UNIVERSIDADE FEDERAL FLUMINENSE

LEANDRO LOPES MORETT

LUIZ ANDRE ALVES DE BARROS JUNIOR

APLICATIVO ANDROID PARA TROCA DE MENSAGENS DE TEXTO UTILIZANDO REDE TOLERANTE A ATRASOS VIA BLUETOOTH

Niterói

2016

LEANDRO LOPES MORETT
LUIZ ANDRE ALVES DE BARROS JUNIOR

APLICATIVO ANDROID PARA TROCA DE MENSAGENS DE TEXTO UTILIZANDO REDE TOLERANTE A ATRASOS VIA BLUETOOTH

Trabalho de Conclusão de Curso submetido ao Curso de Tecnologia em Sistemas de Computação da Universidade Federal Fluminense como requisito parcial para obtenção do título de Tecnólogo em Sistemas de Computação.

Orientador:
Marden Braga Pasinato

NITERÓI
2016

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

M845 Morett, Leandro Lopes
Aplicativo android para troca de mensagens de texto utilizando rede tolerante a atrasos via *bluetooth* / Leandro Lopes Morett, Luiz Andre Alves de Barros Junior. – Niterói, RJ : [s.n.], 2016.
49 f.

Projeto Final (Tecnólogo em Sistemas de Computação) – Universidade Federal Fluminense, 2016.
Orientador: Marden Braga Pasinato.

1.Aplicativo Web. 2. Texto de mensagem. 3. Redes tolerantes a atrasos e desconexões. I. Barros Junior, Luiz Andre Alves de. II. Título.

CDD 005.1

LEANDRO LOPES MORETT
LUIZ ANDRE ALVES DE BARROS JUNIOR

APLICATIVO ANDROID PARA TROCA DE MENSAGENS DE TEXTO UTILIZANDO REDE TOLERANTE A ATRASOS VIA BLUETOOTH

Trabalho de Conclusão de Curso submetido ao Curso de Tecnologia em Sistemas de Computação da Universidade Federal Fluminense como requisito parcial para obtenção do título de Tecnólogo em Sistemas de Computação.

Niterói, ___ de _____ de 2016.

Banca Examinadora:

_____

Prof. Marden Braga Pasinato, M. Sc. – Orientador
UFRJ – Universidade Federal do Rio de Janeiro

_____

Prof. Bruno José Dembogurski, D. Sc. – Avaliador
UFRRJ - Universidade Federal Rural do Rio de Janeiro

Dedicamos este trabalho as nossas famílias que nos deram a força e o apoio necessário para concluirmos esta nossa jornada.

AGRADECIMENTOS

Agradeço em primeiro lugar a Deus que iluminou o meu caminho durante esta caminhada, agradecemos aos nossos professores e aos nossos colegas que nos ajudaram na conclusão deste trabalho.

“Que os vossos esforços desafiem as impossibilidades, lembrai-vos de que as grandes coisas do homem foram conquistadas do que parecia impossível.”

Charles Chaplin

RESUMO

O presente trabalho pretende desenvolver um aplicativo que provê comunicação entre *smartphones* através de mensagens de texto utilizando o *bluetooth*, onde os aparelhos se comportam como os nós de uma Rede Tolerante a Atrasos e Desconexões (*Delay and Disruption Tolerant Neworks* - DTN). Guardando as mensagens e repassando para seus vizinhos, beneficiando-se desta forma da mobilidade para disseminar as mensagens e almejando atingir, dentro de um período determinado, o objetivo principal que é fazer com que a mensagem chegue no destinatário.

Palavras-chaves: DTN, *bluetooth* e comunicação.

ABSTRACT

This study aims to develop an application that provides communication between smartphones via text messages using bluetooth, where the devices behave like the nodes of a Delay and Disruption Tolerant Networks (DTN). Holding posts and passing on to their neighbors, benefiting from mobility to disseminate messages with the aim to achieve, within a certain period, the main objective that is to deliver the message to the receiver.

Key words: DTN, *bluetooth* and communication.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

DTN – *Delay and Disruption Tolerant Networks*

SMS – *Short Message Service*

ISM - *Indutrial, Scientifc, Medical*

app – applications

MAC – *Media Access Control*

ART - *Android Runtime*

HAL - camada de abstração de *hardware*

ACK - *Acknowledgement*

AOT - *Ahead-of-time*

JIT - compilação *just-in-time*

IDE – *Integrated Development Environment*

SUMÁRIO

1 INTRODUÇÃO

O objetivo deste trabalho é desenvolver um aplicativo que proporcione um meio alternativo de comunicação, que não dependa de equipamentos sofisticados ou empresas de telecomunicações, onde os próprios usuários que se beneficiam da comunicação também colaboram com ela.

Em situações adversas onde as estruturas convencionais de comunicação estejam danificadas, seja por catástrofes naturais, como furacões ou terremotos; seja por ações humanas, como guerras, ou até mesmo em locais onde não existam tais estruturas, este trabalho pode ser considerado como um importante meio de comunicação e não apenas um meio alternativo, pois a mobilidade dos nós pode tornar possível a comunicação em qualquer meio que possa propagar uma onda eletromagnética utilizada no *bluetooth*.

A lógica do aplicativo foi inspirada no roteamento epidêmico das redes DTN (*Delay and Disruption Tolerant Neworks* - Redes Tolerantes a Atrasos e Desconexões) que mantém cópias das mensagens nos nós e procura "infectar" outros nós que mantiverem conexão.

Para a codificação foi utilizada a linguagem JAVA juntamente com a IDE (*Integrated Development Environment* – Ambiente de Desenvolvimento Integrado) eclipse com o plugin de desenvolvimento para *Android* instalados em um computador *desktop*.

Para a persistência dos dados, foram utilizados apenas arquivos, o que torna o aplicativo compatível com todas as versões do *Android*, porém, toda a responsabilidade em manter os dados íntegros fica por conta da aplicação.

O aplicativo não trata a possível perda ou corrompimento do arquivo que guarda as mensagens. Ao invés disso, ele tira proveito da grande redundância presente nas cópias das mensagens em vários smartphones para tentar recompor as mensagens de um aparelho que tenha corrompido ou perdido seu arquivo de mensagens.

Este trabalho está dividido da seguinte forma:

- No capítulo 2, será apresentado o conceito de redes DTN, explicando como funcionam e seus principais protocolos;
- No capitulo 3, serão são apresentadas as tecnologias utilizadas, ressaltando o *bluetooth* e o sistema *Android* para smartphones;
- No capitulo 4, será apresentado a Arquitetura do sistema, com seu diagrama de classes e casos de uso, e a lógica que norteia a aplicação;
- No capitulo 5, mostraremos o funcionamento do sistema de arquivos e as telas do aplicativo bem como as funcionalidades oferecidas por cada tela de interação com o usuário.

2 REDES DTN

2.1 O QUE SÃO REDES DTN

As Redes Tolerantes a Atrasos e Desconexões (*Delay and Disruption Tolerant Neworks* - DTN) foram originalmente desenvolvidas para uso na comunicação inter-planetária com a proposta de superar o desafio da latência (tempo que a informação leva para chegar ao seu destino) e a ocorrência de janelas de comunicação, onde os pares podem se comunicar por um determinado período de tempo, geralmente sendo obstruídos por outros corpos celestes [1].

Zonas de guerras ou de desastres naturais, locais de acidentes e comunicação submarina também são cenários onde a comunicação convencional não flui como o esperado, pois a infraestrutura esta comprometida ou inexistente [2]. Sendo assim, as redes DTN podem ser soluções para os inconvenientes das redes convencionais, possuindo as seguintes características:

- Conectividade Intermitente: é normal a ausência de conectividade em redes, principalmente nas redes sem fio, onde nem sempre existe um caminho fim a fim entre o nó fonte e o nó destino;
- Atrasos na entrega: o congestionamento da rede pode ocasionar atrasos na entrega da mensagem, e se o atraso for muito grande, a mensagem pode ser perdida. Dependendo da configuração, a rede DTN poderá tolerar atrasos de horas ou até mesmo de dias, até que a mensagem possa ser encaminhada ao seu destino.
- Erros de transmissão: na rede convencional, quando ocorre erro de transmissão pode ser necessário reenviar o pacote inteiro novamente, ocasionando mais processamento e mais congestionamento no tráfego da rede. Na DTN

esse processo é menos custoso, pois são necessárias menos retransmissões, uma vez que o pacote esta armazenado nos nós da rede.

2.2 COMO FUNCIONAM AS REDES DTN

2.2.1 Princípio de funcionamento (*Store-and-forward)*

As redes DTN funcionam através da comutação de mensagens. Tais mensagens passam por vários nós onde são armazenadas de maneira não volátil (armazenamento persistente) e retransmitidas para outros nós da rede, até que chegue ao seu destino. Sendo assim, não existe um caminho pré-determinado para o "trânsito" da mensagem. O nó recebe a mensagem e a armazena, após isso, reenvia aos seus nós vizinhos, que podem ser o nó destino ou não. Desta maneira, não se torna necessário que o nó de destino esteja ativado no momento em que o nó de origem envia a mensagem, uma vez que os nós intermediários da rede possuem a mensagem armazenada e podem entrega-la posteriormente. O processo descrito acima é chamado de *store-and-forward* (armazena-e-encaminha) e está ilustrado na figura 1 [3].

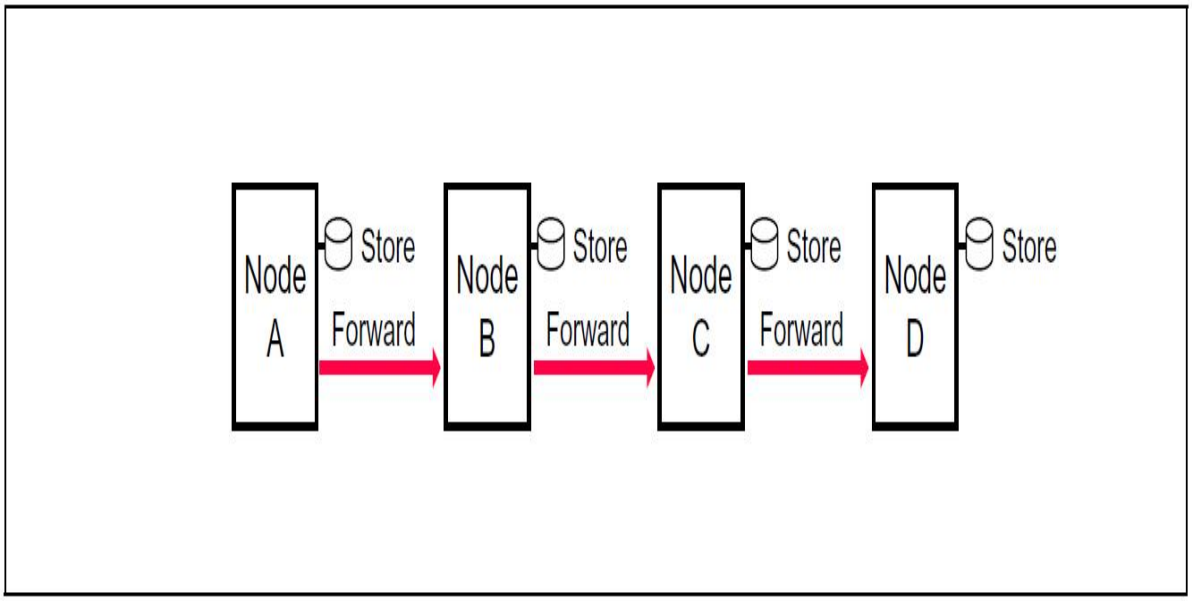

Figura 1. Processo Armazena-e-Encaminha (*Store-and-Forward*) [3]

2.2.2 Camada de Agregação

A arquitetura DTN prevê a utilização da técnica de comutação de mensagens (*store-and-forward*) e o armazenamento persistente dos dados definindo uma rede sobreposta (*Overlay Network*). Esta nova camada é denominada camada de agregação (*Bundle Layer*). Ela está disposta entre a camada de aplicação e a camada de transporte, com o beneficio de poder reutilizar a infraestrutura existente para transmitir cada segmento entre os nós intermediários da comunicação.

Para garantir interoperabilidade com qualquer tipo de rede, esta camada se situa acima da camada transporte das redes que se servem do perfil de protocolos TCP/IP. Como ilustrado na Figura 2, as camadas abaixo da camada de agregação são definidas de acordo com a conveniência do ambiente de comunicação de cada região, podendo ser específicas para cada região englobada pela DTN [4].

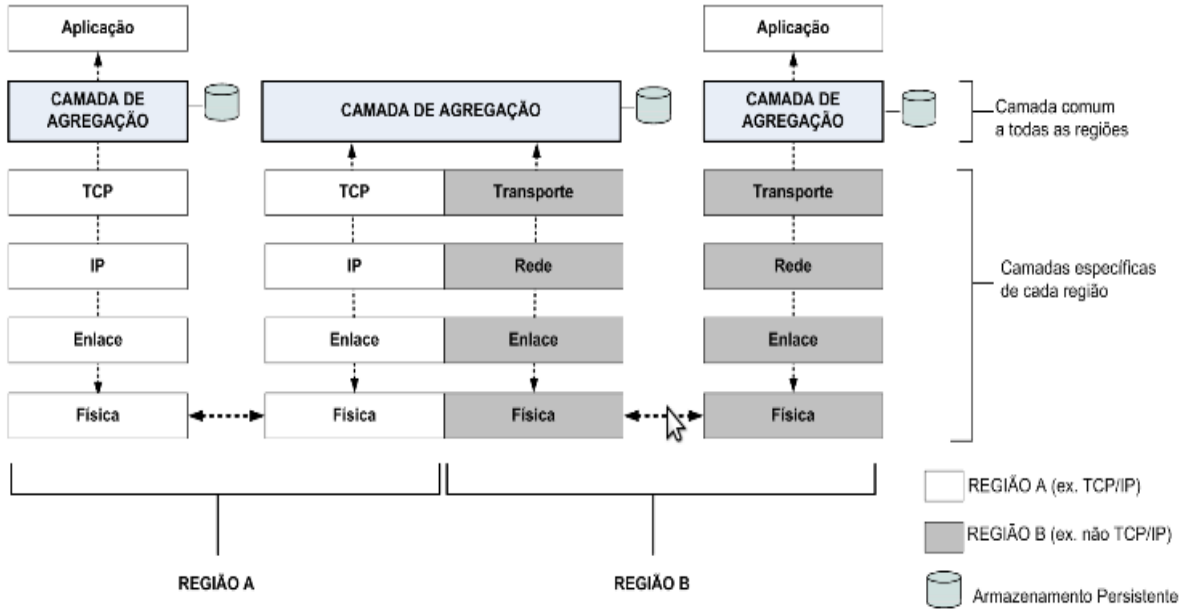

Figura 2. Camada de Agregação [4]

As aplicações das DTNs enviam mensagens de tamanhos variáveis chamadas de unidades de dados da aplicação *(Application Data Units* - ADUs). As mensagens são transformadas pela camada de agregação em uma ou mais unidades de dados de protocolo (*Protocol Data Units* - PDUs) denominadas agregados (*bundles*), que são armazenados e encaminhados pelos nós DTN. Na solicitação de transferência de um arquivo, este pode conter dados como *login* e senha, nome do arquivo solicitado e para onde este deve ser enviado. Estes dados são transformados em *bundles* e enviados de uma única vez para evitar as inúmeras trocas de mensagens, como as que são feitas numa rede TCP/IP padrão. Múltiplas cópias do mesmo agregado podem existir simultaneamente em diferentes partes da rede, tanto na memória local de um ou em mais nós DTN quanto em trânsito entre os nós.

Pode haver situações em que o agregado terá que ser fragmentado devido a restrições da rede regional em que esteja passando. Neste caso, funções de fragmentação e reagrupamento do agregado são executadas pelo protocolo de agregação. Um agregado pode ser fragmentado diversas vezes e cada um destes serão tratados como um novo agregado. De forma análoga, dois ou mais fragmentos reagrupados serão tratados também como um novo agregado.

2.2.3 A Transferência de Custódia

Como falado anteriormente, as redes DTN fazem uso do protocolo de agregação e dos protocolos que operam nas camadas abaixo da camada de agregação para a retransmissão nó a nó em caso de perdas ou dados corrompidos. Porém, como os protocolos abaixo da camada de agregação não são executados de modo fim-a-fim na DTN, estes protocolos só podem ser implementados na própria camada de agregação [3].

Para efetuar esta retransmissão nó a nó a camada de agregação faz uso do método denominado Transferência de Custódia. Ela representa o compromisso que nó intermediário tem de reter os dados em si, até estes sejam transferidos para o nó seguinte, iniciando na origem e sendo completada no destino [5]. A camada de agregação utiliza um temporizador (*time-to-live*) e retransmissões para elaborar um método de reconhecimento. Na Figura 3, vemos que quando um nó DTN (nó fonte) encaminha um agregado para o nó intermediário, ele solicita a transferência de custódia e inicia o temporizador de retransmissão. Caso a camada de agregação do próximo nó aceitar a custódia, um sinal de reconhecimento (ACK) para o nó fonte.

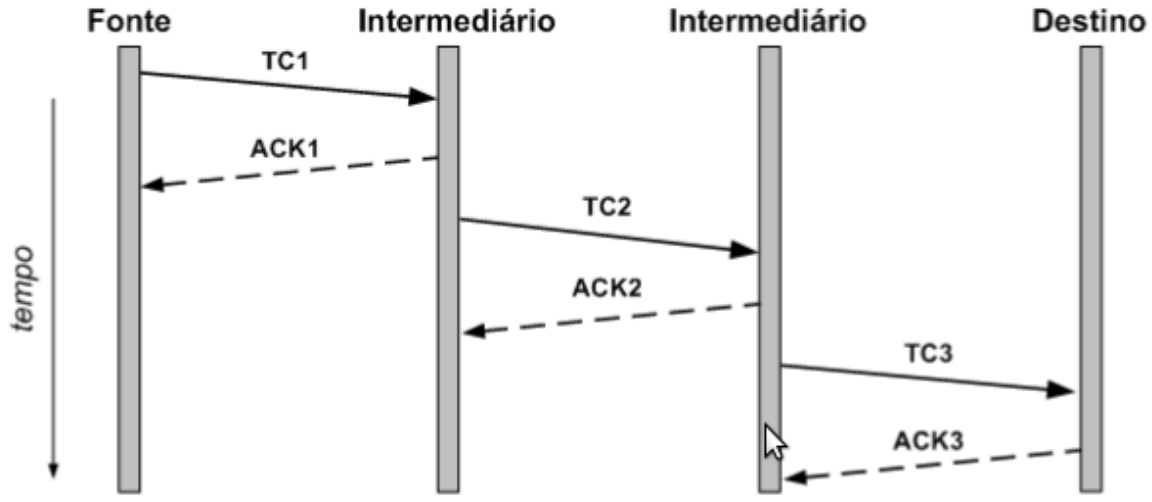

Figura 3. Transferência de Custódia [5]

Contudo, se não houver um ACK de reconhecimento até que o temporizador complete seu ciclo, o nó fonte reenvia o agregado. O temporizador de retransmissão pode ter seu valor calculado baseado em experiências passadas ou

apenas ser distribuído entre os nós junto com a informação de roteamento. Os nós que aceitam a transferência de custódia são denominados custódios.

A arquitetura DTN não exige que todos os nós DTN aceitem a transferência de custódia [6]. Por isso, não pode ser considerado um mecanismo salto-a-salto legítimo. Se uma transferência de custódia por qualquer motivo não for aceita, o temporizador e o mecanismo de retransmissão não serão aplicados, de maneira que outros protocolos deverão ser empregados para que a entrega das mensagens seja bem sucedida. Cada nó DTN é capaz de optar se aceita a custódia de dados ou não. Para isso ele pode se basear, no tamanho, na prioridade, em políticas de segurança, no roteamento, ou no tempo máximo de vida da mensagem.

O acesso ao armazenamento em cada nó é um dos recursos mais disputados em DTN. À medida que nas redes convencionais as mensagens são simplesmente descartadas quando a memória acaba, em DTN isto não ocorre caso a custódia tenha sido aceita. Existem apenas dois caos em que o custódio apaga um agregado: primeira, o agregado é transferido para outro nó custódiante; segunda, na hipótese do ciclo de vida do agregado expirar.

2.2.4 O Roteamento Epidêmico

O roteamento epidêmico é considerado ideal para redes com conectividade intermitente e sucessivas desconexões [7]. Este roteamento é classificado como estocástico, isto é, ele garante a entrega de mensagens aos seus destinatários com conhecimento mínimo da topologia de rede. Este roteamento é eficiente e garante a entrega de mensagens mesmo não havendo um caminho totalmente conectado entre a origem e o destino.

O roteamento epidêmico pressupõe que um nó fonte desconhece a localização do nó destino e nem mesmo sabe qual a melhor rota para alcançá-lo. A ideia é que a mobilidade dos nós na rede possibilite que eles entrem no alcance de transmissão uns dos outros periodicamente e, o mais importante, de maneira aleatória.

No roteamento epidêmico, a mobilidade dos nós é uma condição facilitadora para a entrega de mensagens.

Para garantir a entrega de mensagens, é necessária uma conectividade periódica par-a-par. No momento que dois nós estabelecem um contato, listas com informações que identificam as mensagens armazenadas em cada nó são trocadas. Com esta troca, o nó estipula quais das mensagens armazenadas do nó vizinho ele ainda não detêm. Após a identificação das mensagens, há uma solicitação de copias das mensagens que ainda não possui. Toda vez que um nó estabelece contato com um novo nó vizinho, este processo de troca é repetido, facilitando a rápida disseminação das mensagens pelos nós conectados a rede. Assim sendo, a probabilidade da mensagem chegar ao seu destino será maior, pois, haverá várias copias de uma mesma mensagem espalhadas pela rede e, consequentemente, menor será o atraso.

Na Figura 4 podemos ver o roteamento epidêmico numa rede sem fio. Nela, os *notebooks* representam os nós móveis da rede sem fios, seus alcances são representados pelos círculos pontilhados e as setas ilustram o deslocamento dos nós. Na Figura 4 (a), o nó origem F deseja transmitir uma mensagem para o nó de destino D que não se encontra dentro de seu alcance. Prontamente, o nó F começa a disseminar sua mensagem para todos os nós que entram em seu alcance. Na Figura 4 (b) podemos ver na cor cinza os nós contaminados por F. Na Figura 4 (c) a contaminação continua não só pelo nó origem F, mas também pelos nós intermediários da rede. Na Figura 4 (d), vemos quando o nó de destino D é alcançado por um nó contaminado da rede, o qual lhe transmite a mensagem do nó origem F.

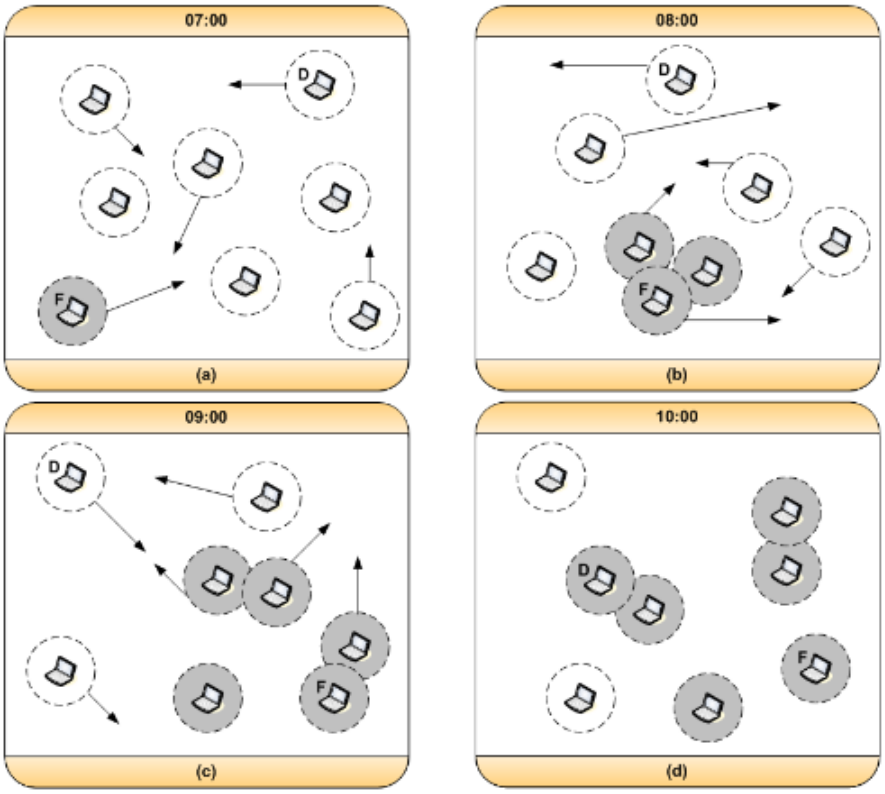

Figura 4. Roteamento Epidêmico [7]

3 TECNOLOGIAS UTILIZADAS

3.1 BLUETOOTH

O *bluetooth* é um padrão mundial de comunicação sem fio que permite transmissão de dados e arquivos de maneira rápida e segura através de aparelhos de telefone celular, notebooks, câmeras digitais, consoles de videogame digitais, impressoras, teclados, mouses e até fones de ouvido, entre outros equipamentos.

O sistema utiliza uma faixa de frequência de rádio de onda que opera entre 2.4 GHz e 2.5GHz, chamada *ISM* (*Indutrial, Scientifc, Medical)*, para criar uma comunicação entre aparelhos habilitados. A ISM é uma faixa de frequência aberta, portanto, pode ser utilizada por qualquer sistema de comunicação. Seu alcance é curto e só permite a comunicação entre dispositivos próximos e sua principal vantagem é seu baixo consumo de energia. Quando estão no raio de alcance, os equipamentos podem ser encontrados, mesmo estando em ambientes diferentes, porém esse alcance varia em relação a potência [8]:

- Classe 1: alcance de 100 metros / potência máxima de 100 miliwatts.
- Classe 2: alcance de 10 metros / potência máxima de 2,5 miliwatts.
- Classe 3: alcance de 1 metro / potência máxima de 1 miliwatt.

Nas primeiras versões, as taxas de transmissão no *bluetooth* variavam entre 721Kbps a 3Mbps. Mesmo tendo baixa velocidade de transmissão, as conexões eram eficientes na maioria dos dispositivos. Na versão 3.0 lançada em 2009, a taxa de transmissão evoluiu para 26 Mbps e também houve uma melhora no consumo de energia. Contudo, no fim de 2009 foi lançada a versão 4.0, que é a versão ainda utilizada atualmente, e sua principal característica foi o uso mais eficiente no consumo da bateria [9].

3.1.1 Redes Bluetooth

Quando uma conexão *bluetooth* entre dois ou mais dispositivos é realizada, dizemos que eles formam uma rede *Piconet*. Nela, um dos dispositivos é eleito mestre o qual tem a responsabilidade de regular a transmissão de dados na rede e o sincronismo entre os dispositivos.

Sendo a faixa de frequência ISM uma faixa aberta e não exclusiva para a rede *bluetooth*, é necessário garantir que não haja interferências de outros sistemas de comunicação. Isto é obtido através de uma sequência de saltos de frequência usando o relógio e do endereço MAC do mestre.

Os demais dispositivos conectados ao mestre, são denominados escravos. Estes sabem o endereço MAC do mestre e sabem o atraso dos seus próprios relógios em relação ao mestre, e assim podem sintonizar seus rádios na mesma sequência. Em cada rede *Piconet* o limite de dispositivos conectados é 8 (1 mestre e 7 escravos), no entanto, é possível aumentar este número.

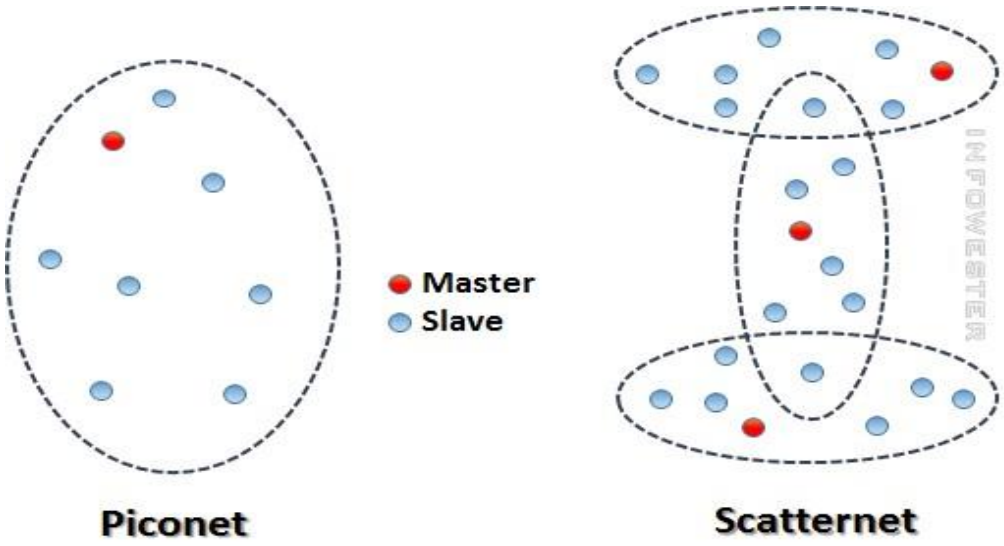

Figura 5. Piconet e Scatternet [12]

Na figura 5 podemos ver que um mesmo dispositivo pode participar de mais de uma *Piconet*, desde que na condição de escravo. Como cada *Piconet* salta as frequências numa sequência diferente, um escravo de duas *Piconet*s é forçado a dividir seu tempo entre elas. Em outras palavras, este procedimento consiste em fazer com que uma *Piconet* se comunique com outra que esteja dentro do limite de alcance, esquema este denominado *Scatternet* [10].

3.2 ANDROID

Trataremos aqui as características do sistema *Android* contemporâneo à elaboração deste trabalho, a saber, segundo semestre do ano de 2016.

O *Android* é um sistema operacional baseado no *kernel* do sistema operacional *Linux*, porém, tem pouca semelhança com distribuições *Linux* tradicionais presentes em *desktops* como *Ubuntu* e *Debian*. O *Android* “habita” principalmente em celulares, mas atualmente existem até relógios que “rodam” o *Android*. A figura 6 mostra os principais componentes da plataforma *Android*.

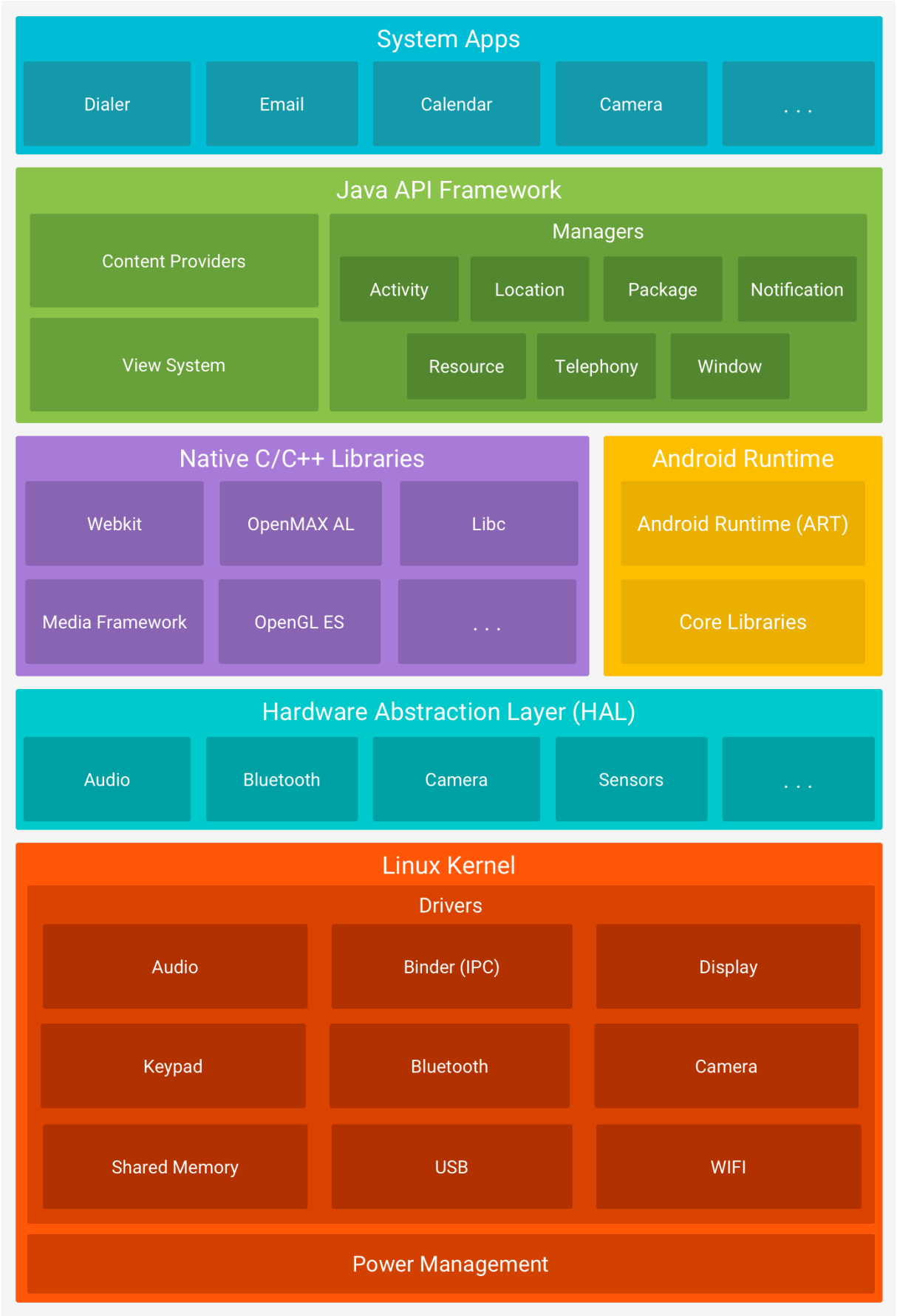

Figura 6. Componentes da Plataforma *Android* [11]

3.2.1 *Kernel Linux*

A base da plataforma *Android* é o *kernel Linux*. Por exemplo, o *Android Runtime* (ART) baseia-se no *kernel* do *Linux* para funcionalidades subjacentes tais como *threading* e gerenciamento de memória de baixo nível. Usando um *kernel Linux* o *Android* pode tirar proveito dos principais recursos de segurança e permite que os fabricantes de dispositivos desenvolvam *drivers* de *hardware* para um *kernel* bem conhecido.

3.2.2 Camada de Abstração de *Hardware* (*Hardware Abstraction Layer)*

A camada de abstração de *hardware* (HAL) fornece interfaces padrão que expõem os recursos de *hardware* do dispositivo para o nível mais alto – *Java* API *Framework*. O HAL é composto por vários módulos de biblioteca, cada qual implementa uma interface para um tipo específico de componente de hardware, como a câmera ou módulo *bluetooth* . Quando uma *Java* API *Framework* faz uma chamada para acessar o *hardware* de um dispositivo, o sistema *Android* carrega o módulo de biblioteca para esse componente de hardware.

3.2.3 *Android Runtime*

Para dispositivos que executam o *Android* versão 5.0 (API nível 21) ou superior, cada aplicativo é executado em seu próprio processo e com a sua própria instância do *Android Runtime* (ART). ART é escrito para executar várias máquinas virtuais em dispositivos de pouca memória ao executar arquivos DEX, um formato de *bytecode* projetado especialmente para *Android* otimizado para o mínimo consumo de memória. Outro conjunto de ferramentas importante é o *Jack*, que compila fonte

Java em *bytecode* dex *Android*. *Jack* substitui o conjunto de ferramentas Android anterior, que consiste em várias ferramentas, como *javac, ProGuard, JarJar* e *dx.*

Algumas das principais características do ART incluem o seguinte:

- *Ahead-of-time* (AOT) e (JIT) compilação *just-in-time*
- Coleta de lixo otimizado (GC-*garbage collector*-)
- Melhor suporte a depuração, incluindo um *profiler* de amostragem dedicado, exceções de diagnóstico detalhados e relatórios de falhas, e a capacidade de definir pontos de controle para monitorar áreas específicas.

Antes do *Android* versão 5.0 (API nível 21), *Dalvik* foi o *Runtime* do *Android*. Se um aplicativo funciona bem em ART, em seguida, ele deve funcionar em *Dalvik* também, mas o inverso não pode ser verdade.

Android também inclui um conjunto de bibliotecas de tempo de execução do núcleo que fornecem a maior parte da funcionalidade da linguagem de programação Java, incluindo algumas características da linguagem *Java* 8 , que o *Java API framework* usa.

3.2.4 Bibliotecas nativas C/C++

Muitos componentes do sistema *Android* do núcleo e serviços, tais como ART e HAL, são construídos a partir de código nativo que necessitam de bibliotecas nativas escritas em C e C ++. A plataforma Android fornece APIs *framework Java* para expor a funcionalidade de algumas dessas bibliotecas nativas para apps. Por exemplo, é possível acessar *OpenGL* ES através do *framework Android* Java *OpenGL* API para adicionar suporte para o desenho e manipulação de gráficos 2D e 3D em um aplicativo .

Se o desenvolvimento de um aplicativo requerer código C ou C ++, é possível utilizar um conjunto de ferramentas (*Android NDK*) que permite implementar partes deste aplicativo usando linguagens de código nativo, e assim acessar algumas dessas bibliotecas nativas da plataforma diretamente do código nativo.

3.2.5 *Java API Framework*

Todo o conjunto de recursos do sistema operacional *Android* está disponível para os desenvolvedores através de *APIs* escritas na linguagem Java. Essas *APIs* formam os blocos de construção para criar aplicações Android, simplificando a reutilização de núcleo, componentes e serviços do sistema modular, que incluem o seguinte:

- Um rico e extensível *View System* que pode ser usado para construir UI de um aplicativo, incluindo listas, grades, caixas de texto, botões, e até mesmo um navegador web embutido.
- Um Gerenciador de Recursos, fornecendo acesso a recursos não código como cadeias localizadas, gráficos e arquivos de layout;
- Um Gestor de notificação que permite que todos os aplicativos possam exibir alertas personalizados na barra de status;
- Um Gerenciador de atividade que gerencia o ciclo de vida de aplicativos e fornece uma comum navegação volta de pilha;
- Provedores de conteúdo que permitem que aplicativos acessem dados de outros aplicativos, como o aplicativo de contatos, ou compartilhem seus próprios dados.

Os desenvolvedores têm acesso completo às mesmas estruturas das *APIs* que os aplicativos do sistema *Android* usam.

3.2.6 Sistema de aplicativos

Android vem com um conjunto de aplicativos essenciais para *e-mail*, mensagens SMS, calendários, navegação na internet, contatos e muito mais. Aplicativos incluídos com a plataforma não têm nenhum status especial entre os aplicativos que o usuário optar por instalar. Assim, um aplicativo de terceiros pode tornar-se o navegador do usuário da web padrão, mensageiro *SMS*, ou mesmo o teclado padrão.

Os aplicativos do sistema funcionam tanto como aplicativos para os usuários e para fornecer capacidades-chave que os desenvolvedores possam acessar a partir de seu próprio aplicativo. Por exemplo, se um aplicativo quisesse entregar uma mensagem SMS, não seria necessário desenvolver essa funcionalidade, em vez disso pode-se invocar qualquer *app SMS* já está instalado para entregar uma mensagem para o destinatário especificado.

4 ARQUITETURA

4.1 DIAGRAMA DE CASOS DE USO

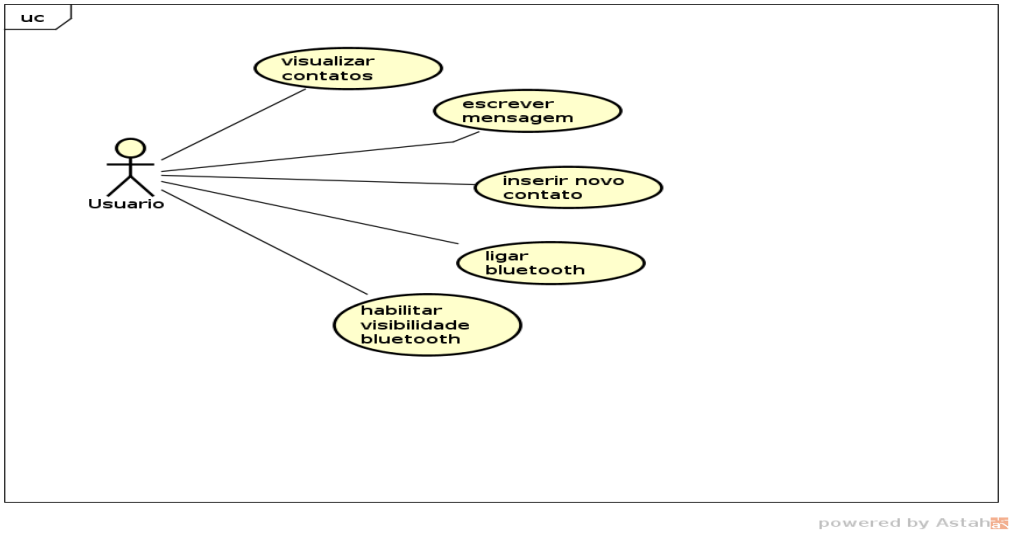

Figura 7 diagramA de casos de uso

4.2 DIAGRAMA DE CLASSES

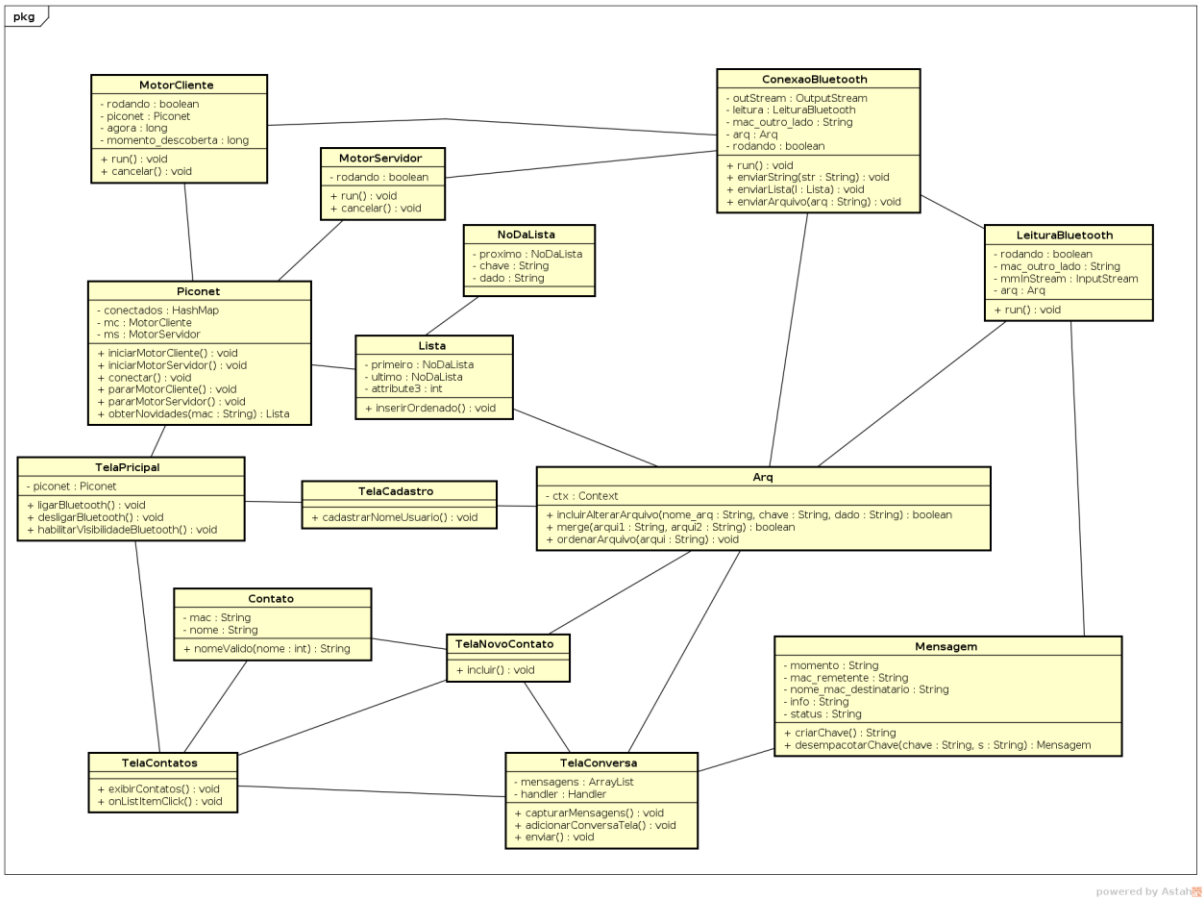

Figura 8 diagrama de classes

4.3 ARQUIVOS

Todos os dados da aplicação ficam guardados em apenas cinco arquivos: *pri.txt, ctt.txt, msg.txt, nuv.txt* e *udt.txt*. Os arquivos *pri.txt* e *udt.txt* guardam uma única linha cada um. Todos os outros arquivos podem guardar muitos dados, e têm características em comum: uma linha é chave e a próxima é dado; as chaves não podem se repetir mas os dados sim ; as chaves são inseridas ordenadamente.

No código fonte, temos a classe *Arq* que manipula os arquivos. Alguns métodos desta classe são essenciais à aplicação. Um deles é o método *merge()* que recebe dois arquivos ordenados e produz um único arquivo também ordenado e com chaves exclusivas. Outro método importante é o método *incluirAlterarArquivo()* que introduz uma nova chave e um novo dado ou altera o dado de uma chave já existente de maneira ordenada, ou seja após a inclusão (se for o caso) o arquivo continua ordenado.

4.3.1 Arquivo *pri.txt* e arquivo *udt.txt*

Estes arquivos são editados apenas na primeira execução do programa. É nele que a aplicação guarda o nome de usuário. Nas execuções seguintes, o aplicativo obtém o nome de usuário acessando o conteúdo deste arquivo.

4.3.2 Arquivo *ctt.txt*

Neste arquivo estão armazenados todos os contatos. Os dados deste arquivo são os nomes de usuários de cada contato, já as chaves são preferencialmente o endereço MAC de cada contato, mas num primeiro momento não se tem esta informação ainda, e é usado como chave também o nome.

Para a aplicação seria mais confortável assumir que os nomes de usuário realmente seriam únicos (pois deveriam ser) e assim seriam chaves e já até estariam ordenados em ordem alfabética para exibição. Mas logo se percebe que dois usuários distintos poderiam, por exemplo, cadastrar o nome de usuário “joao paulo” e ambos enviarem mensagens para “manoel joaquim”. Neste cenário, a aplicação no aparelho de “manoel joaquim” não seria capaz de diferenciar os remetentes da mensagem. Por esse motivo foi usado o MAC do hardware *bluetooth* como chave e o nome de usuário como dado.

Por simplicidade, não trataremos neste trabalho a possibilidade deste MAC ser alterado e assumiremos que os endereços MAC realmente serão únicos. Não seria confortável para o usuário do aplicativo inserir o MAC do contato toda vez que quisesse incluir um novo contato, por isso, será inserido apenas o nome de usuário do contato e o aplicativo salvará no arquivo *ctt.txt* o nome como chave e também como dado. A obtenção do MAC não será responsabilidade do usuário, e sim da aplicação. Logo, no exemplo acima, o *app* no aparelho de “manoel joaquim” , assim que chegassem as mensagens dos dois usuários com o nome “joao paulo” , salvaria no arquivo *ctt.txt* os MAC deles - que já estariam presentes na mensagem – e seus nomes repetidos semelhante ao exposto abaixo:

FF:FF:FF:FF:AA
joao paulo
FF:FF:FF:FF:BB
joao paulo

4.3.3 Arquivo *msg.txt* e arquivos *xxxnuv.txt*

Estes arquivos são os mais importantes da aplicação e são usados em conjunto. O arquivo *msg.txt* guarda, em teoria, todas as mensagens de todos os usuários. Isso só ocorreria se o grafo de conectividade de todos os nós (aparelhos celulares com o aplicativo instalado) fosse conexo e houvesse espaço de armazenamento suficiente em um aparelho. Talvez, na prática, isso nunca venha acontecer, (principalmente se houverem muitos usuários), mas, o fato é que, o arquivo *msg.txt* guardará as mensagens que forem uma conversa do próprio usuário e também as conversas entre outros usuários. Obviamente, que conversas entre “joao paulo” e “manoel joaquim” não serão exibidas no aparelho de “ana maria” , mas o celular de “ana maria”, se tiver conexão com um deles, passará a guardar no arquivo *msg.txt* as mensagens deles. Um arquivo com sufixo *'nuv.txt'* guarda temporariamente as mensagens que acabam de chegar de um *SocketBluetooth*.

Na verdade, o nome *xxxnuv.txt* representa oito arquivos, todos com o sufixo *'nuv.txt'*, porém o prefixo *xxx* é substituído pelos nomes de cada conexão. Como, na aplicação podem ocorrer até oito conexões ao mesmo tempo, todas elas podem estar a escrever o seu próprio arquivo *'nuv.txt'* ao mesmo tempo, mas apenas uma de cada vez pode usar o método *merge()* para alterar o arquivo *'msg.txt'* , enriquecendo-o com novas mensagens contidas em *'xxxnuv.txt'* .

5 ALGORÍTMO E TELAS

A lógica da aplicação está toda voltada para a troca de mensagens via *bluetooth*, guardando as mensagens em arquivos e não tabela de banco de dados, no intuito de fornecer a mesma mensagem para outras conexões futuras que poderão servir de ponte, visando enviar uma mensagem de um remetente *A* para um destinatário *B*, mesmo que *B* não esteja ao alcance (conexão *bluetooth*) de *A*. Obviamente, os destinatários (contatos) também serão salvos em arquivos.

Para que a aplicação não fique presa em métodos bloqueantes foi necessária a utilização de *threads* que rodam aparentemente em paralelo. Uma *thread MotorCliente* fica responsável por fazer descobertas de dispositivos na vizinhança e requisitar conexão com os dispositivos encontrados. Outra *thread MotorServidor* fica à espera de conexões e para chamada do método *accept()* até que algum celular na vizinhança com o aplicativo instalado requisite conexão.

Uma vez estabelecida a conexão, a troca de mensagens entre os dois celulares também deverá ocorrer em novas *threads ConexaoBluetooth* e *LeituraBluetooth* em paralelo, pois a leitura de dados de um socket *bluetooth* também paralisa a execução (chamada ao método *read()*) até que realmente cheguem dados. A escrita (envio de dados no *socket*), por sua vez, também ficará em um *loop*, monitorando um *HashMap* contendo possíveis novidades (mensagens que o aparelho conectado ainda não tem) a serem enviadas ao aparelho do outro lado da conexão. Essas novidades podem ser mensagens que o próprio usuário digita ou mensagens recebidas por outras *threads* que gerenciam conexões com outros aparelhos.

Identificar uma novidade para um aparelho conectado nem sempre é uma tarefa simples. Suponhamos uma situação em que o aplicativo está rodando em um aparelho *A* e este tem conexão *bluetooth* com três aparelhos (*B,C* e *D*). Neste cenário, existem no aparelho *A* três *threads ConexaoBluetooth* e três *threads LeituraBluetooth* rodando. Para um melhor entendimento, vamos dar nomes intuitivos para essas *threads* com os seguintes nomes: *envia_para_B*, *envia_para_C*, *envia_para_D*, *recebe_de_B*, *recebe_de_C* e *recebe_de_D*.

Quando é iniciada a *thread envia_para_B* (cada *thread* de envio tem seu próprio *HashMap* de possíveis novidades) ela não sabia das mensagens que o aparelho *B* já tinha, então envia todo o seu arquivo *'msg.txt'* para ele e avisa previamente com uma mensagem de controle que está enviando todo o arquivo e não apenas parte dele. Neste envio pode ter muita redundância, ou seja, podem ser enviadas muitas mensagens que *B* já tinha em seu arquivo *'msg.txt'*. Mas após o envio, é seguro considerar que *A* não tem mais novidades neste momento para *B*.

Quando a *thread recebe_de_B* recebe o arquivo *'msg.txt'* completo de *B*, ela executa o método *merge()*, e dentro deste método, captura todas as mensagens que *A* não tinha, e neste momento *A* considera que estas mensagens são também novidades para *C* e *D* , e a *thread recebe_de_B* insere estas mensagens também nos *HashMaps* de novidades das *threads envia_para_C* e *envia_para_D* . Ainda no método *merge()*. *A* também captura as suas próprias mensagens das quais *B* não tinha e considera novidades apenas para *B*, e logo em seguida, a t*hread recebe_de_B* insere estas mensagens no *HashMap* de novidades da *thread envia_para_B*, porém, se o arquivo recebido não fosse completo, *A* não poderia inferir novidades para *B*.

Complementando o descrito acima, *A* também considerará uma novidade para *C* e *D* quando receber de *B* mensagens com status 1 (destinatário recebeu) e *A* tivesse estas mensagens com status 0 (destinatário não recebeu). Por outro lado, as mensagens que *A* tem com status 1 e *B* com status 0, *A* considera novidades para *B*, isso caso o recebimento do arquivo *'msg.txt'* vier sinalizado como completo.

Um caso no qual se identifica uma novidade mais facilmente, é quando o usuário, no aparelho *A*, digita uma mensagem na tela de conversa para um contato qualquer. Neste caso, a mensagem é considerada uma novidade para todos os conectados, ou seja, para *B, C* e *D*, e logo em seguida a própria *thread* da tela de

conversa insere as mensagens nos *HashMaps* de novidades das *threads* *envia_para_B, envia_para_C* e *envia_para_D*.

O aplicativo visa espalhar a mensagem de maneira semelhante à proliferação de um vírus biológico, pois um aparelho infecta outros aparelhos com as mensagens que tem, e os outros aparelhos fazem o mesmo, na intenção de disseminar a mensagem para que esta alcance o seu destinatário. Neste processo, a mensagem carrega consigo, o MAC do remetente e o MAC ou nome de usuário do destinatário para que seja possível vir à tona no aparelho do destinatário como conversa com um de seus contatos.

5.1 NOME DE USUÁRIO

Em um primeiro momento, assim que o usuário instala o aplicativo, será exibida uma tela com um campo de entrada de texto – onde o usuário fornecerá o seu “nome de usuário” para a aplicação – e um botão com a descrição ”cadastrar”.

Não será possível utilizar o aplicativo sem que se cumpra esta etapa de cadastro de nome de usuário. Após o clique no botão ”*cadastrar*”, o nome inserido na caixa de texto será submetido à uma validação. Tal validação não permitirá nomes começando nem terminando com o caractere “espaço em branco”, também não será permitido dois caracteres “espaço em branco” juntos, e letras maiúsculas e números não são permitidos no nome de usuário. Outra peculiaridade no nome, é que deve haver pelo menos um caractere “espaço em branco” no meio do nome, na tentativa de incentivar o usuário a colocar o seu nome e seu sobrenome em caixa baixa.

Como dito anteriormente, só será possível utilizar o aplicativo após passar por esta validação de nome de usuário. Após inserir um nome válido, a aplicação irá salvar em um arquivo o nome de usuário. Este nome de usuário será utilizado na identificação das mensagens.

5.2 MENSAGEM

Na aplicação, uma mensagem (existe a classe *Mensagem* no código fonte) será uma concatenação de alguns campos (todos do tipo *String* de *Java*). A lógica para a composição das mensagens e estipulação dos campos foi inspirada nos protocolos de rede, mais especificamente, da camada de enlace, para encaminhamento de pacotes. Os campos da mensagem são os sequintes:

1 - momento: Uma *String* que representa o momento da criação da mensagem, mas essa *String* não tem nenhum formato textual de data, mas sim um número inteiro (*long*) que representa a hora atual do sistema *Android* em um dado celular convertido no sistema operacional *UNIX* (01 janeiro – 1969) até o momento atual. Em aplicações *Android*, obtemos este dado capturando o retorno do método estático *System.currentTimeMilis()* no momento em que o usuário digita um texto para algum contato e clica em enviar. Após a obtenção deste *long*, o transformamos para *String* e já temos o campo *momento*.

2 - MAC do remetente: Uma *String* contendo o endereço MAC do *hardware* de *bluetooth* presente no aparelho. Obtemos este dado através do método *getAdress()* da classe *BluetootDevice.*

3 - MAC ou nome do destinatário: Uma *String* contendo o MAC ou o nome de usuário do destinatário.

4 - info: Uma *String* contendo o texto a ser lido pelo remetente. Este texto será colhido da caixa de texto na tela de conversa com destinatário e passará por uma filtragem para substituir alguns caractéres ou *substrings* que serão de controle da aplicação.

5 - *status*: Uma *String* contendo apenas um caractére. Os possíveis valores desta *String* são apenas “0”, indicando que a mensagem ainda não foi recebida pelo destinatário, ou “1” indicando que a mensagem já foi recebida pelo destinatário.

Estes cinco campos são atributos da classe mensagem, e além destes, temos o atributo *sep* (uma final *String*), que é utilizado como separador de campos.

A classe Mensagem, possui o método *criarChave()* que concatena os primeiros 4 atributos de um objeto do tipo *Mensagem*, mas sempre concatenando o

atributo *sep* entre eles. O método retorna a *String* resultante desta concatenação. Tal *String* é usada como chave para o arquivo *msg.txt* e o dado, que também é uma *String*, é apenas o atributo status do objeto do tipo *Mensagem*. Como o campo *momento* é produzido no instante da criação da mensagem e capturado do aparelho onde foi criada. Para que o aplicativo funcione bem, é necessário que os relógios dos celulares estejam razoavelmente sincronizados, caso contrário, a exibição da ordem das mensagens no tempo pode não refletir a realidade.

5.2.1 Tempo de Vida da Mensagem

Um problema muito questionado durante a construção do aplicativo foi o tempo de existência de uma mensagem. Ao analisarmos a lógica do aplicativo, percebemos que a mesma mensagem pode se proliferar e habitar, ao mesmo tempo, em vários aparelhos e isso pode fazer com que o arquivo *'msg.txt'* tome proporções gigantescas em muitos aparelhos.

A mensagem se espalha na tentativa de alcançar o destinatário, e neste processo ela pode seguir (de salto em salto) em direções que nunca irão interceptar o destinatário, mas também pode seguir na direção do destinatário e alcançá-lo. Quando uma mensagem chega ao destinatário, o aplicativo (no celular do destinatário) muda o *status* da mensagem para ”1”, e a mesma mensagem, mas agora com *status* ”1”, se espalha tentando afetar suas cópias que estejam com *status* ”0”. Este processo é importante, pois o tempo de expiração da mensagem foi implementado em 3 níveis. No primeiro nível (com maior durabilidade) estão as mensagens que pertencem ao próprio usuário, ou seja, aquelas que são conversas do usuário com algum de seus contatos. No segundo nível ficam as conversas entre outros usuários e que estejam com status ”0”. No terceiro nível (com menor durabilidade) ficam as conversas entre outros usuários e que estejam com status ”1”.

Para estipular o tempo de durabilidade destes níveis, foi considerada a variação nas capacidades de armazenamento dos dispositivos e então foi utilizada a memória disponível no aparelho para definir os tempos. Ficou definido que, cada

byte de memória disponível equivale a 16 milissegundos de durabilidade para as mensagens do 1º nível, 4 milissegundos para as do 2º nível e 1 milissegundo para as do 3º . Assim, um aparelho com maior capacidade pode armazenar as mensagens por mais tempo e as mensagens do 3º nível, ao perambularem na rede, podem trazer suas cópias para o nível 3 que expira muito mais rapidamente do que as do nível 2.

Tal implementação permite ao aplicativo proteger-se de si mesmo quanto ao consumo de armazenamento, pois quando o dispositivo estiver com pouco espaço disponível, as mensagens vão expirar mais rapidamente, e o arquivo *'msg.txt'* irá diminuir de tamanho.

5.2.2 Criptografia

Com mensagens de vários usuários presentes em um mesmo aparelho, torna-se evidente a necessidade de criptografar as mensagens com o intuito de garantir a confidencialidade das mensagens. Foram utilizadas funções de criptografia já presentes no *Java*, juntamente com uma solução própria que não será revelada neste trabalho, para assim garantir maior segurança aos possíveis usuários do aplicativo na prática, pois a criptografia torna o texto ininteligível para aqueles que não tenham acesso às convenções combinadas. Obviamente foi utilizada a técnica de descriptografia, que, quando aplicada em um texto criptografado produz o texto original, para que o usuário possa ler suas conversas.

O processo de criptografar e descriptografar demanda muito processamento, por isso foi aplicada apenas no campo *info* da mensagem, que é o texto que o usuário digita.

5.3 TELAS

Nas aplicações *Android*, cada atividade, geralmente está atrelada a uma tela. Em nosso aplicativo, as classes que representam as telas, estendem da classe *Activity*.

Uma *activity* é um componente de aplicativo que fornece uma tela com a qual os usuários podem interagir para fazer algo, como discar um número no telefone, tirar uma foto, enviar um *e-mail* ou ver um mapa. Cada atividade recebe uma janela que exibe a interface do usuário. Geralmente, a janela preenche a tela, mas pode ser menor que a tela e flutuar sobre outras janelas.

Para criar uma atividade, é preciso criar uma subclasse de *Activity* (ou uma respectiva subclasse existente). Na subclasse, é preciso implementar um método de retorno de chamada que o sistema chama quando ocorre a transição entre os diversos estados de seu ciclo de vida, como na criação, interrupção, retomada ou destruição da atividade. Os dois métodos mais importantes de retorno de chamada são:

onCreate()

O sistema o chama ao criar a atividade e, na implementação, é preciso inicializar os componentes essenciais da atividade. E, fundamentalmente, é quando se deve chamar *setContentView()* para definir o layout da interface do usuário da atividade.

onPause()

O sistema chama esse método como o primeiro indício de que o usuário está saindo da atividade (embora não seja sempre uma indicação de que a atividade será destruída). É quando geralmente se deve confirmar qualquer alteração que deva persistir além da sessão do usuário atual (porque o usuário pode não retornar).

Há outros métodos de retorno de chamada do ciclo de vida que se pode usar para oferecer uma experiência fluida ao usuário entre atividades e manipular interrupções inesperadas que venham a parar ou até a destruir a atividade [11].

5.3.1 Tela de Cadastro

Esta tela possui apenas quatro componentes visuais dispostos verticalmente: dois *Textview* que apenas exibem texto para orientar o usuário, uma *Edittext* que é uma caixa de texto onde o utilizador fornece o nome de usuário para a aplicação e um *button* que é um botão onde o usuário toca para se cadastrar.

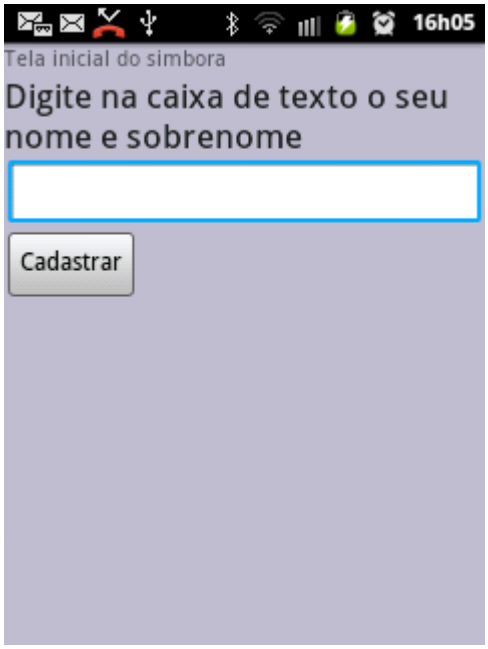

Figura 9 Tela de Cadastro

Após o toque no botão cadastrar o sistema chama um método de validação e, caso o nome for válido, ele é guardado no arquivo *'pri.txt'*. Esta tela só aparece na primeira execução do aplicativo.

5.3.2 Tela Principal

Esta tela fornece informações e botões para o usuário interagir e abrir outra tela. No topo da tela tem um *Textview* que descreve o estado do *Bluetooth,* que pode ser visível (com fundo azul), invisível (com fundo amarelo) ou desligado (com

fundo vermelho). Logo abaixo temos outro *Textview* que diz quantas conexões *bluetooth* estão ocorrendo em um dado momento. Mais abaixo temos quatro botões:

- botão com a legenda “exibir meu MAC”

Após tocar neste botão, será exibida parte do MAC do aparelho em uma caixa de exibição desaparece após alguns segundos.

- botão com a legenda “contatos“

Ao tocar neste, abre-se a tela de contatos.

- botão com a legenda “ligar bluetooth”

Ao tocar neste, o aplicativo solicita ao *Android* a ativação do *bluetooth*, e este, por sua vez, solicitará ao usuário a permissão para ligar o *bluetooth*.

- botão com a legenda “tornar visível”

Após o toque neste botão, o aplicativo solicita ao *Android* a ativação da visibilidade do *bluetooth* e este, novamente pede ao usuário a sua permissão.

Mais abaixo tem um *TextView* que informa se está ocorrendo descoberta de dispositivos e exibe também os nomes dos celulares encontrados. Em seguida tem outro *TextView* que, se houverem conexões no momento, informa quais *threads* estão dormindo (esperando para executar o método *merge()*) e qual está executando o método *merge().*

Mais abaixo, outro *TextView* informa (se houverem) os nomes dos dispositivos conectados. Por fim, um *TextView* com fundo preto e letras brancas, exibe informações de diferentes *threads*, que informam tentativas de conexões, fim de uma conexão exceções e etc..

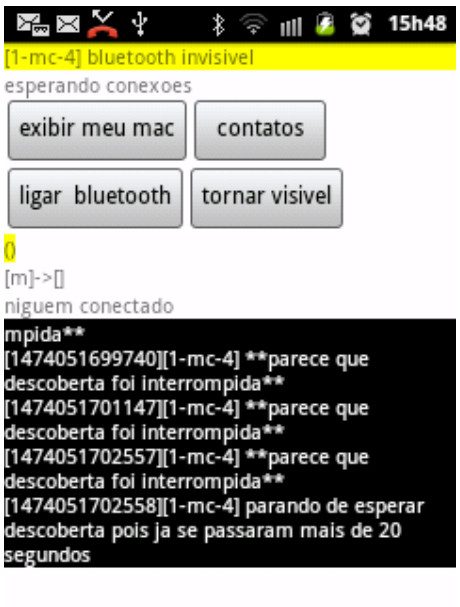

Figura 10 Tela Principal

Na classe que representa a tela principal existem várias variáveis que são usadas em outras classes, inclusive em classes que representam outras telas.

5.3.3 Tela de Contatos

Nesta, um *ListView* exibe os contatos. Em cada item do *ListView* é exibido o nome do contato, o final do seu *MAC* e, se houver, a ultima frase da conversa. Ao tocar em uma opção, será aberta a tela para conversar com o contato escolhido.

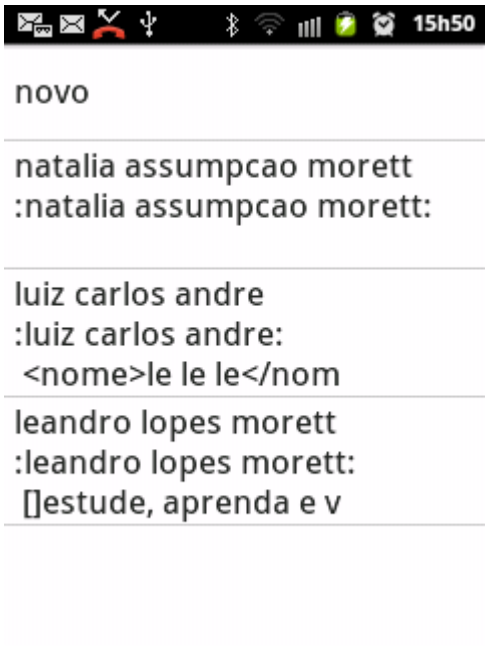

Figura 11 Tela de Contatos

A exibição dos contatos está em ordem alfabética decrescente, ordenado pelos nomes dos contatos.

5.3.4 Tela Incluir Novo Contato

Um *TextView* no topo da tela instrui o usuário a escrever o nome do contato no *EditText* que fica abaixo. Mais abaixo o botão *"incluir "* possibilita gravar o contato no arquivo *ctt.txt*.

Figura 12 Tela Incluir Novo contato

Como visto na seção 4.3.2 onde falamos do arquivo *'ctt.txt'*, será inserido o nome como chave e também o nome como dado, pois não temos ainda o MAC do contato.

No momento em que é salvo um novo contato, o aplicativo cria uma mensagem automática para ser guardada no arquivo *'msg.txt'* e ser enviada para possíveis aparelhos conectados. Esta mensagem é de controle e não será visualizada. Ela contém no campo *info* uma concatenação (em *Java*, concatena-se *Strings* com o operador '+') de *Strings "<nome>"+nome_de_usuario+"</nome>"*, onde *nome_de_usuario* é o nome que está no arquivo *'pri.txt'* conforme explicado anteriormente na seção 4.3.1.

Caso a mensagem chegue ao aparelho do contato, o aplicativo, no aparelho do contato, irá salvar automaticamente no seu arquivo *'ctt.txt'* este nome de usuário enviado e o MAC que também estará presente na mensagem no campo "MAC *do remetente*"' conforme explicado na seção 5.2 onde falamos da mensagem.

5.3.5 Tela de Conversa

Na parte superior desta tela, o usuário obtém informações sobre o estado do *bluetooth*. Logo abaixo é exibido o nome do contato com o qual o usuário está a

enviar mensagens. Em seguida uma caixa de texto permite inserir a mensagem, e o botão “vai” chama o método *enviar()* da classe *TelaConversa*. A função deste método é salvar o texto no arquivo *'msg.txt'*, disponibilizar a mensagem para ser enviada para os aparelhos conectados no momento e introduzi-lá no *TextView* abaixo onde fica a conversação.

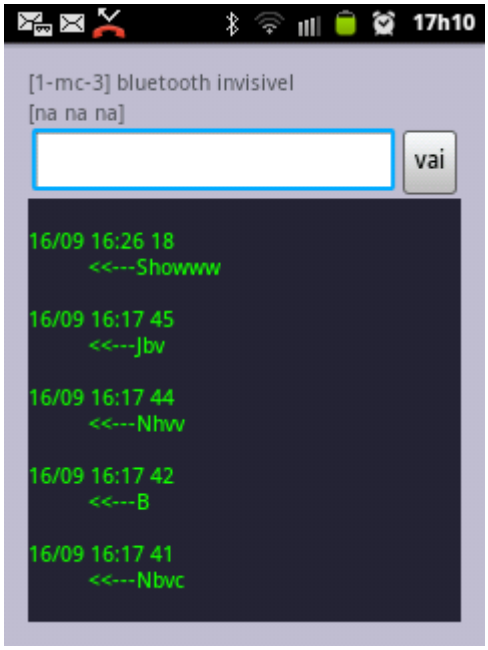

Figura 13 Tela de Conversa

Um detalhe diferente na exibição da conversa, é que esta é lida de baixo para cima, ou seja, as mensagens mais novas ficam no topo.

6 CONCLUSÕES E TRABALHOS FUTUROS

O aplicativo não foi testado com um número considerável de aparelhos, logo muitas otimizações ainda podem ser feitas ou detectadas assim que o aplicativo for utilizado por mais usuários.

Também não foi testado o aplicativo com o arquivo *msg.txt* tomando proporções maiores, o que talvez ocasione demoras no processamento e na transmissão do arquivo de um aparelho para outro, dado que o *bluetooth* não possui alta velocidade de transmissão.

Uma possível melhoria para o aplicativo, para aumentar o armazenamento de mensagens, é fazer uma versão desktop que venha interagir com os smartphones.

Um problema que talvez ocorra na utilização do aplicativo com muitos usuários, pode ser contemplado num cenário onde existe um grupo com muitas conexões e cada nó já se encontra satisfeito por conexões, ou seja, em todos aparelhos, todas as threads que esperam por conexões ou pedem conexões, já estão em execução. Neste cenário, um aparelho que acaba de se aproximar deste grupo pode não conseguir se conectar.

Uma tentativa de evitar este fato já foi adotada. Ela consiste em, antes de solicitar conexão a um vizinho, analisar se este já tem conexão com algum aparelho que já esteja conectado, e caso positivo, evita-se a conexão. Mas, não foi possível detectar se este procedimento pode resolver o problema.

Um trabalho futuro pode ser uma adaptação do aplicativo pra utilizar *wireless* no lugar do *bluetooth*, obtendo um maior alcance de conexão, porém consumindo mais bateria.

REFERÊNCIAS BIBLIOGRÁFICAS

1. Jackson,J:Interplanetaryinternet ;http://spectrum.ieee.org/telecom/internet/the-interplanetary-internet. 12/07/2016
2. Fall, K. Routing in delay tolerant network (2004).
3. Warthman, F. Delay-Tolerant Networks (DTNs): A Tutorial v1.1. [S.l.], 2003
4. Scott, K; Burleigh, S: Bundle protocol specification, 2007. http://www.ietf.org/rfc/rfc5050.txt. 22/08/2016
5. Fall, K.; Hong, W.; Madden, S. Custody transfer for reliable delivery in delay tolerant networks [S.l.], 2003
6. Cerf, V. et al. RFC4838: Delay-Tolerant Networking Architecture. [S.l.], 2007.
7. Vahdat, A.; Becker, D. Epidemic routing for partially-connected ad hoc net-works. [S.l.], 2000.
8. Miller, Michael. Descobrindo *bluetooth* / Michael Miller – Campus, 2001
9. Olhar Digital: Bluetooth: as diferenças entre as versões 2.0, 3.0, 4.0; http://olhardigital.uol.com.br/video/bluetooth-as-diferencas-entre-as-versoes-2-0-3-0-e-4-0/45294; 11/09/2016
10. Matias, F; Alcantara,G; Antunes, L;Calmon, T: Bluetooh; http://www.gta.ufrj.br/grad/10_1/bluetooth/index.html; 11/09/2016
11. https://developer.android.com/guide/components/activities.html?hl=pt-br; 29/08/2016
12. Alecrim, E: Tecnologia Bluetooth: o que é e como funciona?: http://www.infowester.com/bluetooth.php; 22/10/2016