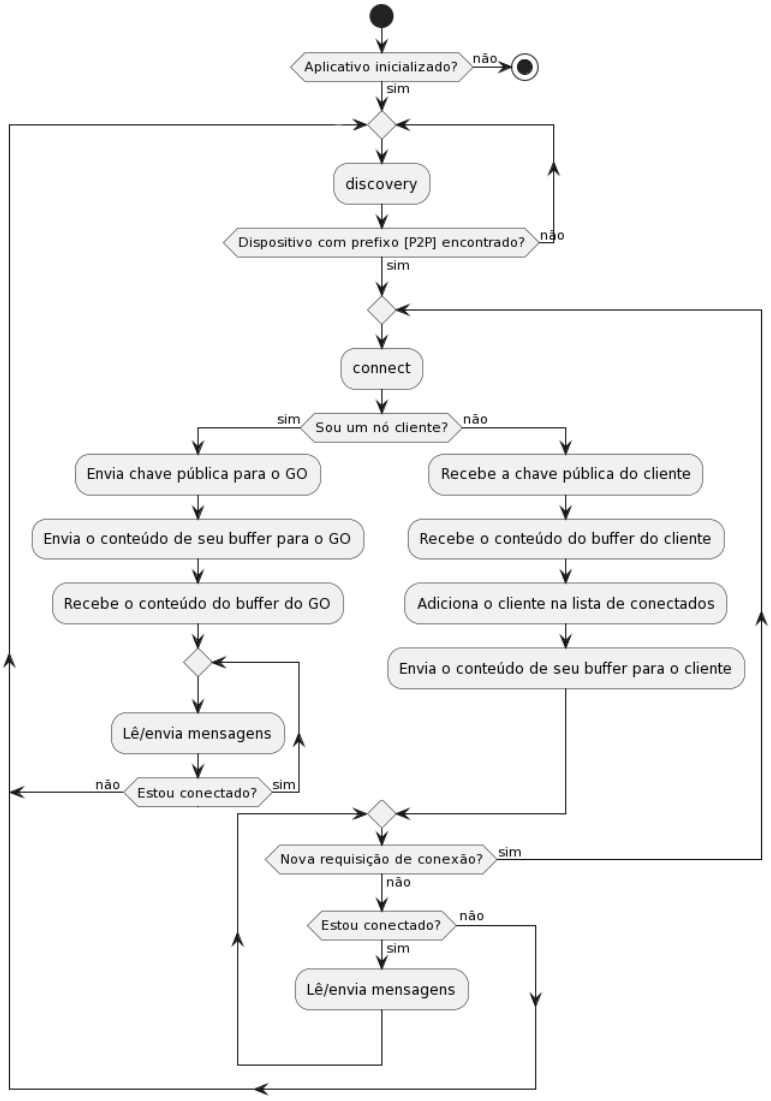
Neste projeto, a fim de evitarmos conexão com outros dispositivos como televisões (que também usam o *Wi-Fi Direct*), modificamos o nome do dispositivo para incluir uma *tag* (prefixo) de forma que seja possível filtrar apenas *smartphones* rodando o aplicativo durante a rotina de descoberta/conexão. A *tag* escolhida foi [P2P].

Concluída a conexão entre dois dispositivos, o GO continua sendo visível para outros dispositivos que iniciarem o método de descoberta *discovery*, possibilitando assim conectar múltiplos dispositivos em topologia estrela (RF4).

É durante o processo de conexão também que acontece a troca de mensagens de reconhecimento (ACK), onde são trocadas as chaves públicas a fim de ambos conhecerem um ao outro. Além da mensagem ACK, também é realizada a atualização dos *buffers* dos nós conectados, com mensagens que eles possam não possuir ainda.

O aplicativo também é capaz de perceber uma eventual desconexão, reiniciando a rotina de descoberta/conexão (RF3). Toda a rotina aqui explicada é exibida na figura 8.

**Figura 8 – Rotina de conexão entre dispositivos**




**Fonte: Autoria própria (2022).**

### 3.2.5 Comunicação entre dispositivos de um mesmo grupo

Como mencionado na seção 2.2, o *Wi-Fi Direct* permite a criação de uma rede 1-N, porém não permite a comunicação entre grupos distintos, nem entre clientes de um mesmo grupo. Portanto, foi desenvolvido um protocolo de roteamento, onde para possibilitar a comunicação de cliente com cliente, deve-se primeiro enviar a mensagem para o GO, de forma que este encaminhe para o cliente destino correto. Ou seja, a função de roteamento é atribuída apenas a nós GO. Já a comunicação cliente-GO, ou vice versa, é feita de forma direta, sem necessidade de roteamento algum (RF8 e RF9). A parte verde das figuras 9 e 10 explica a lógica do roteamento de mensagens para membros de um mesmo grupo. Já a parte rosa, que contempla a parte do roteamento epidêmico, é explicada na subseção 3.2.6.

### 3.2.6 Comunicação entre dispositivos de grupos distintos

Para possibilitar a comunicação entre grupos distintos foi aplicado o conceito de roteamento epidêmico. Quando um membro de um grupo, seja ele GO ou cliente, envia mensagem para algum nó que não faz parte desse grupo, o GO irá armazenar essa mensagem em seu *buffer* e encaminhar para todos os membros do grupo, para que esses guardem a mensagem em seu *buffer* também. Dessa forma, caso qualquer um desses nós saia do grupo atual, e venha a se conectar a outro grupo, fará o envio das mensagens em seu *buffer* durante o processo de conexão (RF10 e RF11), como explicado na subseção 3.2.4. A parte rosa das figuras 9 e 10 explica a lógica do roteamento epidêmico. Já a parte verde, que contempla a parte do roteamento de mensagens para membros de um mesmo grupo é explicada na subseção 3.2.5.

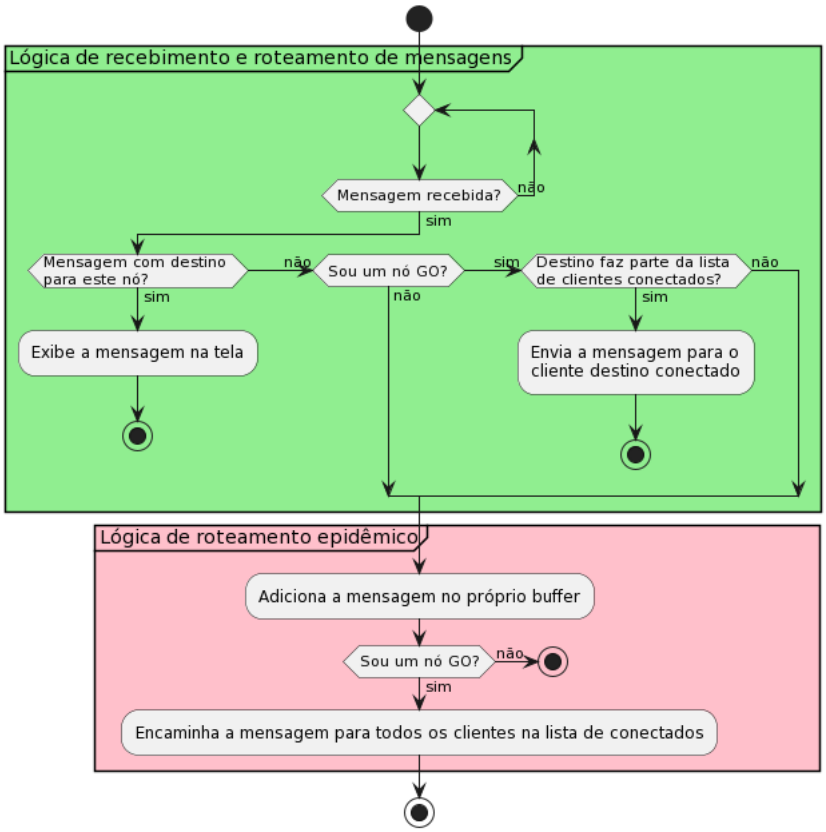
### 3.2.7 Comunicação criptografada

Como as vezes é necessário o roteamento de mensagens por nós intermediários, é importante que as mensagens sejam criptografadas para que estes não tenham acesso ao conteúdo da mensagem que estão repassando, garantindo a privacidade de quem utiliza o aplicativo (RF6). A criptografia implementada nesse projeto foi a híbrida, utilizando RSA + AES (RNF1) como modelos de criptografia assimétrica e simétrica, respectivamente.

### 3.2.8 Cadastro de contatos

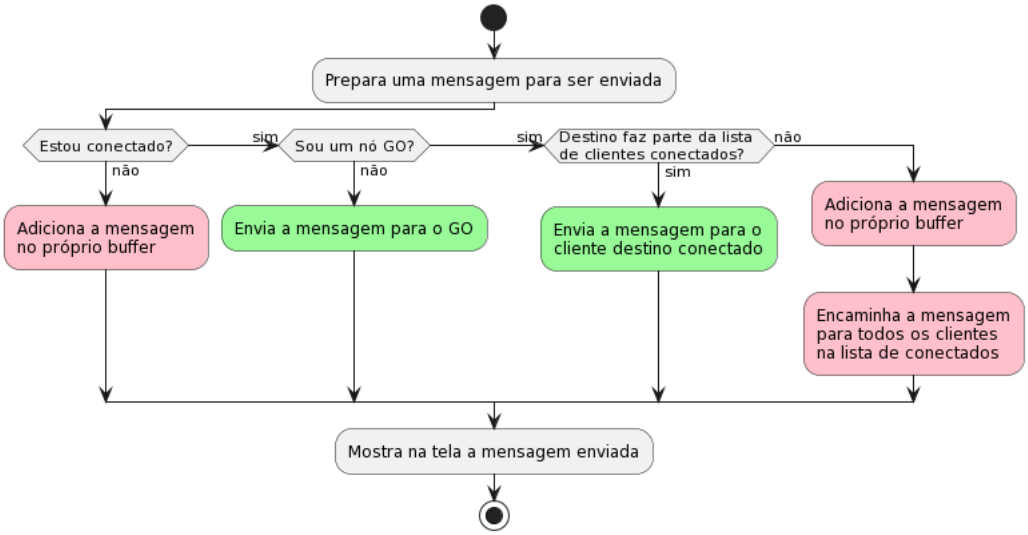
Assim como no campo da telefonia é necessário conhecer previamente o número de telefone do destinatário, no aplicativo aqui proposto, precisa-se haver a troca prévia da chave pública entre os usuários que desejam se comunicar (RF15). Para facilitar a troca dessa chave, visto que essa é uma sequência de caracteres muito maior que um simples número de telefone,

**Figura 9 – Rotina de recebimento e roteamento de mensagens**




**Fonte: Autoria própria (2022).**

**Figura 10 – Rotina de envio de mensagens**



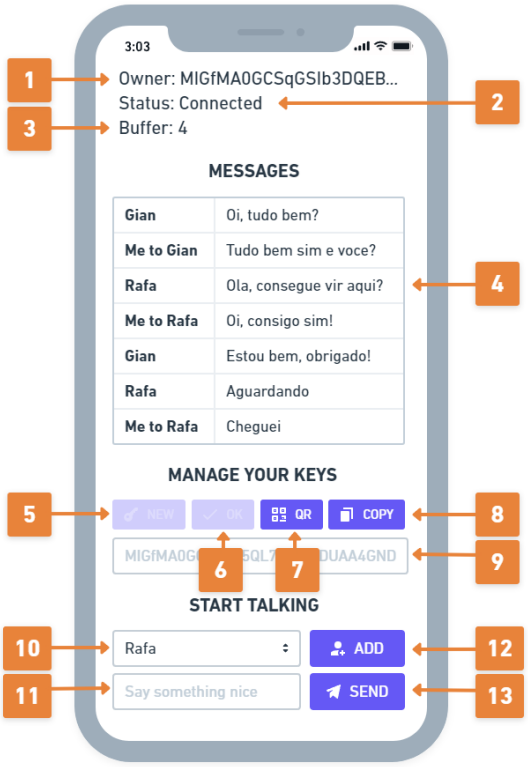

**Fonte: Autoria própria (2022).**

foi implementada a codificação da chave pública para o formato QR, assim como a capacidade de exibição e escaneamento de códigos QR (RNF3).

### 3.2.9 Interface do usuário

Para possibilitar que o usuário realize as funções propostas acima, principalmente as referentes a exibição (RF7 e RNF2), foi projetada uma interface simples (porém completa), de uma só tela (RNF5), exibida na Figura 11. A função prevista para cada um dos componentes é descrita na Tabela 2.

**Figura 11 – Projeto da tela do aplicativo**




**Fonte: Autoria própria (2022).**

**Tabela 2 – Componentes da interface gráfica**

| Nº | Nome | Tipo | Descrição |
|---|---|---|---|
| 1 | Owner | Texto | Exibe a chave pública, ou o nome (em caso de ser um nó já cadastrado nos contatos), do nó que é o GO no momento. |
| 2 | Status | Texto | Indica o status do dispositivo no momento. Pode ser um dos seguintes: *Waiting for keys, Discovery, Discovery failed, Wifi is disabled, Connecting* e *Connected* |
| 3 | Buffer | Texto | Indica a quantidade de mensagens no *buffer* de mensagens a serem encaminhadas. |
| 4 | Mensagens | Lista | Contém tanto as mensagens enviadas pelo usuário quanto as mensagens destinadas a ele. |
| 5 | NEW | Botão | Botão que gera um novo par de chaves aleatório. O par de chaves gerado é exibido no item 9 da tela. |
| 6 | OK | Botão | Botão que confirma o par de chaves e completa a inicialização do aplicativo. |
| 7 | QR | Botão | Exibe a chave pública em formato de QR. |
| 8 | COPY | Botão | Copia o par de chaves para a área de transferência do dispositivo, permitindo assim que seja exportada. |
| 9 | Chaves | Texto editável | Conjunto de caracteres que representam tanto a chave pública quanto a privada. Aqui é possível inserir uma chave criada previamente. |
| 10 | Contatos | Dropdown | Exibe e permite selecionar o destinatário da mensagem a ser enviada. |
| 11 | Mensagem | Texto editável | Permite escrever qualquer mensagem para ser enviada. |
| 12 | ADD | Botão | Abre o leitor de código QR que permite adicionar um novo contato. |
| 13 | SEND | Botão | Envia a mensagem para o destinatário. Pode ser usado assim que o dispositivo foi inicializado, não sendo necessário estar conectado para enviar. Mensagens enviadas dessa forma vão automaticamente para o *buffer*. |

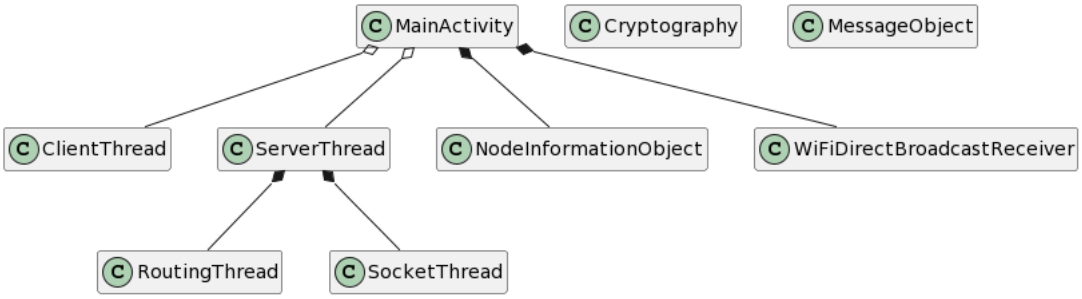
**Fonte: Autoria própria (2022).**

# 4 RESULTADOS

## 4.1 Implementação do sistema

A implementação do sistema foi desenvolvida na linguagem Java através da programação orientada a objetos (RNF4), onde cada classe possui uma função especifica. O diagrama UML apresentado na Figura 12 representa a estrutura do código desenvolvido (FORASTIERI, 2022). As subseções a seguir (subseção 4.1.1 à subseção 4.1.8) apresentam uma breve explicação da função de cada uma das classes do diagrama da Figura 12. Além disso, no apêndice seção A.1 é possível encontrar trechos de código Java que complementam entendimento de cada uma dessas classes.

**Figura 12 – Diagrama UML**




**Fonte: Autoria própria (2022).**

### 4.1.1 Classe *MainActivity*

A classe *MainActivity* é a classe raiz, responsável pelos elementos da interface gráfica, tanto na questão da atualização da interface, quanto pelo tratamento dos comandos inseridos pelo usuário através da mesma. É responsável também por instanciar objetos fundamentais como *ServerThread/ClientThread* (subseção 4.1.6), *NodeInformationObject* (subseção 4.1.7) e *WifiBroadcastReceiver* (subseção 4.1.8). Além disso, essa classe instancia os *pipes* que permitem a comunicação com as *threads ServerThread* e *ClientThread*. Os métodos *discover()* e *connect()*, explicados na subseção 3.2.4, também são implementados aqui.

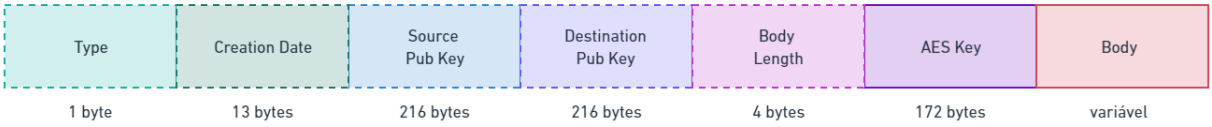
### 4.1.2 Classe *Cryptography*

A classe *Cryptography* é uma classe auxiliar, que possui a implementação de métodos utilizados para a codificação e decodificação de mensagens, além de funções para a criação de chaves simétricas e assimétricas. Essa classe não é instanciada em momento algum, apenas utiliza-se de suas funções durante o projeto.

### 4.1.3 Classe *MessageObject*

A classe *MessageObject* é responsável pelo tratamento das mensagens, contendo métodos referentes a criação, leitura e envio de mensagens. Funções da classe *Cryptography* (subseção 4.1.2) são amplamente utilizadas aqui. A estrutura de mensagem é descrita na Figura 13. Os campos pontilhados indicam as informações não criptografadas, enquanto os outros possuem seu conteúdo criptografado (apenas para mensagens do tipo 0 ou 1). O tipo da mensagem, indicado pelo campo *Type*, pode assumir os valores delimitados na Tabela 3.

**Figura 13 – Estrutura de uma mensagem**




**Fonte: Autoria própria (2022).**

**Tabela 3 – Tipos de mensagens**

| Type (1 byte) | Criptografia | Classificação | Uso |
|---|---|---|---|
| 0 | Sim | Mensagem normal | Usada no envio das mensagens escritas pelos usuários. |
| 1 | Sim | Mensagem de sistema | Usada para notificar usuários sobre mudanças na topologia do grupo. |
| 2 | Não | Mensagem de sistema | Usada para troca de mensagens ACK. |

**Fonte: Autoria própria (2022).**

### 4.1.4 Classe *RoutingThread*

A classe *RoutingThread* é uma classe que extende da classe padrão *Thread* e é instanciada apenas por nós GO, implementando a função de roteamento das mensagens. Todas as mensagens recebidas por um GO passam pela classe *RoutingThread*, e podem ser redirecionadas para si próprio ou para outros nós conectados.

### 4.1.5 Classe *SocketThread*

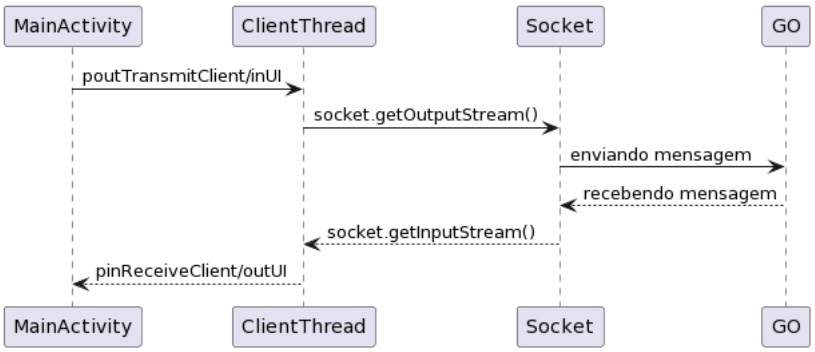
A classe *SocketThread* é uma classe que extende da classe padrão *Thread* e é responsável por interfacear com o *socket* que conecta os nós. Quando uma conexão é concretizada, cada um dos nós, seja ele GO ou cliente, instancia um objeto *SocketThread* do seu lado.

### 4.1.6 Classes *ClientThread* e *ServerThread*

As classes *ClientThread* e *ServerThread* são classes que extendem da classe padrão *Thread* e são responsáveis por tratar a conexão, o envio e recebimento das mensagens de nós. Essas classes agem como intermediárias entre a *MainActivity* (subseção 4.1.1) e a *SocketThread* (subseção 4.1.5), mantendo *pipes* para comunicar entre elas.

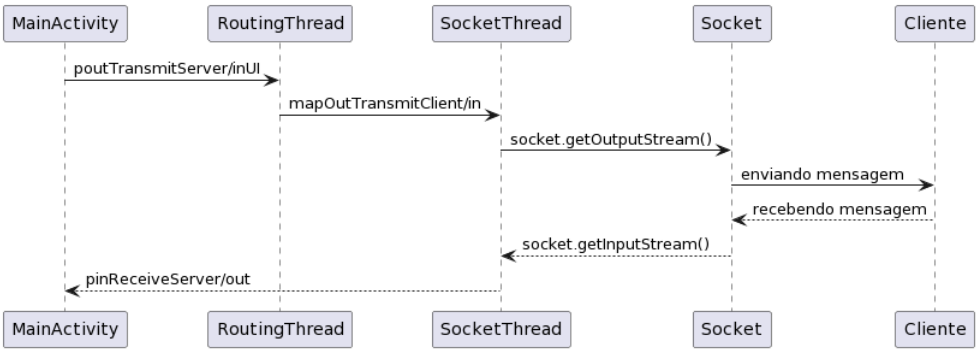
A dinâmica entre essas *threads* quando se envia/recebe uma mensagem é explicada na Figura 14 e na Figura 15. Nessas imagens pode-se ver quais métodos e quais pipes utiliza-se para mover a mensagem entre as *threads* até alcançar o destino.

**Figura 14 – Lógica de interação entre as classes para um nó cliente**




**Fonte: Autoria própria (2022).**

**Figura 15 – Lógica de interação entre as classes para um nó GO**




**Fonte: Autoria própria (2022).**

### 4.1.7 Classe *NodeInformationObject*

A classe *NodeInformationObject* é responsável por armazenar todos os valores referentes ao estado atual do nó, como seu par de chaves, se está conectado no momento, seu papel na rede (GO ou cliente), seus contatos adicionados, as mensagens em seu *buffer*, todas as mensagens recebidas e todas as enviadas. Essas informações são os dados fundamentais para uma eventual implementação de persistência de dados, para tornar possível reiniciar o

aplicativo sem perder os dados que importam ao usuário. A questão da persistência de dados é explicada na subseção 4.3.2.

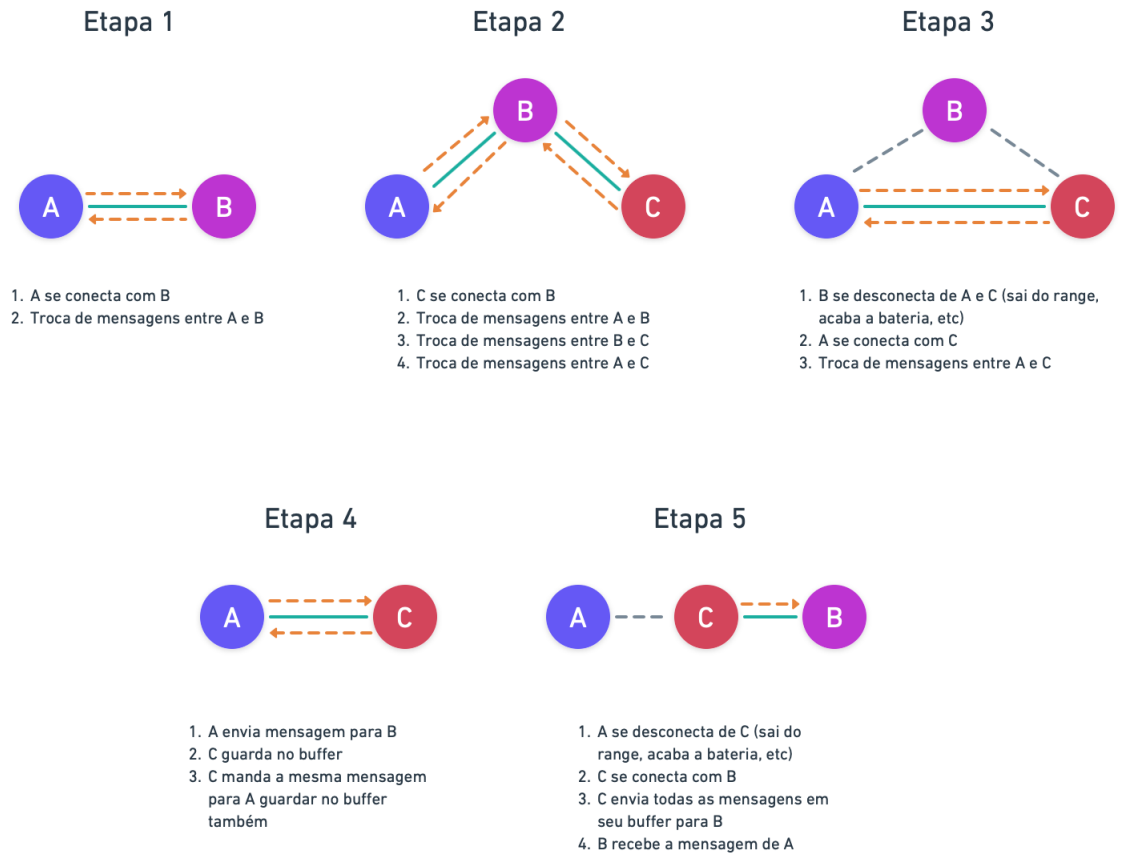
### 4.1.8 Classe *WiFiDirectBroadcastReceiver*

Classe responsável por implementar os métodos relacionados ao *Wi-Fi Direct*, tais como a identificação de dispositivos próximos e mudanças no estado do dispositivo relacionados ao *Wi-Fi Direct* em si.

## 4.2 Resultados práticos

A fim de exemplificar toda a capacidade do aplicativo, e levando em conta a limitação de se possuir apenas três *smartphones* disponíveis para uso (Tabela 1), um teste em 5 etapas foi planejado conforme a Figura 16.

**Figura 16 – Teste de capacidades do aplicativo**




**Fonte: Autoria própria (2022).**

Para simular a movimento de saída do alcance do *Wi-Fi* durante os testes, utilizou-se do mecanismo de colocar o *smartphone* em modo avião, forçando assim a desconexão. Cada uma

das etapas do teste projetado testa um ou mais dos requisitos funcionais conforme apresentado na Tabela 4.

**Tabela 4 – Requisitos funcionais exemplificados em cada uma das etapas**

| Etapa | Requisitos funcionais exemplificados |
|---|---|
| 1 | RF1, RF2, RF5, RF6 e RF8 |
| 2 | RF1, RF2, RF5, RF6, RF4, RF8 e RF9 |
| 3 | RF1, RF2, RF3, RF5, RF6 e RF8 |
| 4 | RF5, RF6 e RF10 |
| 5 | RF1, RF2, RF3, RF5, RF6 e RF11 |

**Fonte: Autoria própria (2022).**

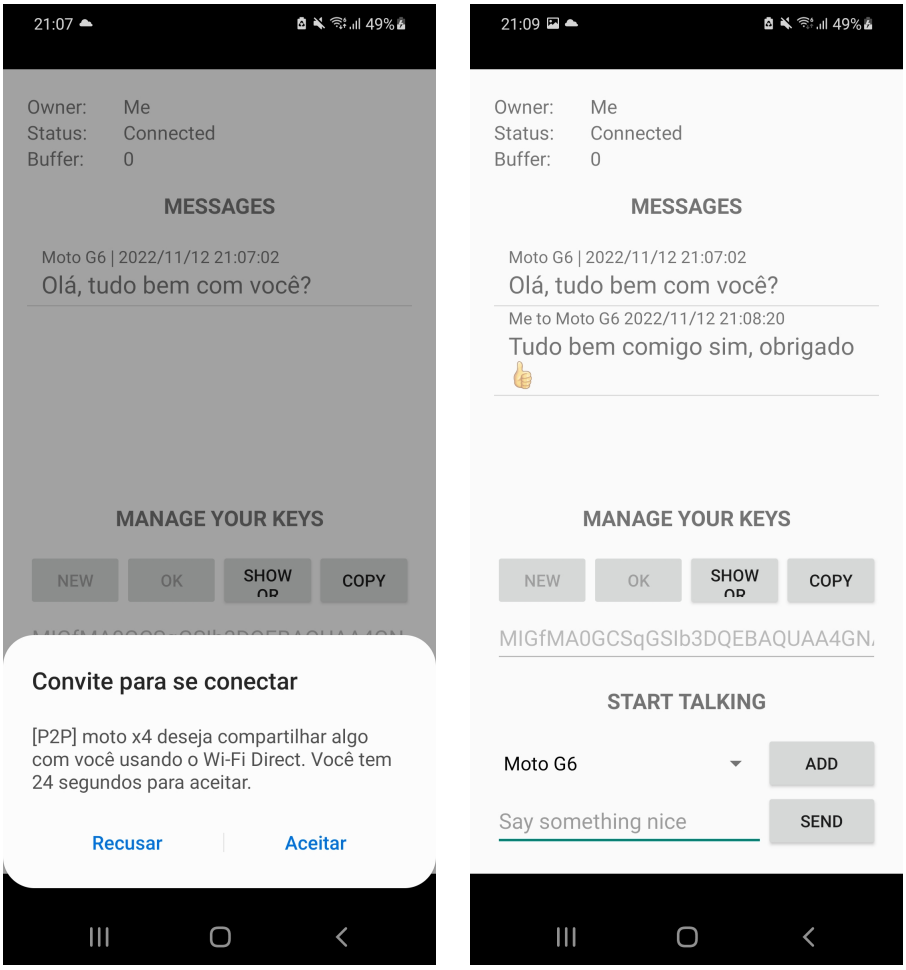
O teste em 5 etapas foi executado 5 vezes seguidas, e o resultado foi 100% satisfatório, onde a conexão foi efetivada 100% das vezes, e 100% das mensagens foram entregues aos destinatários. Durante esses testes a distância entre os *smartphones* era de no máximo 5 metros, por ter sido realizado dentro de casa.

Testes de distância máxima também foram realizados, com o auxílio do instrumento *range finder* (Figura 7). Os resultados variaram a depender da situação, onde a distância máxima para a conexão de dois dispositivos foi de 130 metros durante o processo de *discovery*. Para dispositivos já conectados, a troca de mensagens foi possível até 180 metros. Usando 1 salto (cliente - GO - cliente), alcançou-se a distância de 260 metros. Os números apresentados aqui representam uma média obtida a partir de 5 amostragens.

As funcionalidades descritas nos requisitos funcionais (subseção 3.2.1) que não foram contempladas no **??** (RF7, RF12, RF13, RF14 e RF15), são relacionadas a funcionalidades da interface gráfica. O resultado final da interface implementada é exibido na Figura 17, e pode-se perceber que este ficou bastante similar ao que foi projetado na Figura 11. A rotina de adicionar um usuário é demonstrada na Figura 18.

**Figura 17 – Tela desenvolvida**

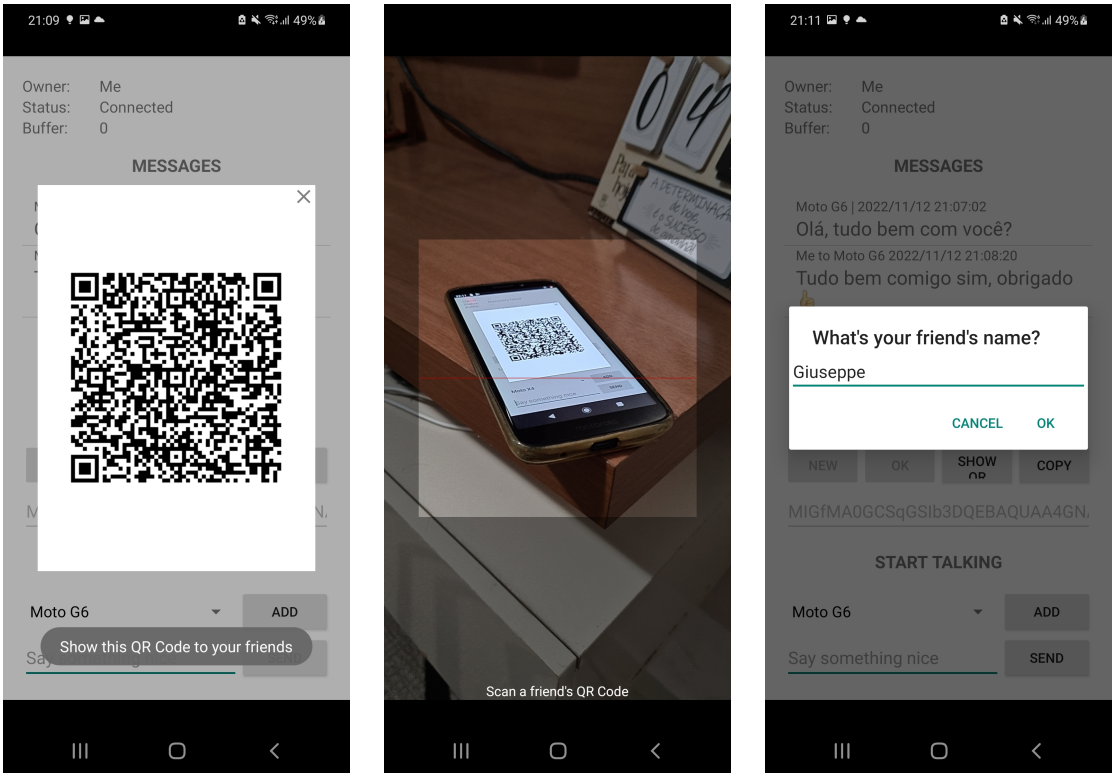
**(a) Requisição para se conectar** **(b) Tela completa**




**Fonte: Autoria própria (2022).**

**Figura 18 – Rotina de adicionar um contato**

**(a) Exibir a própria chave pública** **(b) Escanear uma chave pública** **(c) Escolher um nome para o usuário**




**Fonte: Autoria própria (2022).**

## 4.3 Limitações

### 4.3.1 Relógios têm que estar sincronizados

Devido à lógica da implementação do tempo de vida (TTL) das mensagens do *buffer* envolver uma comparação entre a data e horário de envio da mensagem (informada pelo *smartphone* remetente da mensagem) e a data e horário presente no *smartphone* destino, caso essas estejam dessincronizados, pode ser que o destinatário considere essa mensagem como uma mensagem expirada, ou seja, que já pode ser descartada pois o TTL já foi alcançado.

Falta de sincronia é algo de se esperar em situações críticas como desastres ou guerra, porém, essa dessincronia ocorreria na casa dos segundos, ou dos minutos, e dificilmente alcançaria horas ou dias, pois passaria a ser perceptível aos usuários. Caso um usuário não ajuste o seu relógio, ou ativamente altere o horário de seu *smartphone* para um valor incorreto, o único prejudicado seria esse mesmo usuário, pois suas mensagens seriam rapidamente descartadas pelos outros nós. Há um incentivo então para que os usuários mantenham seus dispositivos com o horário correto.

Portanto, o problema de dessincronia baixa pode ser contornado usando uma politica de TTL na casa de horas ou dias, assim dificilmente uma mensagem legítima seria descartada de forma prematura.

### 4.3.2 Persistência de dados

No aplicativo desenvolvido não foi implementada a questão da persistência de dados. Ou seja, caso o aplicativo seja encerrado e reiniciado, não são restauradas as informações do usuário, como por exemplo o seu par de chaves, seus contatos adicionados, suas mensagens antigas e suas mensagens no *buffer* da última sessão. A implementação dessa funcionalidade ficou fora do escopo do projeto apenas por questões de prazos de desenvolvimento, mas o código atual permite a inclusão posterior dessa funcionalidade sem muitos ajustes posteriormente.

### 4.3.3 Usuários precisam permitir a conexão

Devido a mecanismos de segurança implementados nativamente no *Android*, conexões *Wi-Fi Direct* não podem ser aceitas programaticamente (via código), apenas através de um *popup* que aparece na tela para que usuário tenha ciência da conexão que está sendo requisitada. Esse *popup* pode ser visto na Figura 17 (a).

A única maneira de contornar isso seria através de *root* do sistema, procedimento na qual um usuário torna-se superadministrador e passa a ter controle sobre áreas e recursos do dispositivo no qual não se tinha antes. O problema do *root* é que, ao realizá-lo, perde-se a

garantia de fábrica do dispositivo, e arrisca inserir diversas falhas de segurança no dispositivo. Por isso neste projeto, foi evitado seguir pelo caminho do *root*, e manter essa limitação em questão.

### 4.3.4 O aplicativo precisa estar aberto

Como explicado na subseção 4.3.3, devido ao fato de que o usuário precisa aceitar as conexões através de um *popup* na tela, é necessário que o aplicativo seja mantido aberto, com a tela ligada, pra que as conexões possam ser aceitas.

### 4.3.5 Não existe confirmação de entrega das mensagens

Essa funcionalidade ficou de fora do projeto devido ao fato de estar-se trabalhando com DTNs, onde já se pode haver um grande atraso na entrega da mensagem, quem dirá na entrega da mensagem de confirmação. Além disso, cada mensagem transmitida na rede acabará eventualmente entrando no *buffer* de diversos nós pelo caminho, portanto, cada mensagem a mais na rede contribui para uma sobrecarga na rede.

Além disso, devido ao fato de não existir confirmação de recebimento, como explicado acima, a única forma de uma mensagem deixar o *buffer* de um nó é através do TTL indicar mensagem expirada. Portanto existe um dilema, aumentar o TTL aumenta a chance de uma mensagem chegar ao destinatário, porém aumenta o uso do *buffer* dos nós.

### 4.3.6 Não é possível a troca de mensagens de voz e imagens

Nesse projeto não foi implementado o envio de mensagens de voz e imagens. O envio desse formato seria possível através de uma conversão direta para o formato texto, porém aumentaria drasticamente o tamanho das mensagens e a ocupação dos *buffers*.

### 4.3.7 Não há entrega de mensagens em caso de usuários estáticos no espaço

Devido à forma que o projeto foi realizado, um nó, após conectado, só se desconecta em caso de perder o sinal *Wi-Fi*, o que acontece apenas se esses ficarem além da distância de alcance, desligarem a antena *Wi-Fi* manualmente ou acabar a bateria.

Portanto, em casos de grupos onde os usuários são estáticos, ou seja, não se movem no espaço, estes tendem a permanecer no grupo em que estão e nunca carregar mensagens de um grupo para o outro (processo de *carry* explicado na seção 2.1). Logo, para um bom funcionamento do projeto, é preciso que existam sempre usuários que estejam se movimentando e que possam fazer a ponte entre os grupos.

### 4.3.8 Versões mais recentes do *Android* não permitem alterações de parâmetros do telefone programaticamente

Para o correto funcionamento do projeto, precisa-se que esse, durante o método de descoberta (*discovery*), tenha o prefixo [P2P] no nome do telefone e da rede visível a todos (como explicado na subseção 3.2.4). Nas versões mais antigas do *Android*, como por exemplo na versão 9, a inserção desse prefixo é possível programaticamente, ou seja, diretamente através de código do aplicativo, fazendo com que o usuário não tenha que realizar essas alterações de forma manual.

A partir da versão 10 já não é mais possível alterar o nome do telefone e o nome da rede *Wi-Fi Direct* programaticamente, fazendo com que o usuário tenha que realizar o passo de alterar o nome de seu dispositivo e rede *Wi-Fi Direct* manualmente, nas configurações do dispositivo, antes mesmo de poder usar o aplicativo.

### 4.3.9 Todas as mensagens recebidas e enviadas são exibidas em uma única lista

Diferentemente dos aplicativos de mensageria populares como o *WhatsApp*, que possui telas distintas para cada uma das conversas com usuários distintos, o aplicativo aqui desenvolvido foi projetado para utilizar apenas uma lista com todas as mensagens, tanto as recebidas, quanto as enviadas, como especificado pelo requisito não funcional RNF2. O resultado pode ser visto na Figura 11. Dessa forma, fica para o próprio usuário a responsabilidade de acompanhar o desenvolvimento das conversas que ele está participando.

O aplicativo foi desenvolvido dessa forma apenas para simplificar a implementação, não havendo prejuízo ao projeto como prova de conceito.

### 4.3.10 Distância alcançada

Como apresentado na seção 4.2, a distância máxima de conexão foi de 130 metros, e para transmissão de mensagens entre cliente e GO (0 saltos) foi de 180 metros, valores inferiores à distância de 200 metros encontrada na literatura como de se esperar (ALLIANCE, 2022). Além disso, o resultado de distância de troca de mensagens quando utilizado 1 salto (cliente - GO - cliente), foi de 260 metros, o que não representa o dobro da distância encontrada nos testes entre cliente e GO, como esperado.

Os motivos para isso podem ser vários, como o fato dos dispositivos utilizados serem de modelo e fabricante diferentes (Tabela 1) e o fato dos testes terem sido realizados em uma avenida movimentada, com muitas casas ao redor e muitas pessoas transitando.

É de se esperar que em avenidas como a do teste em questão, com a presença de, por exemplo, dispositivos *bluetooth* e outros roteadores *Wi-Fi*, que a atenuação dos sinais será maior por haver interferência (MAHANTI *et al.*, 2010).

# 5 CONCLUSÃO

Dados os objetivos do projeto definidos inicialmente, o desenvolvimento deste projeto permitiu com que fosse comprovada a possibilidade de se criar uma rede de comunicação independente de infraestrutura, tolerante a atrasos, com mensagens criptografadas e utilizando de hardware amplamente acessível à população, os *smartphones*.

Com isso, o aplicativo aqui desenvolvido poderia ser utilizado para comunicação em locais remotos, em eventual censura da comunicação por parte dos governos, ou em outras situações onde exista falta de conectividade e a necessidade de comunicação.

Apesar dos resultados expostos nos capítulos anteriores confirmarem que, de fato, os objetivos do projeto foram alcançados com sucesso, o aplicativo apresenta diversas limitações conforme descrito na seção 4.3. Porém, nenhuma dessas limitações inviabiliza a utilidade demonstrada pelo aplicativo, pois a maioria dessas limitações surgiram a partir das escolhas de *design* adotadas durante a fase de planejamento do projeto.

As soluções para boa parte das limitações poderiam ser implementadas posteriormente caso fossem necessárias, como por exemplo, a possibilidade de envio de imagens, a persistência de dados, e até mesmo a confirmação de entrega de mensagens.

Já as limitações relacionadas aos mecanismos de segurança implementados nativamente no *Android* poderiam ser corrigidas caso fosse adotada a utilização do *root* de sistema, que novamente, não foi utilizado por escolhas de *design*, por limitar a adoção do aplicativo por parte dos usuários.

Vale salientar que os testes comprovam o funcionamento para três dispositivos, pois era a quantidade disponível para o uso nesse trabalho, e não significa que funcionaria para adoção em larga escala. Neste sentido, como continuação deste trabalho, seria possível implementar as soluções para as limitações expostas acima, melhorias na experiência do usuário como um todo, através de uma interface mais intuitiva e agradável além de executar testes mais extensos com mais dispositivos fazendo parte da rede.

# REFERÊNCIAS


ALLIANCE, W.-F. **Wi-Fi Direct**. 2022. Disponível em: https://www.wi-fi.org/content-tags/wi-fi-direct?page=1.

COUTINHO, G. L. **Delay Tolerant Networks (DTN)**. 2021. Disponível em: https://www.gta.ufrj.br/grad/06_2/gustavo/arquitetura.htm.

FORASTIERI, G. **Serviço de mensageria utilizando redes tolerantes a atraso**. 2022. Disponível em: https://github.com/levelgigio/p2p-dtn-messenger.

GOOGLE. **Top Apps Gratuitos no Brasil**. 2021. Disponível em: https://play.google.com/store/apps/top.

GOOGLE. **Criar conexões P2P com o Wi-Fi Direct**. 2022. Disponível em: https://developer.android.com/training/connect-devices-wirelessly/wifi-direct.

ISO. **QR Code bar code symbology specification**. [*S.l.*]: ISO/IEC, 2015.

LAGO, D. **Há mais de um smartphone por habitante no Brasil**. 2020. Disponível em: https://veja.abril.com.br/blog/matheus-leitao/ha-mais-de-um-smartphone-por-habitante-no-brasil/.

MAHANTI, A. *et al.* Ambient interference effects in wi-fi networks. v. 6091, p. 160–173, 04 2010.

MAZIERO, C. **Sistemas Operacionais: Conceitos e Mecanismos**. [*S.l.*: *s.n.*], 2020. ISBN 978-85-7335-340-2.

NASA. **Delay/Disruption Tolerant Networking Overview**. 2022. Disponível em: https://www.nasa.gov/directorates/heo/scan/engineering/technology/disruption_tolerant_networking_overview.

PUTRA, G. D. *et al.* Comparison of energy consumption in wi-fi and bluetooth communication in a smart building. p. 1–6, 2017.

SAID, A. *et al.* The performance of dtn routing protocols: A comparative study. **WSEAS Transactions on Communications**, v. 14, p. 121–130, 06 2015.

UFRJ. **DTN - Delay Tolerant Networks**. 2021. Disponível em: https://www.gta.ufrj.br/ensino/eel879/vf/dtn/epidemic.html.

USA, M. **Wildgame Innovations Halo R400 Rangefinder 6x Black**. 2022. Disponível em: https://www.midwayusa.com/product/1008706929.

VENTURA, F. **WhatsApp chega a 99% dos celulares no Brasil; Telegram cresce**. 2020. Disponível em: https://tecnoblog.net/326932/whatsapp-chega-a-99-por-cento-celulares-brasil-telegram-cresce/.

WIKIPEDIA. **Delay Tolerant Networking**. 2021. Disponível em: https://pt.wikipedia.org/wiki/Delay-tolerant_networking.

# APÊNDICE A – Trechos do código fonte

## A.1 Código fonte

**Listagem 1 – Classe MainActivity**

```
1  public class MainActivity {
2      // Atributos da classe
3      NodeInformationObject nodeInformationObject;
4      WiFiDirectBroadcastReceiver mReceiver;
5      ClientThread clientThread;
6      ServerThread serverThread;
7      PipedOutputStream poutTransmitClient, poutReceiveClient;
8      PipedInputStream pinTransmitClient, pinReceiveClient;
9      PipedOutputStream poutTransmitServer, poutReceiveServer;
10     PipedInputStream pinReceiveServer, pinTransmitServer;
11     Timer timerDisplay, timerStoreAndForwardTTL;
12     Handler handlerUpdateUI;
13
14     // Métodos da classe (construtores, getters e setters foram omitidos)
15     void onCreate()
16     void onDestroy()
17     void discover()
18     void connect()
19     void startClientThread()
20     void startServerThread()
21     int receiveMessage()
22 }
```

**Fonte: Autoria própria (2022).**

**Listagem 2 – Classe Cryptography**

```
1  public class Cryptography {
2      // Atributos da classe
3      int DEFAULT_PUB_KEY_LENGTH = 216;
4      int DEFAULT_AES_KEY_LENGTH = 172;
5
6      // Métodos da classe (construtores, getters e setters foram omitidos)
7      String encryptRSA()
8      String decryptRSA()
9      String generateKeyPairString()
10     String getNewAESSecretKey()
11     String encryptAES()
12     String decryptAES()
13 }
```

**Fonte: Autoria própria (2022).**

**Listagem 3 – Classe MessageObject**

```
1  public class MessageObject {
2      // Atributos da classe
3      int DEFAULT_DATE_LENGTH = 13;
4      Integer messageType;
5      byte[] creationDateBuffer;
6      byte[] sourceBuffer;
7      byte[] destinationBuffer;
8      byte[] AESKeyBuffer;
9      int messageBodyLength;
10     byte[] messageBodyBuffer;
11
12     // Métodos da classe (construtores, getters e setters foram omitidos)
13     void sendMessage()
14     MessageObject readMessage()
15 }
```

**Fonte: Autoria própria (2022).**

**Listagem 4 – Classe NodeInformationObject**

```
1  public class NodeInformationObject {
2      // Atributos da classe
3      int STORE_AND_FORWARD_TTL_POLICY = 12 * 60 * 60 * 1000; // 12 horas
4      String thisDeviceAddress;
5      KeyPair myKeyPair;
6      boolean isInitialized;
7      boolean hasServerThread;
8      boolean isServer;
9      boolean isConnected;
10     HashMap<String, String> myContacts;
11     ConcurrentHashMap<Integer, MessageObject> storeAndForwardMessages;
12     List<MessageObject> allReceivedMessages;
13
14     // Métodos da classe (construtores, getters e setters foram omitidos)
15     void resetConnection()
16     boolean hasPreviouslyReceivedThisMessage()
17 }
```

**Fonte: Autoria própria (2022).**

**Listagem 5 – Classe RoutingThread**

```
1  public class RoutingThread extends Thread {
2      // Atributos da classe
3      String myPublicKey;
4      DataOutputStream outUI;
5      ConcurrentHashMap<String, DataOutputStream> mapOutTransmitClient;
6      DataInputStream inUI;
7      ConcurrentHashMap<String, DataOutputStream> mapInReceiveClient;
8
9      // Métodos da classe (construtores, getters e setters foram omitidos)
10     void addClient()
11     void removeClient()
12     void MsgForward()
13     void run()
14 }
```

**Fonte: Autoria própria (2022).**

**Listagem 6 – Classe SocketThread**

```
1  public class SocketThread extends Thread {
2      // Atributos da classe
3      Socket socket;
4      DataOutputStream out;
5      DataInputStream in;
6
7      // Métodos da classe (construtores, getters e setters foram omitidos)
8      void run()
9  }
```

**Fonte: Autoria própria (2022).**

**Listagem 7 – Classe ClientThread**

```
1  public class ClientThread extends Thread {
2      // Atributos da classe
3      NodeInformationObject nodeInformationObject;
4      String hostAddress;
5      DataOutputStream outUI;
6      DataInputStream inUI;
7
8      // Métodos da classe (construtores, getters e setters foram omitidos)
9      MessageObject getServerAcknowledgmentMessage()
10     void getServerStoreAndForwardMessages()
11     void sendClientAckMessage()
12     void sendClientStoreAndForwardMessages()
13     void run()
14 }
```

**Fonte: Autoria própria (2022).**

**Listagem 8 – Classe ServerThread**

```
1  public class ServerThread extends Thread {
2      // Atributos da classe
3      ConcurrentHashMap<String, PipedOutputStream> mapOutTransmitClient,
4          mapOutReceiveClient;
5      ConcurrentHashMap<String, PipedInputStream> mapInReceiveClient,
6          mapInTransmitClient;
7      ConcurrentHashMap<String, String> mapDeviceAddress2PublicKey;
8      RoutingThread routingThread;
9      ConcurrentHashMap<String, SocketThread> mapSocketThread;
10     PipedOutputStream outUI;
11     PipedInputStream inUI;
12
13     // Métodos da classe (construtores, getters e setters foram omitidos)
14     void addNewClient()
15     void removeClient()
16     MessageObject getClientAckMessage()
17     void getClientStoreAndForwardMessages()
18     void sendServerAcknowledgmentMessage()
19     void sendServerStoreAndForwardMessages()
20     void run()
21 }
```

**Fonte: Autoria própria (2022).**

**Listagem 9 – Classe WiFiDirectBroadcastReceiver**

```
1  public class WiFiDirectBroadcastReceiver {
2      // Atributos da classe
3      WifiP2pManager manager;
4      Channel channel;
5      MainActivity activity;
6      ArrayList<WifiP2pDevice> lastConnectedPeers;
7      ArrayList<WifiP2pDevice> nearbyPeers;
8      PeerListListener myPeerListListener;
9      ConnectionInfoListener myConnectionListener;
10
11     // Métodos da classe (construtores, getters e setters foram omitidos)
12     void onReceive()
13 }
```

**Fonte: Autoria própria (2022).**